Verifiable Outsourcing of Computations Using Garbled Onions[⋆]

Tahsin C. M. Dönmez[0000−0003−4881−3206]

Department of Future Technologies, University of Turku, Turku, Finland
tcmdon@utu.fi

Abstract. Solutions to the verifiable outsourcing problem based on Yao's Garbled Circuit (GC) construction have been investigated in previous works. A major obstacle to the practicality of these solutions is the single-use nature of the GC construction. This work introduces the novel technique *onion garbling*, which circumvents this obstacle by using only a symmetric-key cipher as its cryptographic machinery. This work also proposes a non-interactive protocol for verifiable outsourcing which utilizes the onion garbling technique. The protocol works in a 3-party setting, and consists of a preprocessing phase and an online phase. The cost of a preprocessing phase which can support up to N computations is independent of N for the outsourcing party. For the other two parties, the memory and communication cost of N-reusability is proportional to $N \cdot m$, where m is the bit-length of the input. The cost of input preparation and verification is $\mathcal{O}(m + n)$ symmetric-key cipher operations, where n is the bit-length of the output. The overall costs associated with the outsourcing party are low enough to allow verifiable outsourcing of arbitrary computations by resource-constrained devices on constrained networks. Finally, this work reports on a proof-of-concept implementation of the proposed verifiable outsourcing protocol.

Keywords: Verifiable computation · Outsourcing · Garbled onion.

1 Introduction

Verifiable outsourcing of computations involves a possibly computationally weak outsourcing party (outsourcer), and one or more worker parties (evaluators) who are possibly untrusted by the outsourcer. The outsourcer sends the inputs for the computation to the evaluator, and the evaluator sends back the result of the computation along with some additional information which enables the outsourcer to verify the received result. How much the outsourcer benefits from outsourcing depends on how much less the cost of verification is compared to the cost of performing the computation, $Cost_C$. Obviously, if the cost of verification is greater than or equal to $Cost_C$, the outsourcer would rather perform the computation itself. It is also desirable that, the cost of the verifiable computation to the evaluator is as close as possible to $Cost_C$.

[⋆] This work is supported in part by Tekes.

Current and emerging trends such as cloud computing, fog computing, and more recently, multi-access edge computing (MEC) increase the interest in finding solutions to the verifiable computation problem. Furthermore, the number of computationally weak devices have increased drastically in recent years due to the ongoing realization of the Internet of Things (IoT).

There are different approaches to the verifiable computation problem. Some solutions target specific computations, whereas others are general-purpose solutions which allow arbitrary computations. Efficiency of general purpose solutions based on probabilistically checkable proofs and fully-homomorphic encryption are not yet at the acceptable level for practical applications, and the efforts to reduce the verification cost below the cost of computation continue [16]. There are also general-purpose solutions which are based on Yao's Garbled Circuit (GC) construction [18, 19]. These solutions enjoy the non-interactivity and inherent verifiability of secure 2-party computations using GCs. In this case, the verification can be as simple as comparing a key value $k_{revealed}$ with two others k^0 and k^1, where the verification succeeds if and only if $k_{revealed} \in \{k^0, k^1\}$.[1] However, the single-use nature of the GC construction is a major obstacle to practical verifiable outsourcing using GCs. Following an evaluation, the evaluator learns either $k_{revealed} = k^0$ or $k_{revealed} = k^1$. If the same GC is reused for a second evaluation, nothing stops the evaluator from submitting the output key revealed in the first evaluation ($k_{revealed}$), even though the second evaluation revealed $k'_{revealed} \neq k_{revealed}$. Because the GC is reused, $k'_{revealed} \in \{k^0, k^1\}$, and the verifiability property is lost.

In many cases, using a new garbled circuit for each computation is not practical, as Boolean circuits for non-trivial computations can be quite large, resulting in unacceptable memory and communication costs. Achieving full reusability in GC-based protocols is possible [7, 8], however these solutions rely on (relatively) costly cryptographic techniques such as fully-homomorphic encryption and functional encryption. In case of full reusability, the cost associated with the construction of the single GC can be amortized over several computations. This work introduces the *onion garbling* technique, which provides N-reusability using only a symmetric-key cipher. In case of N-reusability, N computations still require N GC constructions, however the memory and communication costs are limited to those of a single GC, plus a term which is independent of the circuit size. This work also proposes a protocol for verifiable outsourcing of computations, which utilizes the onion garbling technique. The protocol: (1) is non-interactive in the sense that the outsourcer's online time complexity is linear in the input length; (2) does not provide privacy of inputs, outputs, or the computation; (3) works in a 3-party setting, and consists of a preprocessing phase and an online phase. The construction of a garbled onion (of N layers) is carried out in the preprocessing phase by a computationally capable party (constructor) that is trusted by the outsourcer. The cost of a preprocessing phase which can support up to N computations is $\mathcal{O}(K)$ for the outsourcing party, where K is the security parameter (key size of the GC). The memory and communication cost

[1] For simplicity, a single output wire is assumed.

for the other two parties is $\mathcal{O}(|C| \cdot R + N \cdot m \cdot K)$, where $|C|$ is the number of gates in the Boolean circuit, R is the ciphertext size for the encryption scheme used for encrypting the circuit, and m is the bit-length of the input. For large circuits, this is significantly smaller compared to the cost of constructing and transferring N independent garbled circuits, which is $\mathcal{O}(|C| \cdot R \cdot N)$. In other words, N-reusability is achieved at a cost proportional to $N \cdot m$. The cost of input preparation and verification to the outsourcer is $\mathcal{O}(m+n)$ symmetric-key cipher operations, where n is the bit-length of the output. The overall costs associated with the outsourcer are low enough to allow verifiable outsourcing of arbitrary computations by resource-constrained devices on constrained networks [11].

The rest of this paper is organized as follows. Section 2 mentions related works. Section 3 describes the onion garbling technique. Section 4 presents the proposed protocol, and Section 5 discusses its security. Section 6 reports on the implementation of the protocol. Sections 7 and 8 conclude the paper and discuss future work.

2 Related Work

Applebaum et al. [1] show how to convert the privacy property in secure multiparty computation to verifiability, using MACs and symmetric encryption. Ishai et al. [10] define partial garbling schemes, where the security goals of garbling schemes [3] are relaxed. They propose a verifiable computation scheme building upon the garble+MAC paradigm of [1], where the only private input is a one-time MAC, and the rest of the inputs are public.

Gennaro et al. [7] note that Yao's GC Construction provides a one-time verifiable computation, in addition to providing secure two-party computation. In the same work [7], they formalize the notion of verifiable computation, and propose a protocol for verifiable outsourcing of computations, which uses fully-homomorphic encryption to overcome the single-use nature of the GCs.

3 How to Garble Onions

A garbled onion is a construction built upon a stripped down version of Yao's garbled circuits. A single garbled onion, or simply onion, consists of N layers of garbled circuits. The construction features two operations *AddLayer* and *Peel*. *AddLayer* operation adds a new GC to the onion, placing it at the outermost layer. As a result, the GC which was previously at the outermost layer is wrapped and no longer exposed. Each GC added via an *AddLayer* operation is constructed in a way that depends on the GC at the layer immediately below. This dependency allows the peeling of onion layers via the *Peel* operation, which removes the outermost layer to expose the layer below.

While the description above is useful for introducing the idea, the construction would not be as useful if it were merely a collection of co-existing GCs, as suggested by the mental image of an onion. Thanks to the dependency between the consecutive layers, an onion is fully defined by the single GC at its outermost layer, the input mappings for each layer, and two seed values. What *AddLayer* actually does is the transformation of this single GC, so working on a single circuit object suffices for the construction of an onion. Similarly, the *Peel* operation is the transformation of the single GC at the outermost layer into the GC at the lower layer. Consequently, the communication of only one GC (plus the input mappings for each layer, and the two seed values) is sufficient for N verifiable computations, where N is the number of onion layers. Fig. 1 depicts a 3-layer garbled onion.

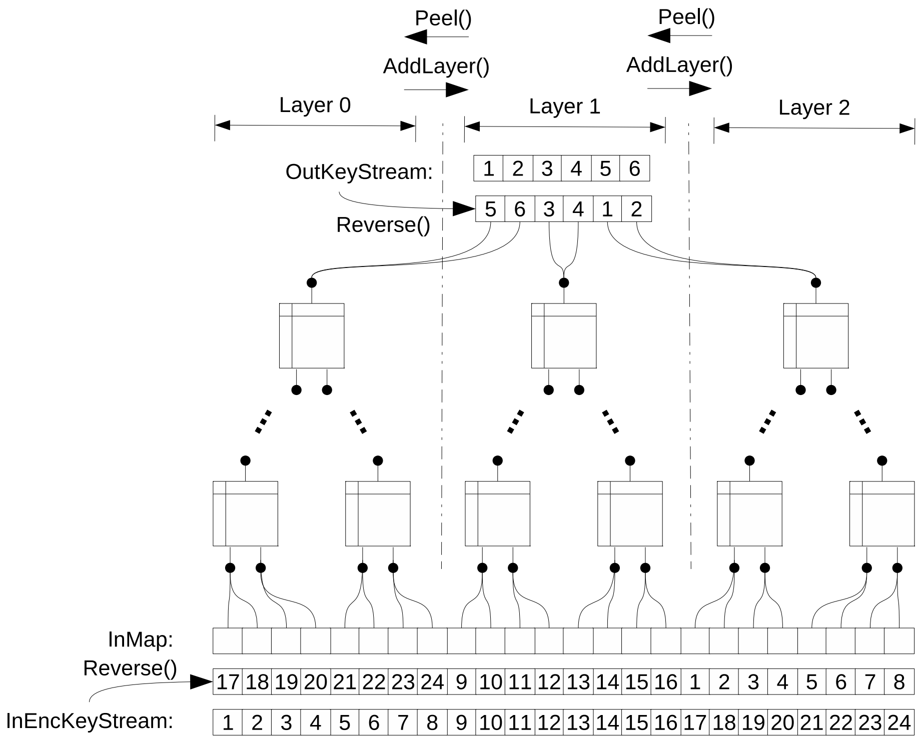

Fig. 1. A 3-layer garbled onion. For each layer, only the two input gates and the single output gate are shown. Numbers inside the boxes indicate the order of generation within the stream.

Before we move on to describe the construction, evaluation, and peeling of garbled onions in the following subsections, we explain what we mean by a stripped down version of Yao's GC.[2] In Yao's garbled circuit construction, each wire is assigned two keys k^0 and k^1 corresponding to the two possible wire values 0 and 1, respectively. For each gate g_r, the keys k_s^0, k_s^1, k_t^0, and k_t^1 are used to double-encrypt the keys k_r^0 and k_r^1, where each one of s and t is either

[2] A more formal definition will also be given in Section 5 (See Definition 1).

a gate index for a gate whose output wire is connected to an input wire of g_r (we will refer to such a gate as a *child gate* in the rest of the paper), or an index of an input wire for the circuit. The two encryption keys and the key to be encrypted are chosen respecting the structure of the truth table (TT), so that the evaluation of the garbled circuit with the garbled inputs mimics the in-the-clear evaluation with the corresponding non-garbled inputs. This process yields the encrypted truth table (ETT) for the gate (see Fig. 2). There are two components that exist in Yao's GC construction, but not in the garbled onion construction: row shuffling and row selection.[3] Yao's GC construction involves the additional step of shuffling the rows of the ETTs, so that the values on a gate's input wires cannot be inferred from the index of the row opened during the evaluation. Furthermore, evaluation of a GC requires at each gate the selection of the row which should be decrypted using the keys assigned to the gate's input wires. Row selection is achieved via trial and error (only possible if authenticated encryption is used), or via the point-and-permute technique [2]. In order to have the verifiability property, it is sufficient that an evaluation with a particular set of garbled inputs exposes one and only one of the keys for each non-input wire of the circuit. When one is concerned solely with verifiability, all requirements about privacy can be dropped, and hiding neither the orderings of ETT rows, nor the truth tables is necessary. Therefore, the GCs in a garbled onion do not have their ETT rows shuffled, and during the evaluation of an onion layer we let the in-the-clear evaluation of the circuit guide the row selections.

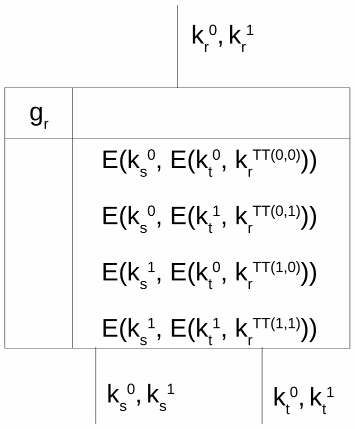
k_r^0, k_r^1
g_r
$E(k_s^0, E(k_t^0, k_r^{TT(0,0)}))$
$E(k_s^0, E(k_t^1, k_r^{TT(0,1)}))$
$E(k_s^1, E(k_t^0, k_r^{TT(1,0)}))$
$E(k_s^1, E(k_t^1, k_r^{TT(1,1)}))$
k_s^0, k_s^1
k_t^0, k_t^1

Fig. 2. An encrypted gate. Note that the rows are not shuffled.

[3] Even though the construction involves the encryption of TTs rather than full garbling thereof, it is named as *Garbled Onion* in order to make its close connection with garbled circuits apparent.

3.1 Construction

Construction of an N-layered onion involves successively (N times) adding layers via the *AddLayer* operation. The resulting garbled onion consists of the garbled circuit C_o (which is the GC at the outermost layer), the input mappings $InMap$, the seed value $SeedInEncKeyStream$ (which generates the keystream $InEncKeyStream$), and the seed value $SeedOutKeyStream$ (which generates the keystream $OutKeyStream$). $InMap$ holds the key values (both k^0 and k^1) that are assigned to the input wires of the GC at each layer, whereas $OutKeyStream$ determines for every layer the key values that are assigned to the output wires of the GC. $SeedInEncKeyStream$ and the corresponding stream $InEncKeyStream$ are optional components of the construction. $InEncKeyStream$ is a stream of encryption keys which are used for encrypting the input mappings $InMap$. Encryption of the input mappings is necessary, for example, when their transfer between two mutually trusting parties have to involve an untrusted third party as a conveyor.

All keys of a GC can be assigned values freely. Traditionally, this fact is taken advantage of in GC optimizations such as the free-XOR technique [13] and garbled row reduction [15], which make the keys in the same circuit inter-dependent, or fix some of the keys to some known value. For a GC in an onion, the keys associated with the output wires are assigned from $OutKeyStream$, however every other key can be assigned values freely. We will refer to these keys as *assignable keys*. Garbled onion construction takes advantage of this freedom by making the garbled circuits in neighbouring layers inter-dependent. During the formation of layer l via the $(l+1)^{\text{th}}$ *AddLayer* operation, ETT rows of C_o are stored in the assignable keys,[4] and the keys associated with the output wires are assigned values from $OutKeyStream$. The assignable keys associated with the input wires of the circuit are stored in $InMap$. Finally, ETT rows are overridden by encrypting the circuit with the new keys, and the new transformed C_o is obtained.

Clearly, onion garbling requires that there are enough bits in the assignable keys to store all ETT rows. The number of ETT rows for a garbled gate is equal to the number of keys associated with the input wires for single-input gates (such as NOT and $NAND$) and two-input gates (such as AND and XOR), and greater in any other case. Assuming that the size of ETT rows is equal to the key size K, if one or more gates in a circuit has more than two inputs, there won't be enough bits to store all ETT rows. There exists functionally complete sets whose elements accept either one or two inputs (e.g. $\{AND, XOR\}$, $\{AND, NOT\}$, $\{NAND\}$). Therefore no constraints are posed with regard to which functions are suitable for onion garbling and which are not.

The rules that guide the actual assignments of assignable keys can be chosen arbitrarily. In our implementation (see Section 6), ETT row size is equal to K (due to the use of one-time-pad for encrypting TT rows) and the total number of ETT rows is equal to total number of assignable keys (because we

[4] During the formation of the innermost layer (i.e. for $l = 0$), there are no ETT rows to store, so the assignable keys are assigned random values.

restrict ourselves to gates with 2 inputs). We chose to assign the ETT rows of a gate to the keys associated with the input wires of that gate. With regard to ordering, keys associated with the left wire are assigned from the upper rows of the ETT, and between two keys which are associated with the same wire, the key corresponding to value 0 is assigned from the upper row of the ETT. Fig. 3 depicts the assignment of the assignable keys according to these rules during an *AddLayer* operation.

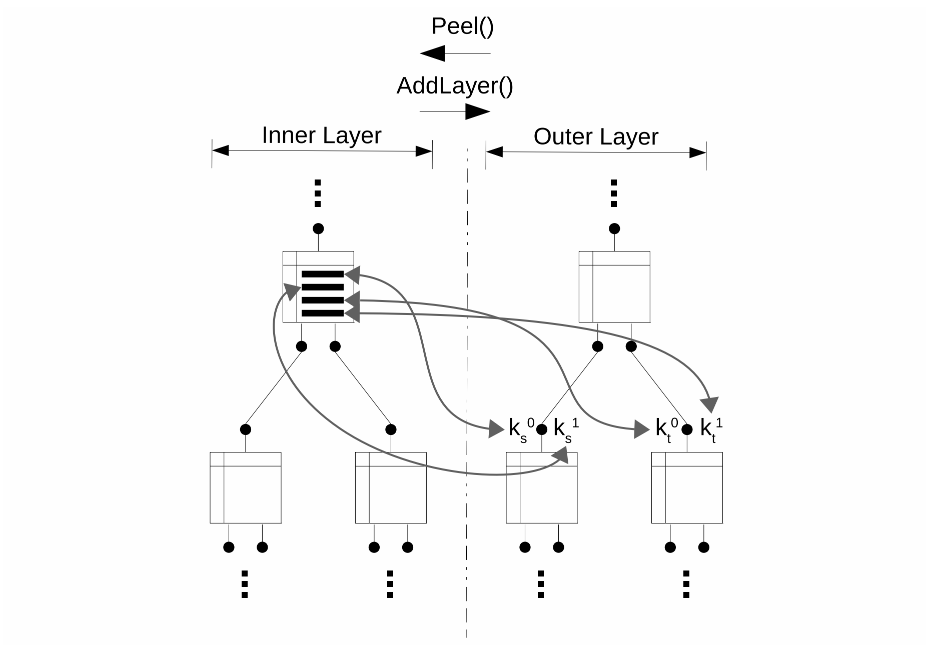

Fig. 3. The mapping between the ETT rows of a parent gate and the keys associated with the output wires of its children, during *AddLayer* and *Peel* operations.

The assignment from *OutKeyStream* to the keys associated with output wires follow a particular rule in order to allow the possibly resource-constrained outsourcer to generate, use, and discard keys one at a time as they become needed during the verifications of the computations for subsequent layers. The stream has to be accessed from opposite ends during the construction and during the computations, as the layers are traversed in reverse order: the innermost layer of the onion is the first layer that is constructed, and it is used in the very last outsourced computation. The computationally capable constructor takes the burden of reversing the order, so that the outsourcer can use the keys in the order they are generated from the stream. The same is also true for *InEncKeyStream*. During the encryption of the input mappings, *InEncKeyStream* is accessed in reverse order, so that the outsourcer can generate, use, and discard encryption keys one at a time as they become needed during the preparation of garbled inputs for subsequent layers. For both streams, the keys associated with the same layer are treated as a block, and the reversal of streams during construction occur only at the block level (See Fig. 1).

Note that the reversal of $InEncKeyStream$ and $OutKeyStream$ is not necessary if they can flow in both directions, i.e. if it is possible to efficiently obtain not only the next stream element, but also the previous one. Whether or not a stream can flow in both directions depends on the cryptographic machinery used. Using AES-CTR [5] with a combination of layer index and gate index as counter allows random access to the generated keystream, which is more than what is needed: a keystream which flows in both directions. In this case, the outsourcer is able to follow the streams backwards one AES operation at a time, so the constructor does not need to reverse them.

3.2 Evaluation

The evaluation of an onion layer, i.e. the evaluation of the garbled circuit C_o using the garbled inputs, is almost identical to regular garbled circuit evaluation except a few differences. The gates of C_o are visited in the usual order. Before a visited garbled gate is evaluated, the corresponding gate in the un-garbled circuit is evaluated, yielding an index i of the TT row, as well as the gate output read at that index. The selection of the ETT row to be decrypted is guided by the in-the-clear evaluation, i.e. the index of the ETT row that has to be decrypted is i. The key for decrypting this row is computed from the garbled outputs of its children, which are already visited and evaluated at this point. Decryption of the row yields the garbled output of the gate. Once all the gates are visited, garbled circuit evaluation is complete.

3.3 Peeling

Peeling removes the outermost layer to expose the GC at the layer below. Peeling can only be carried out when all the keys associated with the circuit's input wires are known. *Peel* operation involves going through the circuit gate by gate from inputs to outputs the same way as it is done during evaluation, but twice. During the first pass, at each gate g, the key that was not revealed during the evaluation is revealed. In order to do this, we first find an index i such that $TT(i) \neq v$, where v is the value output by g during the in-the-clear evaluation. Then, ETT row with index i is opened. This is possible because if g has a child gate, then it already went through this key-revealing process. The second pass through the circuit involves carrying out the assignments made during the $AddLayer$ operation (corresponding to the layer being peeled) in reverse order (See Fig. 3). Using the assignable keys associated with the output wire of gate g, where one is revealed during evaluation, and the other one is revealed during the first pass through the circuit, half of the ETT rows for the parent gate of g is recovered. The other half is recovered using the keys associated with the output wire of the other child.

The reason why the circuit is traversed twice during peeling is that the re-evaluations using the ETTs and overwriting the ETTs of parent gates cannot be done in a single pass through the circuit (from children to parent), if we insist keeping memory usage to a single circuit size. Finally, note that at each layer of the onion, exactly two decryption operations are needed per gate: one during evaluation and the other during peeling.

4 Protocol for Verifiable Outsourcing using Garbled Onions

In this section, we present a protocol for verifiable outsourcing of computations using garbled onions. The protocol works in a 3-party setting, and consists of a preprocessing phase (offline phase) and an online phase. The three parties involved are the outsourcer (a possibly computationally weak party who outsources the computations and verifies the results), the evaluator (a computationally capable untrusted party who performs the computation), and the constructor (a computationally capable trusted party who constructs the garbled onion).[5]

Before going into the details of the protocol, we briefly discuss why only the 3-party setting is considered. When the same party plays the role of both the constructor and the outsourcer, the protocol becomes applicable in the 2-party (outsourcer-evaluator) setting. In this case, it is not necessary to encrypt the input mappings or send them to the evaluator (See steps 5 and 6 of the preprocessing phase), but the outsourcer has to be capable of preparing, storing, and transferring the preprocessing material. Capable outsourcers exist, for example, in the realm of distributed computing projects such as SETI@home[6] and Folding@home[7], where computations are outsourced to CPUs and GPUs of volunteers over the Internet, and the possibility of dishonest evaluators makes verifiability desirable. The problem in this case is that the underlying onion garbling technique requires N circuit preparations for N computations, i.e. memory and communication costs are amortizable over several computations, but the computational cost is not. Therefore, the protocol does not provide practical benefits in the 2-party setting, unless the extra cost -due to the cost of preprocessing exceeding the cost of the computation- can be somehow justified.

The following subsections describe the preprocessing and online phases of the protocol.

4.1 Preprocessing Phase

This subsection describes a preprocessing phase which allows at most N verifiable outsourced computations. The constructor prepares all the preprocessing material without the involvement of the *outsourcer* and the *evaluator*, who may

[5] By (un)trusted we mean (un)trusted by the outsourcer.
[6] https://setiathome.berkeley.edu/
[7] https://foldingathome.org/

receive their share of the preprocessing material anytime before the first outsourced computation begins, and possibly at different times.

Preprocessing Phase

1. *Constructor* generates the non-garbled circuit C corresponding to the computation which will be outsourced.
2. *Constructor* generates two random seed values $SeedInEncKeyStream$ and $SeedOutKeyStream$, and uses them to initialize the streams $InEncKeyStream$ and $OutKeyStream$, respectively.
3. Let the number of input and output wires of C be m and n, respectively. *Constructor* generates the first $2 \cdot N \cdot m$ keys from $InEncKeyStream$, and the first $2 \cdot N \cdot n$ keys from $OutKeyStream$.
4. **Onion construction:** *Constructor* generates an N-layer garbled onion by N successive *AddLayer* operations (Section 3.1).
5. *Constructor* encrypts each key in the input mappings $InMap$ individually using the keys generated from $InEncKeyStream$, and obtains the encrypted input mappings $EncInMap$.
6. *Constructor* sends N, $EncInMap$, and C_o (the garbled circuit at the outermost layer) to the *evaluator*.
7. *Constructor* sends N, $SeedOutKeyStream$, and $SeedInEncKeyStream$ to the *outsourcer*.
8. *Evaluator* generates the non-garbled circuit C independently from the constructor (knowledge of the outsourced computation is sufficient for carrying out this task). Alternatively, C could be sent to the *evaluator* by the *constructor*.

4.2 Online Phase

This subsection describes the online phase for a single computation. Each one of the N possible computations follows the same steps. Parties involved in the online phase are the *outsourcer* and the *evaluator*. Both parties independently keep and maintain as internal state an index l, which is the index of the layer that will be used for the next computation. Initially, $l = N - 1$. The *outsourcer* initializes two keystreams $InEncKeyStream$ and $OutKeyStream$ with the seeds $SeedInEncKeyStream$ and $SeedOutKeyStream$, respectively. Internal state of the *outsourcer* includes, in addition to l, two key values: the last used values from each stream. Whenever the *outsourcer* needs an encryption key to decrypt an input mapping, or a key for verifying a received computation result, it simply gets the next key from the corresponding stream, and updates its internal state.

Online Phase

1. When the *outsourcer* wants to outsource a computation, it checks whether $l \geq 0$. If $l < 0$, verifiable outsourcing is not possible, and the protocol terminates. If $l \geq 0$, *outsourcer* initiates the computation by sending l to *evaluator*.

2. *Evaluator* checks whether both parties agree on the onion layer that will be used for the computation. In case of agreement, *evaluator* sends to *outsourcer* the encrypted input mappings for (only) layer l.
3. *Outsourcer* decrypts the input mappings using encryption keys generated from $InEncKeyStream$, revealing the garbled inputs g_i corresponding to its input bits b_i, as well as those corresponding to $\neg b_i$. We will refer to the latter as *unused garbled inputs* and denote them with g_i'. *Outsourcer* sends b_i and g_i to *evaluator*.
4. **Onion evaluation:** *Evaluator* evaluates the onion layer with index l using b_i and g_i (Section 3.2), and sends the computation result r (the keys associated with the output wires of the circuit) to *outsourcer*.
5. *Outsourcer* interprets and verifies the received computation result r. Let r_j be the key associated with the j^{th} output wire. For each j:
   - *Outsourcer* generates two keys from $OutKeyStream$. Let the key which is generated first be k, and the other one be k'.
   - If $r_j = k$, *outsourcer* accepts $o_j = 0$ as the j^{th} bit of the result.
   - If $r_j = k'$, *outsourcer* accepts $o_j = 1$ as the j^{th} bit of the result.
   - If $r_j \notin \{k, k'\}$, *outsourcer* concludes that *evaluator* tried to cheat, and rejects the received result r. The protocol terminates.

   Let o be the bit string whose j^{th} bit is o_j. If $r_j \in \{k, k'\}$ for all j, then *outsourcer* accepts o as the verified result of the computation.
6. If $l > 0$ (i.e. if there is a layer to peel), *outsourcer* sends the unused garbled inputs g_i' to *evaluator*.
7. **Onion peeling:** In order to prepare for the next computation, *evaluator* peels the outermost layer using g_i' (Section 3.3).
8. Both *outsourcer* and *evaluator* decrement l by one.

5 Proof of Security

This section discusses the security of the protocol with respect to the verifiability property. We follow the formalization of verifiable computation presented in [7]. The security of the protocol is expressed in Theorem 2. First, we note the differences between the "garbled" circuits in the garbled onion construction and Yao's garbled circuits, and define the former based on the differences.

Definition 1. *An onion-garbled circuit (OGC) is a construction built in the same way as a garbled circuit, with the following exceptions:*

- ***Fact 1:*** *ETT rows are not permuted for any of the gates.*
- ***Fact 2:*** *Let $k_{out,j}^{b_j}$ be the keys associated with the output wires of the circuit, where $j \in \{0, ..., n-1\}$, n is the bit-length of the output, and $b_j \in \{0,1\}$ is the value assigned to wire j during the in-the-clear evaluation. $k_{out,j}^{b_j}$ are assigned from a keystream $OutKeyStream$ generated by a stream cipher $\mathcal{SC}$ (instead of being randomly assigned).*
- ***Fact 3:*** *Rest of the keys are not necessarily randomly assigned (but assigned from the ETT rows of another OGC, as described in Section 3.1).*

The following lemma is intuitively clear, so it is stated without a proof.

Lemma 1. *Yao's GC construction is still correct when all ETT row permutations are the identity permutation, and when the keys are chosen arbitrarily (rather than randomly).*

Theorem 1. *An OGC provides one-time secure verifiable computation.*

Proof. (sketch) We argue the one-time secure verifiable property of an OGC, based on that of a GC. Gennaro et al. [6, Theorem 3] show that Yao's garbled circuit scheme is a one-time secure verifiable computation scheme.[8] Their proof depends only on the correctness of Yao's garbled circuit construction, and not on its privacy. By Lemma 1, an OGC is also correct. The proof for one-time secure verifiable property of Yao's GC construction can be informally expressed as follows: in order to cheat successfully, the evaluator must either correctly guess a key which was not revealed during evaluation, or break the encryption scheme used for encrypting the circuit. It is the possibility of guessing the keys that requires our attention, because selection of keys differ between a GC and an OGC. Following a computation, the evaluator learns that one of the two keys associated with output wire j is $k_{out,j}^{b_j}$. If the stream cipher $\mathcal{SC}$ used for generating $OutKeyStream$ is secure, the revealed keys do not give an adversarial evaluator non-negligible advantage for guessing $k_{out,j}^{1-b_j}$. Therefore the one-time secure verifiable property of an OGC can be argued along the same lines as for a GC, except instead of relying on the fact that keys are chosen randomly, one has to consider the security of $\mathcal{SC}$, and rely on the resulting pseudorandomness of $OutKeyStream$. Intuitively, the enabling property behind the verifiability provided by GCs is that the evaluator is able to open at most one row from each gate, given inputs for a single computation, and this property holds for both GCs and OGCs despite their differences (Facts 1-3).

Theorem 2. *The Protocol for Verifiable Outsourcing using Garbled Onions (Section 4) provides up to N verifiable computations, if the garbled onion constructed in the preprocessing phase has N layers.*

Proof. (sketch) First, we observe that a new OGC is constructed for each of the N layers. By Theorem 1, each OGC provides one-time secure verifiable computation. However, the OGCs are constructed in an interdependent fashion, so it is necessary to show that previous computations do not compromise the verifiability of later computations. Consider the $(N-l)^{\text{th}}$ computation which uses layer l of the garbled onion. In order to cheat successfully, the evaluator must correctly guess a key which was not revealed during evaluation. But different from the case in Theorem 1, the evaluator is in possession of all the keys in $OutKeyStream$ which are associated with layers $l_i > l$, in addition to the keys revealed during the current computation ($k_{out,j}^{b_j}$). If the stream cipher used for generating $OutKeyStream$ is secure, knowledge of these keys do not give an adversarial evaluator non-negligible advantage for guessing $k_{out,j}^{1-b_j}$. Finally, we note

[8] This justifies the previously mentioned *inherent verifiability* claim regarding secure 2-party computations using GCs.

that the keys $k_{out,j}^{1-b_j}$ for layer l are revealed to the evaluator (Online Phase, Step 6) only after the verification for layer l is completed (Online Phase, Step 5).

6 Implementation

This section introduces FairEnough [4], a proof-of-concept implementation of the onion garbling technique and the protocol for verifiable outsourcing described in Section 4. Each of the three parties involved in the protocol are implemented in their own classes (`Outsourcer`, `Evaluator`, `Constructor`), and they communicate via TCP sockets to run the protocol. The implementation is based on Fairplay [14], which dates back to 2004. Several CG optimizations (e.g. free-XOR [13], GRR2 [17], FleXOR [12], half gates [20]) have been developed since that time, and these optimizations are not included in Fairplay.[9] As mentioned in Section 3.1, onion garbling takes a different approach compared to these optimizations, and Fairplay was a suitable starting point for our implementation due to being uncluttered with incompatible optimizations. The main functionality kept from the original Fairplay project is the generation of circuit objects from SFDL programs via the SFDL compiler, circuit optimizer, and SHDL parser. SFDL programs describe a 2-party computation, where both parties (referred to as Alice and Bob in Fairplay) may have inputs and outputs. In case of outsourcing, we assume that only the outsourcing party has inputs and outputs. The outsourced computation is described as an SFDL program with only `BobInput` and `BobOutput`, which represent the outsourcer's inputs and the outputs, respectively.

7 Conclusion

This work tackled an efficiency issue related to the use of garbled circuits for verifiable computations, which arise from the single-use nature of a garbled circuit. The onion garbling technique was introduced, which leverage the freedom in the assignment of keys during the construction of a garbled circuit, in order to encode many garbled circuits into a single one. A protocol for verifiable computations, which utilize the onion garbling technique, was proposed. The protocol achieves N-reusability in the sense that the memory and communication cost of N verifiable computations is significantly less compared to the trivial solution, which involves the construction and transfer of N distinct garbled circuits. But most importantly, the costs incurred on the outsourcing party is sufficiently small to allow verifiable outsourcing by a resource-constrained device on a constrained network.

8 Future Work

This work addressed a major obstacle to practical verifiable outsourcing using GCs, namely the single-use nature of the constructed GCs. Another major

[9] For a more recent, optimized framework for circuit garbling, see for example [9].

obstacle to practical verifiable outsourcing using GCs is the size and runtime inflation due to the conversion to Boolean circuit. Running both branches for each branching, and running each loop the maximum number of times it can be run give the resulting circuit its obliviousness property. Obliviousness is essential when privacy is desired, but not when the only security goal is verifiability. In the proposed protocol, the evaluator is provided with the knowledge of the inputs and the computation, and is able to perform the in-the-clear computation. In this case, it would be a good trade-off to let go off the obliviousness property, if in turn the size and runtime inflation could be eliminated. This suggests moving away from the circuit model of computation, but of course one would want to keep the verifiability which comes with the garbled circuit evaluations. Investigation of the applicability of onion garbling beyond the circuit model of computation is left as future work.

The proof-of-concept implementation introduced in Section 6 was useful in writing parts of this work, as following working source code provides some degree of reassurance against possible errors and omissions during the textual description of the ideas. We believe that the codebase could prove to be a useful resource for the motivated reader as well, for clarifying ambiguities and filling in gaps, caused by weaknesses in our writing. We note however that, the implementation was not meant to assess feasibility, and deployment on an actual resource-constrained device is left as future work.

Acknowledgements This work is supported in part by Tekes under the project Wireless for Verticals (WIVE). WIVE is a part of 5G Test Network Finland (5GTNF). We thank Ethiopia Nigussie for encouraging us to think about applications of secure multiparty computation in the context of the Internet of Things and Edge Computing. Finally, the author would like to thank the anonymous reviewers for their valuable comments.

References

1. Applebaum, B., Ishai, Y., Kushilevitz, E.: From secrecy to soundness: Efficient verification via secure computation. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) Automata, Languages and Programming. pp. 152–163. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
2. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols. In: Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing. pp. 503–513. STOC '90, ACM, New York, NY, USA (1990). https://doi.org/10.1145/100216.100287, http://doi.acm.org/10.1145/100216.100287
3. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 784–796. CCS '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2382196.2382279, http://doi.acm.org/10.1145/2382196.2382279

4. Dönmez, T.C.M.: Fairenough. https://gitlab.utu.fi/tcmdon/Fairenough (July 2018)
5. Dworkin, M.: Recommendation for block cipher modes of operation. methods and techniques. Tech. rep., National Inst Of Standards And Technology Gaithersburg MD Computer Security Div (Dec 2001), http://www.dtic.mil/docs/citations/ADA400014
6. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. Cryptology ePrint Archive, Report 2009/547 (2009), https://eprint.iacr.org/2009/547
7. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Rabin, T. (ed.) Advances in Cryptology – CRYPTO 2010. pp. 465–482. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
8. Goldwasser, S., Kalai, Y., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing. pp. 555–564. STOC '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2488608.2488678, http://doi.acm.org/10.1145/2488608.2488678
9. Huang, Y., Shen, C.h., Evans, D., Katz, J., Shelat, A.: Efficient secure computation with garbled circuits. In: Jajodia, S., Mazumdar, C. (eds.) Information Systems Security. pp. 28–48. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
10. Ishai, Y., Wee, H.: Partial garbling schemes and their applications. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) Automata, Languages, and Programming. pp. 650–662. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
11. Keranen, A., Ersue, M., Bormann, C.: Terminology for constrained-node networks. RFC 7228, RFC Editor (May 2014). https://doi.org/10.17487/RFC7228, http://www.rfc-editor.org/info/rfc7228
12. Kolesnikov, V., Mohassel, P., Rosulek, M.: Flexor: Flexible garbling for xor gates that beats free-xor. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology – CRYPTO 2014. pp. 440–457. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
13. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free xor gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) Automata, Languages and Programming. pp. 486–498. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
14. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay—a secure two-party computation system. In: Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13. pp. 20–20. SSYM'04, USENIX Association, Berkeley, CA, USA (2004), http://dl.acm.org/citation.cfm?id=1251375.1251395
15. Naor, M., Pinkas, B., Sumner, R.: Privacy preserving auctions and mechanism design. In: Proceedings of the 1st ACM Conference on Electronic Commerce. pp. 129–139. EC '99, ACM, New York, NY, USA (1999). https://doi.org/10.1145/336992.337028, http://doi.acm.org/10.1145/336992.337028
16. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy. pp. 238–252 (May 2013). https://doi.org/10.1109/SP.2013.47
17. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Matsui, M. (ed.) Advances in Cryptology – ASIACRYPT 2009. pp. 250–267. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

18. Yao, A.C.: Protocols for secure computations. In: 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982). pp. 160–164 (Nov 1982). https://doi.org/10.1109/SFCS.1982.38
19. Yao, A.C.C.: How to generate and exchange secrets. In: 27th Annual Symposium on Foundations of Computer Science (sfcs 1986). pp. 162–167 (Oct 1986). https://doi.org/10.1109/SFCS.1986.25
20. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015. pp. 220–250. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)