Fast Garbling of Circuits Under Standard Assumptions*

Shay Gueron† Yehuda Lindell‡ Ariel Nof‡ Benny Pinkas‡

July 16, 2015

Abstract

Protocols for secure computation enable mutually distrustful parties to jointly compute on their private inputs without revealing anything but the result. Over recent years, secure computation has become practical and considerable effort has been made to make it more and more efficient. A highly important tool in the design of two-party protocols is Yao's garbled circuit construction (Yao 1986), and multiple optimizations on this primitive have led to performance improvements of orders of magnitude over the last years. However, many of these improvements come at the price of making very strong assumptions on the underlying cryptographic primitives being used (e.g., that AES is secure for related keys, that it is circular secure, and even that it behaves like a random permutation when keyed with a public fixed key). The justification behind making these strong assumptions has been that otherwise it is not possible to achieve fast garbling and thus fast secure computation. In this paper, we take a step back and examine whether it is really the case that such strong assumptions are needed. We provide new methods for garbling that are secure solely under the assumption that the primitive used (e.g., AES) is a pseudorandom function. Our results show that in many cases, the penalty incurred is not significant, and so a more conservative approach to the assumptions being used can be adopted.

1 Introduction

In the setting of secure computation, a set of parties with private inputs wish to compute a joint function of their inputs, without revealing anything but the output. Protocols for secure computation guarantee *privacy* (meaning that the protocol reveals nothing but the output), *correctness* (meaning that the correct function is computed), and *independence of inputs* (meaning that parties are not able to make their inputs depend on the other parties' inputs). These security guarantees are to be provided in the presence of adversarial behavior. There are two classic adversary models that are typically considered: *semi-honest* (where the adversary follows the protocol specification but may try to learn more than is allowed from the protocol transcript) and *malicious* (where the adversary can run any arbitrary polynomial-time strategy in its attempt to breach security).

Garbled circuits. One of the central tools in the construction of secure two-party protocols is Yao's garbled circuit [17, 21]. The basic idea behind Yao's protocol is to provide a method of computing a circuit so that values obtained on all wires other than circuit-output wires are never

*This work was funded by the European Research Council under the ERC consolidators grant agreement n. 615172 (HIPS), and under the European Union's Seventh Framework Program (FP7/2007-2013) grant agreement n. 609611 (PRACTICE).

†University of Haifa and Intel Haifa, Israel. email: shay@math.haifa.ac.il.

‡Bar-Ilan University, Israel. email: lindell@biu.ac.il, nofdinar@gmail.com, benny@pinkas.net.

revealed. For every wire in the circuit, two random or garbled values are specified such that one value represents 0 and the other represents 1. For example, let i be the label of some wire. Then, two values k_i^0 and k_i^1 are chosen, where k_i^b represents the bit b. An important observation here is that even if one of the parties knows the value k_i^b obtained by the wire i, this does not help it to determine whether $b = 0$ or $b = 1$ (because both k_i^0 and k_i^1 are independently distributed). Of course, the difficulty with such an idea is that it seems to make computation of the circuit impossible. That is, let g be a gate with incoming wires i and j and output wire ℓ. Then, given two random values k_i^b and k_j^c, it does not seem possible to compute the gate because b and c are unknown. We therefore need a method of computing the value of the output wire of a gate (also a random value k_ℓ^0 or k_ℓ^1) given the value of the two input wires to that gate.

In short, this method involves providing "garbled computation tables" that map the random input values to random output values. However, this mapping should have the property that given two input values, it is only possible to learn the output value that corresponds to the output of the gate (the other output value must be kept secret). This is accomplished by viewing the four possible inputs to the gate, k_i^0, k_i^1, k_j^0, and k_j^1, as encryption keys. Then, the output values k_ℓ^0 and k_ℓ^1, which are also keys, are encrypted under the appropriate keys from the incoming wires. For example, let g be an OR gate. Then, the key k_ℓ^1 is encrypted under the pairs of keys associated with the input values $(1,1)$, $(1,0)$ and $(0,1)$. In contrast, the key k_ℓ^0 is encrypted under the pair of keys associated with $(0,0)$.

Fast garbling and assumptions. Today, secure computation is fast enough to solve numerous problems in practice. This has been achieved due to multiple significant efficiency improvements that have been made on the protocol level, and also due to *garbled circuits themselves*. Many of the optimizations to garbled circuits – described below – come at the price of assuming strong assumptions on the security of the cryptographic primitives being used. For example, the free-XOR technique requires assuming circular security as well as a type of correlation robustness [7], the use of fixed-key AES requires assuming that AES behaves like an ideal cipher [5],[1] reductions in the number of encryption operations from 2 to 1 per entry in the garbled gate requires correlation robustness (when a hash function is used) and a related-key assumption (when AES is used).

Typically, the use of less standard cryptographic assumptions is accepted where necessary, especially in areas like secure computation where the costs are in general very high. However, in practice, solid cryptographic engineering practices dictate a more conservative approach to assumptions. New types of elliptic curve groups are not adopted quickly, people shy away from non-standard use of block ciphers, and more. This is based on sound principles, and on the understanding that deployed solutions are very hard to change if vulnerabilities are discovered. In the field of secure computation, the willingness to take any assumption that enables a faster implementation stands in stark contrast to standard cryptographic practice. In this paper, we propose to pause, take a step back, and ask the question *how much do non-standard assumptions really cost us and are they justified*. We remark, for just one example, that practitioners have warned against assuming that AES is an ideal cipher, due to related key weaknesses that have been found; see e.g., [4, 6]. As in most situations, if the benefit is huge, then more flexibility with respect to the assumptions is justified, whereas if the gains are smaller then a more cautious approach is taken.

The focus of this paper is to study *how much is really gained by relying on non-standard*

[1] To be more accurate, it suffices to assume that AES behaves like an ideal random permutation oracle, even when it is keyed with a public and known fixed key. This is a weaker assumption than the full ideal-cipher assumption. For sake of brevity, we use the term "ideal cipher" throughout, and our intention is the assumption specified here.

assumptions and to provide optimizations that require assuming nothing more than that *AES behaves like a pseudorandom function.*

Garbled circuit optimizations. Before proceeding to describe our work, we present an overview of the most important efficiency improvements to garbled circuits:

- **Point and permute [19]:** In order to prevent the garbled circuit evaluator from knowing what it is evaluating, the original construction randomly permuted the ciphertexts in each garbled gate. Then, when computing the garbled circuit, the evaluator tries each ciphertext in the gate until one correctly decrypts (this requires an additional mechanism to ensure that only one ciphertext decrypts to a valid value). On average, this means that 2.5 entries need to be decrypted per gate (where each costs 2 decryptions). The point and permute method assigns a random permutation or signal bit to each wire, that determines the order of the garbled gate. Then, the encryption of a garbled value includes the bit needed to enable direct access to the appropriate entry in the garbled table (given two garbled values and the two associated bits). This reduces the number of entries to decrypt to 1 (and thus 2 actual decryptions).

- **Free XOR [14]:** The garbled circuit construction involves carrying out encryptions at every gate in the circuit, and storing 4 ciphertexts. The free-XOR method enables the computation of XOR gates *for free* (the computation requires only computing 1-2 XORs, and no ciphertexts need be stored). This is achieved by choosing a fixed random mask Δ and making the garbled values on every wire have fixed difference Δ (i.e., for every i, the garbled values are k_i^0 and $k_i^1 = k_i^0 \oplus \Delta$, where k_i^0 is random). In many circuits, the number of XOR gates is very large and so this significantly reduces the cost (e.g., in the AES circuit there are approximately 7,000 AND gates and 25,000 XOR gates; in a 32x32 bit multiplier circuit there are approximately 6,000 AND gates and 1,000 XOR gates [1]).

  We remark that the free-XOR method is patented, and as such, its use is restricted [15].

- **Reductions in garbled-circuit size [19, 20, 22]:** Historically, the most expensive part of any secure protocol was the cryptographic operations. However, significant algorithmic improvements to secure protocols together with much faster implementations of cryptographic primitives (e.g., due to better hardware) have considerably changed the equation. In many cases, communication can be the bottleneck and thus reducing the size of the garbled circuit is of great importance. In [19], a method for reducing the number of garbled entries in a table from 4 to 3 was introduced; this is referred to as 4-to-3 garbled row reduction (or 4-3 GRR). This improvement is achieved by "forcing" the first ciphertext to be 0 (by setting the appropriate garbled value on the output wire so that the ciphertext becomes 0). In [20], polynomial interpolation was used to further reduce the number of ciphertexts to just 2; this is referred to as 4-to-2 garbled row reduction (or 4-2 GRR).

  Importantly, 4-3 GRR is compatible with free-XOR since only one output garbled value needs to taken as a function of the input values (and the other garbled value can be set according to the fixed Δ). In contrast, 4-2 GRR is *not* compatible with free XOR. Nevertheless, in recent work, a new method called half gates [22] reduces the number of ciphertexts in AND gates from 4 to 2, while maintaining compatibility with free XOR (in fact, half gates *only work* with free XOR).

- **Number of encryptions [18]:** Classically, each entry in a garbled gate contains the encryption of one of the output garbled values under *two* input garbled values, and thus requires *two encryptions*. In [18], it was proposed to use a hash function as a type of key-derivation function, and to encrypt by hashing both input garbled values together and XORing the result with the output garbled value. This is secure in the random-oracle model, or under a "correlation-robustness" assumption [12]. This reduces the number of operations from 2 to 1. (Note however that two AES operations are typically much faster than a single hash operation, especially when utilizing the AES-NI instruction.)

- **Fixed-key AES and use of AES-NI [5]:** AES-NI is a set of CPU instructions that are now part of the Intel architecture. They allow AES computations to be carried out at incredibly fast rates, especially in modes of operation that can be highly pipelined. AES-NI offers instructions for encryption/decryption and for the AES key expansion.

  However, since typical AES usages encrypt multiple blocks with a single key, the key expansion instructions do not highly optimize this part of the processing, and the key schedule generation routine is relatively expensive (compared to encryption/decryption). More importantly, pipelining cannot be carried out between different keys. When computing garbled circuits, 4 different keys are used in *every gate*, requiring many key schedules to be computed and preventing the use of pipelining.

  In light of this, [5] proposed a method of using AES that is secure in the ideal-cipher model (i.e., assuming that AES for every fixed key behaves as a purely random permutation). The method uses a fixed key for AES, applies AES on a combination of the input garbled values, and XORs the result with appropriate output garbled value. This reduces the number of AES computation to 4 per gate. Furthermore, since a fixed key is used, only one key schedule needs to be computed for the entire circuit, and the encryptions within a gate can be fully pipelined. This led to an extraordinary speedup in the computation of garbled circuits, as demonstrated in the JustGarble implementation [5].

We stress that there have been a very large number of works that have provided highly significant efficiency improvements to *protocols* that use garbled circuits. However, our focus here is on improvements to garbled circuits themselves.

Our results. We construct fast garbling methods solely under the assumption that AES behaves like a pseudorandom function. In particular, we do not use fixed-key AES (since this requires assuming that AES is an ideal cipher), and we use two AES encryptions per entry in the garbled gates (since using just one encryption requires some sort of related-key security assumption). In addition, we do not use free-XOR (since this requires circularity). However, this does enable us to use 4-to-2 row reduction. In brief, we construct the following:

- **Fast AES-NI without fixing the key:** We show that, in addition to pipelining encryptions, it is also possible to pipeline the key schedule of AES-NI, in order to achieve very fast garbling times without using fixed-key AES or any other non-standard AES variant. Namely, the key schedule processing of different keys can be pipelined together, so that the amortized effect of key scheduling on Yao garbling is greatly reduced. Our experiments (described below) show that this and other optimizations of AES operations have become so fast that the benefits of using fixed-key AES are almost insignificant. Thus, in contrast to current popular belief, in most cases *fixed-key AES is* ***not*** *necessary for achieving extremely fast garbling.*

- **Low-communication XOR gates:** Over the past years, it has become apparent that in secure protocols, communication is far more problematic than computation. The free-XOR technique is so attractive exactly because it requires no computation but also *no communication* for XOR gates. We provide a new garbling method for XOR gates that requires storing only a single ciphertext per XOR gate; our technique is inspired by the work of [13]. The computational cost is 3 AES computations for garbling the gate, and 1-2 AES computations for evaluating it. (This overhead is for an optimized garbling method that we show. We first present a basic scheme requiring 4 AES computations for garbling and 2 computations for evaluation.)

- **Fast 4-2 row reduction:** As we have mentioned, once we no longer use the free-XOR technique, we are able to use 4-2 GRR on the non-XOR gates. However, the method of [20] that uses polynomial interpolation is rather complex to implement (requiring finite field operations and precomputation of special constants to make it fast). In addition, even working in $GF(2^n)$ Galois fields and using the PCLMULQDQ Intel instruction, the cost is still approximately half an AES computation. We present a new method for 4-2 row reduction that uses a few XOR operations only, and is trivial to implement.

We implemented these optimizations and compared them to JustGarble [5]. There is no doubt that the cost of garbling and evaluation is higher using our method, since we have to run AES key schedules, and we pay for computing XOR gates. However, we show that within protocol executions, the difference is insignificant. We demonstrate this running Yao's protocol for semi-honest adversaries which has nothing but oblivious transfer (for which we use the fast OT extensions of [2]), garbled-circuit evaluation and computation, and communication.[2]

Experimental results. We ran Yao's protocol for semi-honest adversaries inside Amazon EC2. The details of the results can be found in Section 6. The results show removing the ideal-cipher assumption does not noticeably affect the performance of the protocol. Furthermore, in many scenarios, such as small circuits, large inputs, or relaively slow communication channels, garbling under the most conservative assumption (the existence of PRFs) performs on par with the most efficient garbling methods.

Patent-free garbled circuits. Another considerable advantage of using our method for computing XOR gates with low communication is that it does not rely on the free XOR technique and thus is not patented. Since patents in cryptography are typically an obstacle to adoption, we believe that the search for efficient garbling techniques that are not patented is of great importance.

2 Fixed-Key AES vs. Regular AES

Background. Bellare et al. introduced the use of fixed-key AES in garbling schemes and implemented the JustGarble library [5]. This significantly speeds up garbling since the AES key schedule (which is quite expensive) need not be computed at every gate. Note that when constructing the garbled circuit four key schedules are required for every gate, and when evaluating the circuit two

[2]We do not count the base OTs of the OT extension since these would outweigh everything else, and can anyway be precomputed. Our aim here is to see the effect of the change in the garbled circuits and our tests are under optimal conditions for JustGarble-type constructions [5]. For the same reason, we do not look at the effect inside protocols for malicious adversaries since all of the other work will clearly outweigh any additional costs in garbling.

key schedules are required for every gate. This is very expensive. In addition, JustGarble utilizes the AES-NI instruction set and pipelining, significantly reducing the cost of the AES computations.

Despite its elegance, the use of fixed-key AES requires the assumption that AES behaves like an ideal cipher. In particular assuming that even given the secret key, AES behaves like a random permutation. This is a very strong assumption, and one that has been brought into question regarding AES specifically by the block-cipher research community; see, for example, [4,6]. Clearly, the acceptance of this assumption in the context of secure computation and garbling is due to the perceived very high cost of garbling in any other way. However, the comparisons carried out in [5] to prior work are to Kreuter et al. [16] who use AES-256 using AES-NI *without pipelining*, and to Huang et al. [11] who use a hash function only. Thus, it is unclear how much of the impressive speedup achieved by [5] is due to the savings obtained by using fixed-key AES, and how much is due to the other elements that they included (pipelining of the AES computations in each gate, optimizations to the circuit representation, and more).

In this section, we show that it is possible to achieve fast garbling without using fixed-key AES and thus without resorting to the assumption that AES is an ideal cipher. We stress that some penalty will of course be incurred since the AES key schedule *is* expensive. Nevertheless, we show that *when properly implemented*, in many cases the penalty is not significant and it suffices to use regular AES. The goal is to make the performance depend on the throughput (which is excellent when pipelining is used) and not on the latency of a single computation. This goal can be achieved rather easily for the AES encryption alone, but we also achieve the more challenging task of pipelining the key schedule as well as the encryption.

2.1 Pipelining the Garbling

The standard way of garbling a gate uses *double encryption*. Specifically, given 4 keys $k_i^0, k_i^1, k_j^0, k_j^1$ for the input wires and 2 keys k_ℓ^0, k_ℓ^1 for the output wire, four computations of the type $E_{k_i^a}(E_{k_j^b}(k_\ell^c))$ are made, for varying values of $a, b, c \in \{0,1\}$. Observe that since $E_{k_j^b}(k_\ell^c)$ must be known before encrypting again with k_i^a, this means that the encryptions cannot be pipelined. This makes a huge difference when using the AES-NI chip, since the cost of 8 pipelined encryptions is only slightly more than the cost of a single non-pipelined encryption.[3] We therefore garble an AND gate in a way that enables pipelining. This is easily achieved by applying a pseudorandom function F (which will be instantiated as AES) to the gate index and appropriate signal/permutation bits. This ensures independence between all values. For example, an AND gate where both signal bits are 0 can be garbled as follows:

$$F_{k_i^0}(g\|00) \oplus F_{k_j^0}(g\|00) \oplus k_\ell^0 \qquad F_{k_i^0}(g\|01) \oplus F_{k_j^1}(g\|01) \oplus k_\ell^0$$
$$F_{k_i^1}(g\|10) \oplus F_{k_j^0}(g\|10) \oplus k_\ell^0 \qquad F_{k_i^1}(g\|11) \oplus F_{k_j^1}(g\|11) \oplus k_\ell^1$$

Needless to say, 4-to-3 GRR can also be carried out by setting $k_\ell^0 = F_{k_i^0}(g\|00) \oplus F_{k_j^0}(g\|00)$ meaning that the first ciphertext equals 0 and so need not be stored. Observe here that there are 8 encryptions. However, all inputs are known and therefore it is possible to pipeline these computations.

Note that it is essential to take both signal bits as part of the input of F. Otherwise, the scheme is not secure. To understand this, assume that the gate was garbled as in the example above but

[3] Concretely, on a Haswell processor, 8 pipelined AES computations costs approximately 77 cycles, whereas one non-pipelined AES computation costs approximately 70 cycles.

without using signal bits (e.g., the value $F_{k_i^1}(g)$ is used instead of $F_{k_i^1}(g\|10)$), and assume that the evaluator holds the keys k_i^0, k_j^0. The evaluator will compute k_ℓ^0, but then it will also be able to compute $F_{k_i^1}(g)$ and $F_{k_j^1}(g)$ using the second and the third garbled entries (without learning the values of k_j^1 or k_i^1). Now, the evaluator would be able to compute k_ℓ^1 as well, using the fourth garbled entry. Taking both signal bits as part of F's input prevents this from happening, as the evaluator cannot learn $F_{k_i^1}(g\|11)$ and $F_{k_j^1}(g\|11)$.

2.2 Pipelining Key Schedule and Encryption

The computations that are needed for garbling and evaluating garbled circuits are as follows:

- KS4_ENC8: This consists of the computation of 4 AES key schedules from 4 different keys. The resulting keys are then used to encrypt 8 blocks (each key is used for encrypting 2 blocks). This is used for garbling AND (and other non-XOR) gates.

- KS2_ENC2: This consists of the computation of 2 AES key schedules from 2 different keys. The resulting keys are then used to encrypt 2 blocks (each key is used for encrypting 1 block). This is used for evaluating all gates.

- KS4_ENC4: This consists of the computation of 4 AES key schedules from 4 different keys. The resulting keys are then used to encrypt 4 blocks (each key is used for encrypting 1 block). This is used for garbling XOR gates according to our new XOR-gate garbling scheme described in Section 3.2.

A naïve software implementation approach for these computations would use the appropriate sequence of calls to a "key expansion" function, and to a "block encryption" function. To estimate the performance of that approach, we use, as a comparison baseline, the OpenSSL (1.0.2) library, running on the Haswell architecture.[4]

Software running on this processor can use the AES hardware support, known as AES-NI (see [8, 9] for details). On this platform, a call (using the OpenSSL library) to an AES key expansion consumes 149 CPU cycles. A call to an (ECB) encryption function to encrypt 2/4/8 blocks consumes approximately 70+ cycles (explanation is provided below). However, OpenSSL's API does not support ECB encryption with multiple key schedules. For example, this implies that KS4_ENC4 would required 4 calls to the key expansion function, followed by 4 calls to an ECB encryption, each one applied to a single (16B) block. The resulting performance of KS4_ENC4, KS4_ENC8, KS2_ENC2 obtained by calling OpenSSL's functions (namely "aesni_set_encrypt_key" and "aesni_ecb_encrypt") is summarized in middle column of Table 1 at the end of this section.

Our goal is to optimize the computations of KS4_ENC4, KS4_ENC8, KS2_ENC2, and alleviate the overhead imposed by the frequent key replacements. We achieve our optimization by: **(a)** interleaving the encryption of independent blocks; **(b)** optimizing the key expansion; **(c)** aggressive interleaving of the operations; **(d)** building an API that allows for encrypting with multiple key schedules. The details are as follows.

Interleaved encryption. AES encryption on a modern processor is accelerated by using the AES-NI instructions (see [8, 9]). Assuming that the cipher key is expanded to a key schedule of

[4] Haswell (resp., Broadwell) is an Intel Architecture Codename of a recently announced 4th (resp., 5th) Generation Intel® Core™ Processor. For short, we refer to them simply as Haswell (resp., Broadwell).

11 round keys, RK[j], j=0, ..., 10, AES encryption of a 16 bytes block X is achieved by the code sequence

```
XMM = X XOR RK [0]
for j = 1, 2, ..., 9
    XMM = AESENC XMM, RK [j]
end
XMM = AESENCLAST XMM, RK [10]
output XMM
```

If the latency of the AESENC/AESENCLAST instructions is L cycles, then the above flow can be completed in $1 + 10L$ cycles. However, if the throughput of AESENC/AESENCLAST is 1 (i.e., pipelining can be used and the processor can dispatch AESENC/AESENCLAST every cycle, if the data is available), and the computations encrypt more than one block, the software can interleave the AESENC/AESENCLAST invocations. This achieves a higher computational throughput, compared to the single block encryption. Furthermore, the AESENC/AESENCLAST instructions can be applied to any round key, *even those generated by different key schedules*. For example, 2 blocks X and Y, can be encrypted, with 2 different key schedules $KS1$ and $KS2$, by the following code sequence:

```
XMM1 = X XOR RK1 [0]
XMM2 = Y XOR RK2 [0]
for j = 1, 2, ..., 9
    XMM1 = AESENC XMM1, RK1 [j]
    XMM2 = AESENC XMM2, RK2 [j]
end
XMM1 = AESENCLAST XMM1, RK1 [10]
XMM2 = AESENCLAST XMM2, RK2 [10]
output XMM1, XMM2
```

These computations can be completed within $10L + 1$ cycles (the 2 XOR's of the whitening step can be executed in one cycle). Similarly, encrypting 4/8 blocks with an interleaved software flow could (theoretically) terminate after $(2 + 10L + 3)$ $/(4 + 10L + 7)$ cycles. (This idealized estimation assumes that the round keys are fetched from the processor's cache, and ignores the cost of loading/storing the input/output blocks. We point out that the code sequence indeed closely approaches the theoretical performance, under these assumptions.) These computations are dominated only by the throughput of AESENC/AESENCLAST. We note that $L = 7$ on Haswell, and the AESENC/AESENCLAST throughput is 1. As can be seen,

Optimized key expansion. We were able to optimize the computation of AES key expansion so that it computes (and stores) an AES128 key schedule in 96 cycles on Haswell, which is 1.55 times faster than the code used by OpenSSL on the same platform. The details of this optimization are quite low-level, and we provide here only some high-level details. The a full set of key expansion code options, was contributed to the NSS open source library, and can be found in [10].

The AES-NI instruction set includes instructions that facilitate key expansion. For the encryption key schedule, the relevant instruction is AESKEYGENASSIST. However, this instruction does not provide a throughput of 1 and is significantly slower than the AESENC and AESENCLAST operations (the reason being that key schedules are typically run only once and so the cost involved in optimizing this instruction was not justified). We observe that the key schedule consists of S-box

substitutions together with rotation and XOR operations. Likewise, the last round of AES costs of S-box substitutions together with shift rows (and key mixing, which can be effectively cancelled by using a round key of all-zeroes). Thus, the use of AESKEYGENASSIST can be replaced by a combination of a shuffle followed by an AESENCLAST invocation, to isolate the S-box transformation.[5] The shuffle is carried out efficiently using the PSHUFB instruction which also has a throughput of 1. We therefore obtain that the key schedule can be "simulated" using much faster instructions. Additional optimizations can be obtained by judicious usage of the available instructions to generate efficient sequences. We give one example. Consider the following portion of the AES key schedule flow (where RCON = Rcon[i/4]):

```
w[i] = w[i-4] xor Sbox(RotWord(w[i-1])) xor RCON
w[i+1] = w[i-3] xor w[i-4] xor Sbox(RotWord(w[i-1])) xor RCON
w[i+2] = w[i-2] xor w[i-3] xor w[i-4] xor Sbox(RotWord(w[i-1])) xor RCON
w[i+3] = w[i-1] xor w[i-2] xor w[i-3] xor w[i-4] xor Sbox(RotWord(w[i-1])) xor RCON
```

As explained above, the S-box substitution can be isolated by a shuffle followed by AESENCLAST, and if we place (duplicated) RCON in the second operand of AESENCLAST, the addition of RCON is also done by AESENCLAST. The arrangement and XOR-ing of the "words" can be implemented by the following straightforward flow:

```
vpslldq $4, \reg, %xmm3
vpxor %xmm3, \reg, \reg
vpslldq $4, %xmm3, %xmm3
vpxor %xmm3, \reg, \reg
vpslldq $4, %xmm3, %xmm3
vpxor %xmm3, \reg, \reg
```

However, the same functionality can be achieved by a shorter, 4 instructions, flow, as follows:

```
vpsllq $32, \reg, %xmm3
vpxor %xmm3, \reg, \reg
vpshufb (con3), \reg,  %xmm3
vpxor %xmm3, \reg, \reg
      (with the value con3 = -1,-1,-1,-1,-1,-1,-1,-1,4,5,6,7,4,5,6,7)
```

In this way, the 3 shuffles and 3 xors of the straightforward flow, can be replaced by shorter and faster 1 shift, 1 shuffle and 2 flows. With our optimizations, we were able to write a key expansion code that computes and stores an AES128 key schedule in 96 cycles on Haswell (i.e., 1.55 times faster than OpenSSL).

Multiple aggressive interleaving. A higher degree of optimization can be achieved by interleaving the computations of multiple key expansions. This helps in partially alleviating the key expansion's dependency on the latency of AESENC. For example, our code for expanding 2 key schedules consumes 124 cycles (on Haswell), which is significantly less than two independent (without interleaving) key schedules, that are 2×96 cycles. We applied this technique to obtain an optimized KS4_ENC4 and KS4_ENC8 implementation. For KS2_ENC2, optimization is achieved by "mixed interleaving" of the key expansion and the encryptions.

[5] AESENCLAST is used since the last round of AES does not include the MixColumns operation, which is a part of all other rounds and therefore run in the AESENC instruction but not in AESENCLAST.

The performance of the optimized KS4_ENC4, KS4_ENC8, and KS2_ENC2 is summarized in the right column of Table 1.

Computation	Naïve implementation	Optimized imp.
	(cycles)	(cycles)
KS4_ENC4	703	240 (asm - 220)
KS4_ENC8	729	256 (asm - 248)
KS2_ENC2	338	182 (asm - 180)

Table 1: The performance (in cycles) of KS4_ENC4, KS4_ENC8 and KS2_ENC2, measured on the Haswell architecture. The naïve implementation is the result of calling the OpenSSL (1.0.2) functions for AES key expansion and for ECB encryption. The performance of the optimized implementations is of C code (compiled using gcc), and of hand-written assembly implementations (marked with "asm").

2.3 Experimental Results

The results in Table 2 show the garbling and evaluation time of 1000 AES circuits, using the free-XOR technique and 4-to-3 row reduction (as used by JustGarble, in order to make a fair comparison). All methods use pipelining of the encryptions (the last two entries do not use a fixed key and therefore use the encryption pipelining method described in Section 2.1). The last entry is based on using also the key scheduling pipelining method described in Section 2.2. The table shows the results for garbling and evaluating the circuit (with the garble time first, following by the evaluation time). We stress that the times in Table 2 are for 1000 computations; thus, a single garbling of the AES circuit using our pipelined key schedule takes 0.74 milliseconds only.

The results were achieved on the Amazon EC2 c4.large Linux instance (with a 2.59GHz Intel Xeon E5-2666 v3 Haswell processor, a single thread, and 3.75GiB of RAM).

Algorithm	**Time**
Fixed-key AES (JustGarble)	399 / 191
Regular AES, pipelined encryption	1578 / 732
Regular AES, pipelined enc. + key schedule	743 / 389

Table 2: Garbling and evaluation times for the AES circuit 1000 times (in milliseconds)

The results show that pipelining the key schedule as well as the encryptions (3rd row) reduces time by more than 50% over pipelining the encryptions only (2nd row). Fixed-key AES (1st row) does provide a significant improvement and the best performance. However, the gain in using fixed-key AES is not overwhelming, since, as we will show later on, in many settings the main cost of secure computation is no longer the garbling itself. Namely, although AES takes 86% more time without a fixed key, the objective difference is just 0.344 milliseconds. Thus, when run in a protocol that includes communication, this additional time makes almost no difference. We demonstrate this in our experiments described in Section 6.

3 Garbling under a Pseudorandom Function Assumption Only

3.1 Background

The free-XOR technique [14] is one of the most significant optimizations of garbling. When using this technique, the garbling and evaluation of XOR gates are essentially for free, requiring only two XOR operations for garbling and one for evaluating. In addition, no garbled table is used, thereby significantly reducing communication. However, the free-XOR technique also requires non-standard assumptions. Specifically, when using this method, there is a global offset Δ, and on every wire a single random k_i^0 is chosen and the other key is always set to $k_i^1 = k_i^0 \oplus \Delta$. This is secure in the random oracle model [14] or under a circular-secure correlation robustness or related key assumption [7] (correlation robustness is formalized for hash functions whereas related key security is for encryption or pseudorandom functions). The need for this assumption is due to the fact that when a global offset is used, multiple encryptions are made under related keys $k_a, k_a \oplus \Delta$, $k_b, k_b \oplus \Delta$, and so on. In addition, since these keys are used to encrypt the values k_c and $k_c \oplus \Delta$, the ciphertext is related to the secret key which is exactly circular security. We remark that at some additional cost, the circularity assumption can be removed using the FleXOR technique [13]. However, the correlation robustness/related key assumption remains.[6]

We next show that it is possible to efficiently garble a circuit using a pseudorandom function only. We first show a basic version of our garbling scheme, where the garbled table for a XOR gate contains a *single* ciphertext and requires 4 pseudorandom function operations for garbling (instead of 8 for an AND gate), and 2 for evaluation. We then show an optimized version that reduces the number of PRF invocations to 3 calls for garbling, and 1-2 calls for evaluation. The overhead of these schemes is definitely beyond that of the free XOR technique. However, as we will show, the techniques are a considerable improvement over the naive method of computing XOR like an AND gate, they enable the usage of 4-2 garbled row reduction (4-2 GRR), and within protocols (where communication and other factors become the bottleneck) they perform well.

3.2 Garbled XOR With a Single Ciphertext

In order to prove security solely under the assumption that the primitive used is a pseudorandom function, all the garbled values on all wires should be independently chosen. Thus, for all pairs of wires i and j, the keys $k_i^0, k_i^1, k_j^0, k_j^1$ should be *independent* and either uniformly distributed or pseudorandom. It will be useful to equivalently write the keys as k_i^0 and $k_i^1 = k_i^0 \oplus \Delta_i$, and k_j^0 and $k_j^1 = k_j^0 \oplus \Delta_j$ where Δ_i, Δ_j are random independent strings.

We use the point-and-permute method, described briefly in the introduction. In order to avoid confusion, we will call the bit used to determine the order of the ciphertexts in the garbling phase the permutation bit (since it determines the random order), and we call the bit that is viewed by the evaluator when it evaluates the circuit the signal bit (since it signals which ciphertext is to be decrypted). We denote the permutation bit on wire i by π_i, and we denote the signal bit on wire i by λ_i. Observe that if the evaluator has bit v_i on wire i (for which it does not know the value), then it holds that $\lambda_i = \pi_i \oplus v_i$. Thus, if $\pi_i = 0$, then the evaluator will see $\lambda_i = v_i$, and if $\pi_i = 1$ then the evaluation will see $\lambda_i = \overline{v}_i$ (its complement). Since π_i is random, this reveals nothing about v_i whatsoever.

[6] We note that garbling with hash functions is much slower than with AES, especially when an AES-NI supporting architecture is utilized. Thus, related-key security for AES is required, which is a less than ideal assumption.

We now describe the basic XOR gate garbling method that uses just a single ciphertext. The method requires 4 calls to a pseudorandom function for garbling, but as we have seen, this is inexpensive using AES-NI. (We remark that AND gates are garbled in the standard way, independently of this method.) Denote the input wires to the gate by i, j and denote the output wire from the gate by ℓ. We therefore have input keys $k_i^0, k_i^0 \oplus \Delta_i$ and $k_j^0, k_j^0 \oplus \Delta_j$. According to the above, we denote by π_i, π_j the permutation bits on wires i and j respectively. As we will see, the keys on the output wire will be determined as a result of the garbling method. The method for garbling a XOR gate with index g is as follows:

- **Step 1 – translate input keys on wire i:** We first translate the input keys on wire i into new keys $\tilde{k}_i^0, \tilde{k}_i^1$ by applying a pseudorandom function to the gate index. That is, we compute $\tilde{k}_i^0 = F_{k_i^0}(g)$ and $\tilde{k}_i^1 = F_{k_i^1}(g)$, where g is the gate index.

- **Step 2 – set offset of wire ℓ:** The offset of wire ℓ (the output wire) is set to be the offset of the translated values on wire i, namely $\Delta_\ell = \tilde{k}_i^0 \oplus \tilde{k}_i^1$. (Observe that if the same wires are input to multiple gates, independent values will be obtained since the pseudorandom function is applied to the gate index.)

- **Step 3 – translate input keys on wire j:** Next, we translate the input keys on wire j so that they too have the offset Δ_ℓ (this will enable the output key to be computed by XORing the translated input keys, as in the free XOR technique). Thus, we set $\tilde{k}_j^{\pi_j} = F_{k_j^{\pi_j}}(g)$ and $\tilde{k}_j^{\bar{\pi}_j} = \tilde{k}_j^{\pi_j} \oplus \Delta_\ell$, where π_j is the random permutation bit that is associated with the bit 0 on wire j.

- **Step 4 – compute output keys on wire ℓ:** Since $\tilde{k}_i^0 \oplus \tilde{k}_i^1 = \tilde{k}_j^0 \oplus \tilde{k}_j^1 = \Delta_\ell$, we can now use the free-XOR technique and can define $k_\ell^0 = \tilde{k}_i^0 \oplus \tilde{k}_j^0$ and $k_\ell^1 = k_\ell^0 \oplus \Delta_\ell$. (Observe that $\tilde{k}_i^1 \oplus \tilde{k}_j^1 = k_\ell^0$ as required, since $\tilde{k}_i^0 \oplus \tilde{k}_i^1 = \tilde{k}_j^0 \oplus \tilde{k}_j^1$ implies that $\tilde{k}_i^0 \oplus \tilde{k}_j^0 = \tilde{k}_i^1 \oplus \tilde{k}_j^1$. In addition, $\tilde{k}_i^0 \oplus \tilde{k}_j^1 = \tilde{k}_i^1 \oplus \tilde{k}_j^0 = k_\ell^1$ as required, since in both cases the result of the XOR is $\tilde{k}_i^0 \oplus \tilde{k}_j^0 \oplus \Delta_\ell = k_\ell^0 \oplus \Delta_\ell = k_\ell^1$.)

- **Step 5 – set the ciphertext:** Given k_i^a for any $a \in \{0,1\}$, the evaluator can easily compute $\tilde{k}_i^a$. In addition, if it has $k_j^{\pi_j}$ (as we show, this can be implicitly determined from the signal bit λ_i), then it can compute $\tilde{k}_j^{\pi_j}$. The only problem is that it *cannot* compute $k_j^{\bar{\pi}_j}$ since it does not know Δ_ℓ (and furthermore Δ_ℓ cannot be revealed). Thus, the ciphertext for the gate is set to $T = F_{k_j^{\bar{\pi}_j}}(g) \oplus \tilde{k}_j^{\bar{\pi}_j}$. Now, given $k_j^{\bar{\pi}_j}$ it is possible to compute $\tilde{k}_j^{\bar{\pi}_j}$ as well (but without $k_j^{\bar{\pi}_j}$ the value remains hidden since it is masked by a pseudorandom function keyed by $k_j^{\bar{\pi}_j}$).

In order to evaluate a XOR gate g with ciphertext T, given a key k_i on wire i and a key k_j on wire j, the evaluator simply needs to compute $\tilde{k}_i = F_{k_i}(g)$ and either $\tilde{k}_j = F_{k_j}(g)$ if it has signal bit 0, or $\tilde{k}_j = F_{k_j}(g) \oplus T$ if it has signal bit 1. Then, the key on the output wire is obtained by finally computing $k_\ell = \tilde{k}_i \oplus \tilde{k}_j$.

The computational cost of garbling the gate is 4 pseudorandom function computations, and the computational cost of evaluating the gate is 2 pseudorandom function computations. Most significantly, the gate table includes only a *single* ciphertext.

Reducing the number of PRF calls to 3. Observe that the pseudorandom function is used to ensure independence of the Δ values between different gates. If we were to just take $\Delta_\ell = k_i^0 \oplus k_i^1$, then the output Δ from two different gates with the same input wire i would be the same, and once again correlation robustness or a related key assumption would be needed. Thus, it is necessary to compute $\tilde{k}_i^0 = F_{k_i^0}(g)$ and $\tilde{k}_i^1 = F_{k_i^1}(g)$. In contrast, $\tilde{k}_j^0$ *can* be taken to simply be k_j^0 and the pseudorandom function computation is not needed. This is because Δ_ℓ is fixed independently of wire j. Using this method, we can reduce the computational cost of garbling the XOR gate from 4 pseudorandom function computations to 3 pseudorandom function computations (and the computational cost of evaluating the gate is decreased from 2 to either 1 or 2 PRF computations). The proof of security with this optimization is somewhat more involved, and we therefore prove it separately from the basic scheme.[7]

Garbling NOT Gates. When using free XOR, it is possible to efficiently garble NOT gates by simply defining them to be XOR with a fixed wire that is always given value 1. Since the XOR gates are free, this is highly efficient. However, since we are not using free XOR, a different method needs to be found. Fortunately, NOT gates can still be computed for free, and with no additional assumption. In order to see this, let g be a NOT gate with input wire i and output wire j, and let k_i^0, k_i^1 be the garbled values on wire i. Then, we simply define $k_j^0 := k_i^1$ and $k_j^1 := k_i^0$. During the garbling of the circuit, any gates receiving wire j as input will used these "reversed" values. Furthermore, when evaluating the circuit, if the value k_i^0 is given on wire i, then the result of the NOT gate is k_j^1 which *equals* k_i^0. Thus, nothing needs to be done. This trivially preserves security since no additional information is provided in the garbled circuit.

3.3 Garbling Scheme Definitions

We use the notation of Bellare et al. [3] in which a garbling scheme consists of 4 algorithms:

- $\mathsf{Garble}(1^n, c) \to (C, e, d)$ is an algorithm that takes as input a security parameter 1^n and a description of a boolean circuit c, and returns a triple (C, e, d), where C represents a garbled circuit, e represents input encoding information (i.e., all the keys on the input wires) and d represents output decoding information (i.e., all the keys on the output wires).

- $\mathsf{Encode}(e, x) \to X$ is a function that takes as input encoding information e and input x and returns garbled input (i.e., the keys on the input wires that are associated with the concrete input x).

- $\mathsf{Eval}(C, X) \to Y$ is a function that takes as input a garbled circuit C and garbled input X and returns garbled output Y (i.e., the keys on the output wires that are associated with the concrete output $y = c(x)$).

- $\mathsf{Decode}(Y, d) \to y$ is a function that takes as input decoding information d and garbled output Y and returns the real output y of the circuit.

[7] It may be tempting to propose that one of k_i^0, k_i^1 will also remain the same; i.e., set $\tilde{k}_i^{\pi_i} = k_i^{\pi_i}$ and $\tilde{k}_i^{\bar{\pi}_i} = F_{k_i^{\bar{\pi}_i}}(g)$. However, in this case, if the evaluator happens to have $k_i^{\bar{\pi}_i}$ and $k_j^{\bar{\pi}_j}$ then it can compute $T \oplus F_{\tilde{k}_i^{\bar{\pi}_i}}(g) \oplus F_{\tilde{k}_j^{\bar{\pi}_j}}(g)$. Note that $T = F_{k_j^{\bar{\pi}_j}}(g) \oplus \tilde{k}_j^{\bar{\pi}_j} = F_{k_j^{\bar{\pi}_j}}(g) \oplus \tilde{k}_j^{\pi_j} \oplus \Delta_\ell = F_{k_j^{\bar{\pi}_j}}(g) \oplus \tilde{k}_j^{\pi_j} \oplus k_i^{\pi_i} \oplus F_{k_i^{\bar{\pi}_i}}(g)$ and so the result obtained by the evaluator is $\tilde{k}_j^{\pi_j} \oplus k_i^{\pi} = \tilde{k}_j^{\pi_j} \oplus \tilde{k}_i^{\pi}$. If these keys are used in other gates, then an attacker sees the XOR of two keys and encryptions computed with each key separately. This is once again a related-key type assumption.

A secure garbling scheme should satisfy three security requirements:

- **Privacy**: The triple (C, X, d) should not reveal any information about x that cannot be learned directly from $c(x)$. More formally, there exists a simulator $\mathcal{S}$ that receives input $(1^n, c, c(x))$ and outputs a simulated garbled circuit with encoding and decoding information that is indistinguishable from (C, X, d) generated using the real garbling functions $\mathsf{Garble}(1^n, c)$ and $\mathsf{Encode}(e, x)$. Observe that $\mathcal{S}$ knows the output $c(x)$ and does not know the input x.
- **Obliviousness**: (C, X) should not reveal any information about x. More formally, there exists a simulator $\mathcal{S}$ that receives input $(1^n, c)$ and outputs a simulated garbled circuit with encoding information that is indistinguishable from (C, X) generated using the real garbling functions $\mathsf{Garble}(1^n, c)$ and $\mathsf{Encode}(e, x)$. Observe that $\mathcal{S}$ here is not even given the output.
- **Authenticity**: Given (C, X) as input, no adversary should be able to produce *different* garbled output $\tilde{Y}$ that is not obtained by computing $\mathsf{Eval}(C, X)$. More formally, a probabilistic-polynomial time adversary should be able to output $\tilde{Y} \neq \mathsf{Eval}(C, X)$ such that $\mathsf{Decode}(\tilde{Y}, d) \neq \perp$, with only negligible probability.

For each security definition we define an experiment that formalizes the adversary's task. In the following, G denotes a garbling scheme that consists of the 4 algorithms stated above, and $\mathcal{S}$ denotes a simulator.

The privacy experiment $\mathsf{Expt}^{\mathrm{priv}}_{G,\mathcal{A},\mathcal{S}}(n)$**:**
1. Invoke adversary $\mathcal{A}$: compute $(c, x) \leftarrow \mathcal{A}(1^n)$
2. Choose a random $\beta \in \{0, 1\}$
3. If $\beta = 0$: compute $(C, e, d) \leftarrow \mathsf{Garble}(1^n, c)$ and $X \leftarrow \mathsf{Encode}(e, x)$
   Else: compute $(C, X, d) \leftarrow \mathcal{S}(1^n, c, c(x))$
4. Give $\mathcal{A}$ the challenge (C, X, d) and obtain its guess: $\beta' \leftarrow \mathcal{A}(C, X, d)$
5. Output 1 if and only if $\beta' = \beta$

The obliviousness experiment $\mathsf{Expt}^{\mathrm{oblv}}_{G,\mathcal{S},\mathcal{A}}(n)$**:**
1. Invoke adversary: $(c, x) \leftarrow \mathcal{A}(1^n)$
2. Choose a random $\beta \in \{0, 1\}$
3. If $\beta = 0$: compute $(C, e, d) \leftarrow \mathsf{Garble}(1^n, c)$ and $X \leftarrow \mathsf{Encode}(e, x)$
   Else: compute $(C, X) \leftarrow \mathcal{S}(1^n, c)$
4. Give $\mathcal{A}$ the challenge (C, X) and obtain its guess: $\beta' \leftarrow \mathcal{A}(C, X)$
5. Output 1 if and only if $\beta' = \beta$

The authenticity experiment $\mathsf{Expt}^{\mathrm{auth}}_{G,\mathcal{A}}(n)$**:**
1. Invoke adversary: $(c, x) \leftarrow \mathcal{A}(1^n)$
2. Compute $(C, e, d) \leftarrow \mathsf{Garble}(1^n, c)$ and $X \leftarrow \mathsf{Encode}(e, x)$
3. Give $\mathcal{A}$ the challenge (C, X) and obtain its output: $\tilde{Y} \leftarrow \mathcal{A}(C, X)$
4. Output 1 if and only if $\mathsf{Decode}(\tilde{Y}, d) \notin \{\perp, c(x)\}$

The basic non-triviality requirement for a garbling scheme, called correctness, is that for every circuit c and input $x \in \{0, 1\}^{\mathrm{poly}(n)}$, it holds that $\mathsf{Decode}(\mathsf{Eval}(C, \mathsf{Encode}(e, x), d)) = c(x)$ except with negligible probability, where $(C, e, d) \leftarrow \mathsf{Garble}(1^n, c)$.

Definition 3.1 (Garbled Circuit Security) *A garbling scheme is* secure *if it is correct, and achieves privacy, obliviousness and authenticity as follows:*

1. *A garbling scheme G achieves* privacy *if for every probabilistic polynomial-time adversary $\mathcal{A}$ there exists a probabilistic-polynomial time simulator $\mathcal{S}$ and a negligible function μ such that for every $n \in \mathbb{N}$:*
$$\Pr\left[\mathsf{Expt}^{\text{priv}}_{G,\mathcal{A},\mathcal{S}}(n) = 1\right] \leq \frac{1}{2} + \mu(n).$$

2. *A garbling scheme G achieves* obliviousness *if for every probabilistic polynomial-time adversary $\mathcal{A}$ there exists a probabilistic-polynomial time simulator $\mathcal{S}$ and a negligible function μ such that for every $n \in \mathbb{N}$:*
$$\Pr\left[\mathsf{Expt}^{\text{obly}}_{G,\mathcal{A},\mathcal{S}}(n) = 1\right] \leq \frac{1}{2} + \mu(n).$$

3. *A garbling scheme G achieves* authenticity *if for every probabilistic polynomial-time adversary $\mathcal{A}$ there exists a negligible function μ such that for every $n \in \mathbb{N}$:*
$$\Pr\left[\mathsf{Expt}^{\text{auth}}_{G,\mathcal{A}}(n) = 1\right] < \mu(n)$$

3.4 Our Garbling Scheme in Detail

In this section, we provide a full specification of our garbling scheme. In this description, we use the standard 4-3 row reduction technique. In later sections, we will incorporate our new 4-2 row reduction scheme. Our garbling scheme uses a pseudorandom function that takes an n-bit key, and has input and output of length $n+1$. That is, $F : \{0,1\}^n \times \{0,1\}^{n+1} \to \{0,1\}^{n+1}$ (formally, we consider a family of functions, where for every $n \in \mathbb{N}$ the function is of this type). We denote by $F_k(x)[1..n]$ the first n bits of the output of $F_k(x)$, and we denote by $x\|y$ the concatenation of x with y. We begin by defining the method for garbling XOR and AND gates in Figures 1 and 2 (for simplicity we only consider XOR, AND and NOT gates; the AND gate method can be extended to any gate type), and then proceed to the high-level garbling algorithm in Figure 3. Finally, we describe the encoding, evaluation and decoding algorithms.

Procedure $\mathsf{GbXOR}(k_i^0, k_i^1, k_j^0, k_j^1, \pi_i, \pi_j)$**:**

1. Set the output wire permutation bit for the bit '0': $\pi_\ell := \pi_i \oplus \pi_j$
2. Compute translated keys for wire i: $\tilde{k}_i^0 := F_{k_i^0}(g\|\pi_i)[1..n]$ and $\tilde{k}_i^1 := F_{k_i^1}(g\|\overline{\pi}_i)[1..n]$
3. Compute new offset for the output wire: $\Delta_\ell := \tilde{k}_i^0 \oplus \tilde{k}_i^1$
4. Compute translated keys for wire j and the ciphertext for this gate:
   (a) If $\pi_j = 0$, set $\tilde{k}_j^0 := F_{k_j^0}(g\|0)[1..n]$, $\tilde{k}_j^1 := \tilde{k}_j^0 \oplus \Delta_\ell$ and $T := F_{k_j^1}(g\|1)[1..n] \oplus \tilde{k}_j^1$
   (b) If $\pi_j = 1$, set $\tilde{k}_j^1 := F_{k_j^1}(g\|0)[1..n]$, $\tilde{k}_j^0 := \tilde{k}_j^1 \oplus \Delta_\ell$ and $T := F_{k_j^0}(g\|1)[1..n] \oplus \tilde{k}_j^0$
5. Compute the keys for the output wire ℓ: $k_\ell^0 := \tilde{k}_i^0 \oplus \tilde{k}_j^0$ and $k_\ell^1 := k_\ell^0 \oplus \Delta_\ell$
6. Return $(k_\ell^0, k_\ell^1, \pi_\ell, T)$

Figure 1: Garbling XOR gates

Procedure $\mathsf{GbAND}(k_i^0, k_i^1, k_j^0, k_j^1, \pi_i, \pi_j)$**:**

1. Compute $K_0 = F_{k_i^{\pi_i}}(g\|00) \oplus F_{k_j^{\pi_j}}(g\|00)$
2. Set the output wire keys and permutation bits:
   (a) If $\pi_i = \pi_j = 1$, then choose a random $k_\ell^0\|\pi_l \leftarrow \{0,1\}^{n+1}$ and set $k_\ell^1 := K_0[1..n]$
   (b) Else, set $k_\ell^0\|\pi_\ell := K_0$ and choose a random $k_\ell^1 \leftarrow \{0,1\}^n$

   Denote $K_\ell^0 = k_\ell^0\|\pi_\ell$ and $K_\ell^1 = k_\ell^1\|\bar{\pi}_\ell$.
3. Compute the gate ciphertexts: Let $g(\cdot,\cdot)$ denote the gate function. Then,

$$\begin{aligned} T_1 &= F_{k_i^{\pi_i}}(g\|01) \oplus F_{k_j^{\bar{\pi}_j}}(g\|01) \oplus K_\ell^{g(\pi_i,\bar{\pi}_j)} \\ T_2 &= F_{k_i^{\bar{\pi}_i}}(g\|10) \oplus F_{k_j^{\pi_j}}(g\|10) \oplus K_\ell^{g(\bar{\pi}_i,\pi_j)} \\ T_3 &= F_{k_i^{\bar{\pi}_i}}(g\|11) \oplus F_{k_j^{\bar{\pi}_j}}(g\|11) \oplus K_\ell^{g(\bar{\pi}_i,\bar{\pi}_j)} \end{aligned}$$

4. Return $(k_\ell^0, k_\ell^1, \pi_\ell, T_1, T_2, T_3)$

Figure 2: Garbling AND gates

The garbling algorithm $\mathsf{Garble}(1^n, c)$**:**

1. For each input wire j in c:
   (a) Choose two random keys: $k_j^0, k_j^1 \leftarrow \{0,1\}^n$
   (b) Choose a permutation bit for the bit '0': $\pi_j \leftarrow \{0,1\}$
   (c) Prepare the encoding information: $e[j,0] := k_j^0\|\pi_j$ and $e[j,1] := k_j^1\|\overline{\pi}_j$
2. In topological order, for each gate g in circuit c:
   (a) If g is a XOR gate with input wires i, j and output wire ℓ:
      i. $(k_\ell^0, k_\ell^1, \pi_\ell, T) \leftarrow \mathsf{GbXOR}(k_i^0, k_i^1, k_j^0, k_j^1, \pi_i, \pi_j)$
      ii. Set the keys on the output wire ℓ to be k_ℓ^0, k_ℓ^1 and the permutation bit to be π_ℓ
      iii. Set the garbled table for the gate: $C[g] := T$
   (b) If g is an AND gate with input wires i, j and output wire ℓ:
      i. $(k_\ell^0, k_\ell^1, \pi_\ell, T_1, T_2, T_3) \leftarrow \mathsf{GbAND}(k_i^0, k_i^1, k_j^0, k_j^1, \pi_i, \pi_j)$
      ii. Set the keys on the output wire ℓ to be k_ℓ^0, k_ℓ^1 and the permutation bit to be π_ℓ
      iii. Set the garbled table for the gate: $C[g] := (T_1, T_2, T_3)$
   (c) If g is a NOT gate with input wire i and output wire ℓ:
      i. Set $k_\ell^0 = k_i^1$ and $k_\ell^1 = k_i^0$ and set $\pi_\ell = \pi_i$
      ii. There is no garbled gate
3. For each circuit-output wire j in c, prepare the decoding information: $d[j,0] := F_{k_j^{\pi_j}}(\text{out}\|\pi_j)$ and $d[j,1] := F_{k_j^{\bar{\pi}_j}}(\text{out}\|\overline{\pi}_j)$
4. Return (C, e, d)

Figure 3: The full garbling algorithm

We now proceed to describe the encoding, evaluation and decoding algorithms. The encoding and decoding algorithms are straightforward and consist merely of mapping the plaintext bit to the garbled value and vice versa. Observe that in the evaluation algorithm we refer to the signal bit λ_i on wire i. The difference between λ_i here and π_i used in the garbling is that λ_i is the "public" signal bit that the evaluator sees. The invariant over this value is that λ_i always equals the XOR of π_i and the actual value on the wire (associated with the encoding X).

Procedure $\mathsf{Encode}(e, x)$**:**

1. For i=1 to $|x|$: $X[i] := e[i, x_i]$
2. Return X

Figure 4: The encoding algorithm

Procedure $\mathsf{Eval}(C, X)$**:**

1. For every input wire j in c, set $k_j \| \lambda_j := X[j]$
2. For each gate g in c, in topological order:
   (a) If g is a XOR gate with input wires i, j and output wire ℓ:
      i. Compute the output wire key: $k_\ell := F_{k_i}(g\|\lambda_i)[1..n] \oplus F_{k_j}(g\|\lambda_j)[1..n] \oplus \lambda_j \cdot C[g]$
      ii. Compute the output wire signal bit: $\lambda_\ell := \lambda_i \oplus \lambda_j$
   (b) If g is an AND gate with input wires i, j and output wire ℓ:
      i. Compute the output wire key and signal bit: $k_\ell \| \lambda_\ell := T \oplus F_{k_i}(g\|\lambda_i\lambda_j) \oplus F_{k_i}(g\|\lambda_i\lambda_j)$, where T is the entry $T_{\lambda_i\lambda_j}$ in $C[g]$ (note that if $\lambda_i = \lambda_j = 0$ then implicitly we define $T = 0$).
   (c) If g is a NOT gate with input wire i and output wire ℓ, then set $k_\ell := k_i$ and $\lambda_\ell = \lambda_i$
3. For each output wire j in c, set $Y[j] := F_{k_j}(\text{out}\|\lambda_j)$
4. Return Y

Figure 5: The evaluation algorithm

Procedure $\mathsf{Decode}(Y, d)$**:**

1. For i=1 to $|Y|$:
   (a) If $Y[i] = d[i, 0]$, then $y[i] := 0$
   (b) Else, if $Y[i] = d[i, 1]$, then $y[i] := 1$
   (c) Else, return $\perp$
2. Return y

Figure 6: The decoding algorithm

Correctness. We begin by demonstrating correctness. This is immediate for AND and NOT gates; we therefore show that it also holds for XOR gates. Observe that the ciphertext in a XOR gate with input wires i, j and output wire ℓ equals $C[g] = F_{k_j^{\overline{\pi}_j}}(g\|1)[1..n] \oplus \tilde{k}_j^{\overline{\pi}_j}$. However,

$\tilde{k}_j^{\overline{\pi}_j} = \tilde{k}_j^{\pi_j} \oplus \Delta_\ell = F_{k_j^{\pi_j}}(g\|0)[1..n] \oplus \Delta_\ell$ and $\Delta_\ell = \tilde{k}_i^{\pi_i} \oplus \tilde{k}_i^{\overline{\pi}_i} = F_{k_i^{\pi_i}}(g\|0)[1..n] \oplus F_{k_i^{\overline{\pi}_i}}(g\|1)[1..n]$. Thus,

$$C[g] = F_{k_i^{\pi_i}}(g\|0)[1..n] \oplus F_{k_i^{\overline{\pi}_i}}(g\|1)[1..n] \oplus F_{k_j^{\pi_j}}(g\|0)[1..n] \oplus F_{k_j^{\overline{\pi}_j}}(g\|1)[1..n] \quad (1)$$

where π_i, π_j are the permutation bits that are associated with the bit 0 on wires i, j respectively. Now, assume that the evaluator holds the keys $k_i^{v_i}$ and $k_j^{v_j}$ that are associated with the (plain) bits v_i, v_j. Then, according to procedure Eval, it computes: $F_{k_i^{v_i}}(g\|\lambda_i)[1..n] \oplus F_{k_j^{v_j}}(g\|\lambda_j)[1..n] \oplus \lambda_j C[g]$. Thus, if $\lambda_j = 0$ then it computes

$$F_{k_i^{v_i}}(g\|\lambda_i)[1..n] \oplus F_{k_j^{v_j}}(g\|0)[1..n] \quad (2)$$

and if $\lambda_j = 1$ then it computes

$$F_{k_i^{v_i}}(g\|\lambda_i)[1..n] \oplus F_{k_j^{v_j}}(g\|1)[1..n] \oplus F_{k_i^{\pi_i}}(g\|0)[1..n] \oplus F_{k_i^{\overline{\pi}_i}}(g\|1)[1..n] \oplus F_{k_j^{\pi_j}}(g\|0)[1..n] \oplus F_{k_j^{\overline{\pi}_j}}(g\|1)[1..n]. \quad (3)$$

Recall that$\lambda_j = v_j \oplus \pi_j$. Thus, if $\lambda_j = 1$ then $v_j = \pi_j \oplus 1 = \bar{\pi}_j$, and if $\lambda_j = 0$ then $v_j = \pi_j$. Likewise, for λ_i, v_i and π_i.

We first consider the case that $\lambda_j = 0$. Note that in wire i, we have that $\tilde{k}_i^{v_i} = F_{k_i^{v_i}}(g\|\pi_i \oplus v_i)[1..n]$ (see Step 2 in Procedure GbXOR). Thus, by the above relation between λ_i, v_i and π_i, it follows that $\tilde{k}_i^{v_i} = F_{k_i^{v_i}}(g\|\lambda)[1..n]$. Furthermore, by Step 4 in Procedure GbXOR, we have that $\tilde{k}_j^{\pi_j} = F_{k_j^{\pi_j}}(g\|0)$. In this case of $\lambda_j = 0$ we have that $v_j = \pi_j$ and thus $\tilde{k}_j^{v_j} = F_{k_j^{v_j}}(g\|0)$. Combining this with Eq. (2), we conclude that when $\lambda_j = 0$, the evaluator computes

$$F_{k_i^{v_i}}(g\|\lambda_i)[1..n] \oplus F_{k_j^{v_j}}(g\|0)[1..n] = \tilde{k}_i^{v_i} \oplus \tilde{k}_j^{v_j}.$$

Now consider $\lambda_j = 1$. Observe that $F_{k_i^{v_i}}(g\|\lambda_i)[1..n] \in \left\{F_{k_i^{\pi_i}}(g\|0)[1..n], F_{k_i^{\overline{\pi}_i}}(g\|1)[1..n]\right\}$ and that if $\lambda_i = 0$ then $v_i = \pi_i$ and otherwise $v_i = \bar{\pi}_i$. Thus, $F_{k_i^{v_i}}(g\|\lambda_i)[1..n]$ cancels out and

$$F_{k_i^{v_i}}(g\|\lambda_i)[1..n] \oplus F_{k_i^{\pi_i}}(g\|0)[1..n] \oplus F_{k_i^{\overline{\pi}_i}}(g\|1)[1..n] = F_{k_i^{\bar{v}_i}}(g\|\bar{\lambda}_i)[1..n].$$

If $v_i = 0$ then $\lambda_i = \pi_i$ and we have $F_{k_i^{\bar{v}_i}}(g\|\bar{\lambda}_i)[1..n] = F_{k_i^1}(g\|\bar{\pi}_i)[1..n]$, which is exactly $\tilde{k}_i^1$ according to Step 2 of Procedure GbXOR. If $v_i = 1$ then $\lambda_i = \overline{\pi}_i$ and we have $F_{k_i^{\bar{v}_i}}(g\|\bar{\lambda}_i)[1..n] = F_{k_i^0}(g\|\pi_i)[1..n]$, which is exactly $\tilde{k}_i^0$. In both cases, we receive $\tilde{k}_i^{\overline{v}_i}$. Likewise $F_{k_j^{v_j}}(g\|1)[1..n] = F_{k_j^{\overline{\pi}_j}}(g\|1)[1..n]$ because $\lambda_j = 1$ and so $v_j = \overline{\pi}_j$. Thus, this element cancels out and

$$F_{k_j^{v_j}}(g\|1)[1..n] \oplus F_{k_j^{\pi_j}}(g\|0)[1..n] \oplus F_{k_j^{\overline{\pi}_j}}(g\|1)[1..n] = F_{k_j^{\pi_j}}(g\|0)[1..n] = \tilde{k}_j^{\pi_j} = \tilde{k}_j^{\overline{v}_j}.$$

where the second last equality is from Step 4 in Procedure GbXOR. We conclude that when $\lambda_j = 1$ the evaluator receives $\tilde{k}_i^{\overline{v}_i} \oplus \tilde{k}_j^{\overline{v}_j}$.

Since $\tilde{k}_i^0 \oplus \tilde{k}_i^1 = \tilde{k}_j^0 \oplus \tilde{k}_j^1$, we conclude that the output equals $\tilde{k}_i^{v_i} \oplus \tilde{k}_j^{v_j}$ for both values of λ_j. The fact that this yields the correct output is immediate from the way the output wire values are chosen for the gate.

Intuition For security. As just explained, the ciphertext in a XOR gate is the result of XORing the four outputs of the pseudorandom function:

$$C[g] = F_{k_i^{\pi_i}}(g\|0)[1..n] \oplus F_{k_i^{\overline{\pi}_i}}(g\|1)[1..n] \oplus F_{k_j^{\pi_j}}(g\|0)[1..n] \oplus F_{k_j^{\overline{\pi}_j}}(g\|1)[1..n]$$

Each one of these four computations uses a different key, from which only two keys are known to the evaluator. Since we use the gate index as an input to the function, we are guaranteed that when a wire enters multiple gates, the pseudorandom values we compute will be different in each of the gates. Thus, the ciphertext looks like a random string to the evaluator. In addition, the output-wire key values are determined by the result of the pseudorandom function computation as well. Thus, they are new keys that do not appear elsewhere in the circuit. We stress that the four values in the equation above are not the four new translated keys. If that was the case, then XORing them would yield 0, because the same offset is used in both wires after the translation. Instead, the first three values are the translated keys, but the last value is just a pseudorandom string that is used to mask them in a "one-time pad"-like encryption.

A similar argument applies for AND gates. Since the evaluator can compute only two of the eight PRF computations using the two keys it holds, and since the values that are used in computing the garbled table are unique and do not appear elsewhere in the circuit (again, this is ensured by using the gate index and the permutation bits as input to each pseudorandom function computation), the gate ciphertexts that are not associated with the keys known to the evaluator, look random to the evaluator.

3.5 Proof of Security

3.5.1 Preliminaries

We begin by defining an experiment based on pseudorandom functions that will be convenient for proving security of the garbling scheme. As we have mentioned, we consider a family of functions $\mathcal{F} = \{F_n\}_{n\in\mathbb{N}}$ where for every n it holds that $F_n : \{0,1\}^n \times \{0,1\}^{n+1} \to \{0,1\}^{n+1}$. For clarity, we drop the subscript and write $F_k(x)$ where $k \in \{0,1\}^n$ instead of $F_n(k,x)$.

We now define the experiment, call $2PRF$. In this experiment, the distinguisher/adversary is given access to four oracles, divided into two pairs. The second and fourth oracles are always pseudorandom functions F_{k_1} and F_{k_2}, respectively. In contrast, the first and third oracles are either the *same* pseudorandom functions F_{k_1} and F_{k_2}, respectively, or independent truly random functions f^1 and f^3. Clearly, if $\mathcal{A}$ can make the same query to the first and second oracle, or to the third and fourth oracle, then it can easily distinguish the cases. The security requirement is that as long as it does *not* make such queries, it *cannot* distinguish the cases. We prove that this property holds for any pseudorandom function. The experiment is formally defined in Figure 7, and 2PRF security is formalized in Definition 3.2.

Definition 3.2 *Let $\mathcal{F} = \{F_n\}_{n\in\mathbb{N}}$ be an efficient family of functions where for every n, $F_n : \{0,1\}^n \times \{0,1\}^{n+1} \to \{0,1\}^{n+1}$. Family $\mathcal{F}$ is a* 2PRF *if for every probabilistic-polynomial time adversary $\mathcal{A}$ there exists a negligible function μ such that for every n,*

$$\left|\Pr[\mathsf{Expt}^{2PRF}_{\mathcal{F},\mathcal{A}}(n,1)=1] - \Pr[\mathsf{Expt}^{2PRF}_{\mathcal{A}}(n,0)]=1\right| < \mu(n)$$

The following lemma shows that pseudorandomness of F_k is sufficient for it to be 2PRF as well.

Experiment $\mathsf{Expt}^{2PRF}_{\mathcal{F},\mathcal{A}}(n,\sigma)$:

1. Choose random keys $k_1, k_2 \leftarrow \{0,1\}^n$ for the pseudorandom function, and choose two truly random functions f^1, f^2. If $\sigma = 0$, set $(\mathcal{O}^{(1)}, \mathcal{O}^{(2)}, \mathcal{O}^{(3)}, \mathcal{O}^{(4)}) = (F_{k_1}, F_{k_1}, F_{k_2}, F_{k_2})$; else, set $(\mathcal{O}^{(1)}, \mathcal{O}^{(2)}, \mathcal{O}^{(3)}, \mathcal{O}^{(4)}) = (f^1, F_{k_1}, f^2, F_{k_2})$
2. The adversary $\mathcal{A}$ is invoked upon input 1^n
3. When $\mathcal{A}$ makes a query (j,x) to its oracles with $j \in \{1,2,3,4\}$ and $x \in \{0,1\}^{n+1}$, answer as follows:
   - if $j \in \{1,2\}$ and x was already queried to $\{1,2\} \setminus j$, return $\perp$
   - if $j \in \{3,4\}$ and x was already queried to $\{3,4\} \setminus j$, return $\perp$
   - Otherwise, return $\mathcal{O}^{(j)}(x)$
4. $\mathcal{A}$ outputs a bit σ', and this is the output of the experiment

Figure 7: The 2PRF experiment

Lemma 3.3 *If $\mathcal{F}$ is a family of pseudorandom functions, then it is a 2PRF.*

Proof: Assume that F is a PRF. Denote by $\mathsf{Expt}^{g_1,g_2,g_3,g_4}_{\mathcal{A}}(n)$ the experiment where $\mathcal{A}$ is given oracle access to functions g_1, g_2, g_3, g_4 (under the input limitations outlined in the experiment). Using this notation, we have that $\mathsf{Expt}^{2PRF}_{\mathcal{F},\mathcal{A}}(n,0) = \mathsf{Expt}^{F_{k_1},F_{k_1},F_{k_2},F_{k_2}}_{\mathcal{A}}(n)$ and $\mathsf{Expt}^{2PRF}_{\mathcal{F},\mathcal{A}}(n,1) = \mathsf{Expt}^{f^1,F_{k_1},f^2,F_{k_2}}_{\mathcal{A}}(n)$.

First, a straightforward reduction to the security of the pseudorandom function (with a hybrid for two pseudorandom functions) yields that for every probabilistic-polynomial time adversary $\mathcal{A}$ there exists a negligible function μ such that for every n,

$$\left|\Pr\left[\mathsf{Expt}^{F_{k_1},F_{k_1},F_{k_2},F_{k_2}}_{\mathcal{A}}(n)=1\right] - \Pr\left[\mathsf{Expt}^{f^1,f^1,f^2,f^2}_{\mathcal{A}}(n)=1\right]\right| \leq \mu(n).$$

Note that oracle access to the same random function twice or to two different random functions is *identical* when there is a constraint that the same input cannot be supplied to both oracles. Thus, for every adversary $\mathcal{A}$ and for every n,

$$\Pr\left[\mathsf{Expt}^{f^1,f^1,f^2,f^2}_{\mathcal{A}}(n)=1\right] = \Pr\left[\mathsf{Expt}^{f^1,f^3,f^2,f^4}_{\mathcal{A}}(n)=1\right].$$

Next, we claim that for every probabilistic-polynomial time adversary $\mathcal{A}$ there exists a negligible function μ such that for every n,

$$\left|\Pr\left[\mathsf{Expt}^{f^1,f^3,f^2,f^4}_{\mathcal{A}}(n)=1\right] - \Pr\left[\mathsf{Expt}^{f^1,F_{k_1},f^2,f^4}_{\mathcal{A}}(n)=1\right]\right| \leq \mu(n).$$

This follows from a direct reduction to the pseudorandomness of F (the reduction simulates f^1, f^3, f^4 itself and uses its oracle to either have f^2 or F_{k_1}). Likewise, a direction reduction yields that for every probabilistic-polynomial time adversary $\mathcal{A}$ there exists a negligible function μ such that for every n,

$$\left|\Pr\left[\mathsf{Expt}^{f^1,F_{k_1},f^2,f^4}_{\mathcal{A}}(n)=1\right] - \Pr\left[\mathsf{Expt}^{f^1,F_{k_1},f^2,F_{k_2}}_{\mathcal{A}}(n)=1\right]\right| \leq \mu(n).$$

Here the reduction simulates f^1, F_{k_1}, f^2 itself. Combining all of the above, we conclude that for every adversary $\mathcal{A}$ there exists a negligible function μ such that for every n,

$$\begin{aligned}&\left|\Pr\left[\mathsf{Expt}^{2PRF}_{\mathcal{F},\mathcal{A}}(n,0)=1\right]-\Pr\left[\mathsf{Expt}^{2PRF}_{\mathcal{F},\mathcal{A}}(n,1)=1\right]\right| \\ &\quad = \left|\Pr\left[\mathsf{Expt}^{F_{k_1},F_{k_1},F_{k_2},F_{k_2}}_{\mathcal{A}}(n)=1\right]-\Pr\left[\mathsf{Expt}^{f^1,F_{k_1},f^2,F_{k_2}}_{\mathcal{A}}(n)=1\right]\right| \leq \mu(n).\end{aligned}$$

■

3.5.2 The Proof of Security of Our Garbling Scheme

We begin by proving that our garbling scheme achieves privacy. Let G denote our garbling scheme.

Theorem 3.4 *If $\mathcal{F}$ is a family of pseudorandom functions, then the garbling scheme G achieves privacy.*

Proof: We begin by describing a simulator $\mathcal{S}$ for the $\mathsf{Expt}^{\text{priv}}$ privacy experiment. S is invoked with input $(1^n, c, c(x))$ and works as follows. As we will show, $\mathcal{S}$ will define an active key on every wire. This key will be the one that is "obtained" in the evaluation procedure. The other key is not active, and is actually never explicitly defined. Rather, all the ciphertexts in the gates that are not "decrypted" in the evaluation are chosen at random.

1. For each input wire j in circuit c:
   (a) Choose an active key: $k_j \leftarrow \{0,1\}^n$
   (b) Choose an active signal bit $\lambda_j \leftarrow \{0,1\}$
   (c) Prepare the garbled input data: $X[j] = k_j \| \lambda_j$
2. In topological order, for each gate g in c:
   (a) If g is a XOR gate with input wires i, j and output wire ℓ:
      i. Compute the active output-wire signal bit: $\lambda_\ell := \lambda_i \oplus \lambda_j$
      ii. Compute a translated new key for wire i: $\tilde{k}_i := F_{k_i}(g\|\lambda_i)[1..n]$
      iii. Compute a translated new key for wire j, and the ciphertext for this gate:
         A. If $\lambda_j = 0$, set $\tilde{k}_j := F_{k_j}(g\|0)[1..n]$ and $C[g] \leftarrow \{0,1\}^n$ (in this case, the translated key is obtained by computing F and so is correctly computed, but the ciphertext is not used and so is random)
         B. If $\lambda_j = 1$, set $\tilde{k}_j \leftarrow \{0,1\}^n$ and $C[g] := F_{k_j}(g\|1)[1..n] \oplus \tilde{k}_j$ (in this case, the translated key is obtained via the ciphertext, and so the ciphertext is correctly computed, but using a random key)
      iv. Compute the output wire active key: $k_\ell := \tilde{k}_i \oplus \tilde{k}_j$
   (b) If g is an AND gate with input wires i, j and output wire ℓ:
      i. Set the output-wire active key and signal bit:
         A. If $\lambda_i = \lambda_j = 0$, set $k_\ell \| \lambda_\ell := F_{k_i}(g\|00) \oplus F_{k_j}(g\|00)$ (in this case, the output key is computed via F and so must be set in this way)

B. Else, set $k_\ell \| \lambda_\ell \leftarrow \{0,1\}^{n+1}$ (in this case, the output key is computed from the ciphertexts and is chosen at random)

ii. Compute the gate's ciphertexts (they are random except for the one that is opened according to the active signal bits):

A. If $2\lambda_i + \lambda_j \neq 0$, then $T_{2\lambda_i+\lambda_j} := F_{k_i}(g\|\lambda_i\lambda_j) \oplus F_{k_j}(g\|\lambda_i\lambda_j) \oplus k_\ell\|\lambda_\ell$

B. For $\alpha \in \{1,2,3\} \setminus \{2\lambda_i + \lambda_j\} : T_\alpha \leftarrow \{0,1\}^{n+1}$

C. $C[g] \leftarrow T_1, T_2, T_3$

iii. If g is a NOT gate with input wire i and output wire ℓ: set $k_\ell\|\lambda_\ell = k_i\|\lambda_i$

3. For each output wire j in c:

(a) Prepare the decoding information: $d[j, c(x)_j] := F_{k_j}(\text{out}\|\lambda_j)$ and $d[j, \overline{c(x)}_j] \leftarrow \{0,1\}^n$

4. Return (C, X, d)

Note that the garbled tables in the simulator-generated garbled circuit consists of random strings, except for the ciphertexts used in the evaluation itself. Specifically, in an AND gate all ciphertexts are random in the case that $\lambda_i = \lambda_j = 0$ since none are used in evaluation; in all other cases, the single ciphertext which is decrypted is constructed "correctly" whereas all other are random. Likewise, in a XOR gate where $\lambda_j = 0$ the ciphertext is random since in this case the ciphertext is not used in evaluation.

We now show that the simulated garbled circuit is indistinguishable from a real garbled circuit by reduction to the 2PRF experiment, which by Lemma 3.3 follows merely from the fact that F is a pseudorandom function. Let $\mathcal{A}$ be a probabilistic-polynomial time adversary for $\mathsf{Expt}^{\text{priv}}$, and let m denote the number of gates in the circuit. We define a hybrid distribution $H_i(c, x)$ with $0 \leq i \leq m$ as the triple (C, X, d) generated in the following way (note that the procedure for generating $H_i(c, x)$ is given the circuit c and the real input x):

- *Garbling of gates:* The garbled circuit C is generated by garbling the first i gates in the topological order using the simulator garbling procedure, while gates $i+1, \ldots, m$ are garbled using the real garbling scheme. Observe that the simulator generates only a single key per wire; specifically, it generates the active key $k_j^{\lambda_j}$. The first step in the hybrid is therefore to choose an additional key $k_j^{1-\lambda_j}$ for every wire that enters or exits a gate that is garbled according to the real scheme.

- *Encoding information X:* For each circuit *input* wire j that enters a gate g, if g is garbled using the real scheme (i.e., the gate's index $> i$), then $X[j]$ is the garbled value that was chosen to mask the jth bit of the input (recall that in experiment $\mathsf{Expt}^{\text{priv}}$, the adversary knows the input string x and so can choose the correct encoding for x). Else, if g is garbled using the simulator procedure (i.e., the gate's index $\leq i$), then $X[j]$ is the garbled value that was chosen for the active key of that wire.

- *Decoding information d:* For each output wire j that exits from a gate g, if g was garbled using the real scheme then there are two garbled values on wire j, and $d[j, \cdot]$ is generated exactly as in the Garble procedure. Else, if g is garbled using the simulator instructions, then there is only one garbled value on j and $d[j, \cdot]$ is generated exactly as in the simulator procedure.

Note that the hybrid $H_0(x)$ is a real garbled circuit (and is distributed as (C, X, d) in $\mathsf{Expt}^{\text{priv}}_{G,\mathcal{A},\mathcal{S}}$ in the case that $\beta = 0$), while $H_m(x)$ is the output of the simulator S (and is distributed as (C, X, d) in $\mathsf{Expt}^{\text{priv}}_{G,\mathcal{A},\mathcal{S}}$ in the case that $\beta = 1$). Next, for each $0 \leq i \leq m$, we define $\mathcal{A}_i$ to be a probabilistic-polynomial time adversary for $\mathsf{Expt}^{2PRF}_{\mathcal{F},\mathcal{A}_i}(n, \sigma)$ experiment. $\mathcal{A}_i$ is given access to four oracles:

$$(\mathcal{O}^{(1)}(\cdot), \mathcal{O}^{(2)}(\cdot), \mathcal{O}^{(3)}(\cdot), \mathcal{O}^{(4)}(\cdot)) = (f^1(\cdot) \textit{ or } F_{k_1}(\cdot), F_{k_1}(\cdot), f^2(\cdot) \textit{ or } F_{k_2}(\cdot), F_{k_2}(\cdot)).$$

Adversary $\mathcal{A}_i$ runs $\mathsf{Expt}^{\text{priv}}$ with adversary $\mathcal{A}$. First, it invokes $\mathcal{A}$ and receives (c, x). Then, as we will see, it constructs a garbled circuit which will either be distributed according to H_{i-1} or H_i, depending on the oracles it received. Thus, as we will show, if $\mathcal{A}$ can succeed in $\mathsf{Expt}^{\text{priv}}$ with probability that is non-negligibly greater than $1/2$, then $\mathcal{A}_i$ will distinguish in the 2PRF experiment with non-negligible probability.

Formally, adversary $\mathcal{A}_i$ constructs a garbled circuit by generating the first $i-1$ gates in topological order using the simulator procedure, and generating the gates indexed by $i+1, \ldots, m$ using the real Garble instructions (with subroutines GbXOR and GbAND). However, for the ith gate, $\mathcal{A}_i$ will use its oracles to generate a garbled table that is garbled as in the real scheme or as in the simulator code, depending whether it received an oracle access to pseudorandom or to random functions. Assume the input wires of the ith gate are a, b and the output wire is c. In addition, assume that the active keys on the input wires are associated with the bits v_a, v_b (recall that $\mathcal{A}_i$ knows the input to the circuit and thus v_a, v_b are known to it). Knowing $k_a^{v_a}$ and $k_b^{v_b}$, adversary $\mathcal{A}_i$ will (implicitly) use the secrets k_1, k_2 that were chosen for the pseudorandom function in Expt^{2PRF} as $k_a^{\overline{v}_a}$ and $k_b^{\overline{v}_b}$ respectively. Thus, whenever $\mathcal{A}_i$ needs to compute $F_{k_a^{\overline{v}_a}}(x)$ or $F_{k_b^{\overline{v}_b}}(x)$ for some x, it will send x to its oracles $\mathcal{O}^{(1)}$ or $\mathcal{O}^{(3)}$ respectively (recall that these are either also F_{k_1}, F_{k_2} or are random functions f^1, f^2). We remark that $\mathcal{O}^{(2)}$ and $\mathcal{O}^{(4)}$ are used to garble gates $\ell > i$ that use wires a, b as well; this will be described after we present the method for garbling the ith gate. We separately consider the case that the ith gate is a XOR gate and the case that it is AND gate.

Case 1 – the ith gate is a XOR gate: the keys on the input wires to this gate were generated using the simulator procedure. Thus, $\mathcal{A}_i$ holds one key on each input wire a and b, denoted k_a and k_b, respectively. $\mathcal{A}_i$ sets these keys to be $k_a^{v_a}$ and $k_b^{v_b}$, respectively. In addition, $\mathcal{A}_i$ has signal bits λ_a, λ_b that were determined on these wires. $\mathcal{A}$ constructs the gate as follows:

1. $\mathcal{A}_i$ computes the permutation bit for the output-wire c: $\pi_c := \pi_a \oplus \pi_b = (\lambda_a \oplus v_a) \oplus (\lambda_b \oplus v_b)$
2. $\mathcal{A}_i$ computes new translated keys for wire a: $\tilde{k}_a^{v_a} := F_{k_a^{v_a}}(g \| \lambda_a)[1..n]$ and $\tilde{k}_a^{\overline{v}_a} := \mathcal{O}^{(1)}(g \| \overline{\lambda}_a)[1..n]$ (observe that if $\mathcal{O}^{(1)}$ is pseudorandom then this is a "real" key value, whereas if it is a random function then this is an independent random key)
3. $\mathcal{A}_i$ computes the offset of the output wire: $\Delta_c := \tilde{k}_a^{v_a} \oplus \tilde{k}_a^{\overline{v}_a}$
4. $\mathcal{A}_i$ computes new translated keys for wire b and the ciphertext:
   (a) If the signal bit of $k_b^{v_b}$ is 0 (i.e., if $\lambda_b = 0$), set $\tilde{k}_b^{v_b} := F_{k_b^{v_b}}(g \| 0)[1..n]$ and $\tilde{k}_b^{\overline{v}_b} := \tilde{k}_b^{v_b} \oplus \Delta_c$, and define $C[g] := \mathcal{O}^{(3)}(g \| 1)[1..n] \oplus \tilde{k}_b^{\overline{v}_b}$
   (b) Else, if the signal bit of $k_b^{v_b}$ is 1 (i.e., if $\lambda_b = 1$), set $\tilde{k}_b^{\overline{v}_b} := \mathcal{O}^{(3)}(g \| 0)[1..n]$ and $\tilde{k}_b^{v_b} := \tilde{k}_b^{\overline{v}_b} \oplus \Delta_c$, and define $C[g] := F_{k_b^{v_b}}(g \| 1)[1..n] \oplus \tilde{k}_b^{v_b}$
5. $\mathcal{A}_i$ computes the output wire keys: $k_c^0 := \tilde{k}_a^0 \oplus \tilde{k}_b^0$ and $k_c^1 := k_c^0 \oplus \Delta_c$

It easy to see that when $\sigma = 0$ (and the oracle answers are pseudorandom strings), the code is identical to the real garbling scheme. In contrast, when $\sigma = 1$ (and the oracle the answers are random strings), then the result is exactly according to the simulator instructions. In order to see this, observe that if $\lambda_b = 0$ then $C[g]$ is random, exactly as in Step 2(a)iiiA of the simulator. This is because $\mathcal{O}^{(3)}$ is random and so the XOR with $\tilde{k}_b^{\overline{v}_b}$ makes no difference. Likewise, if $\lambda_b = 1$ then the active key $\tilde{k}_b^{v_b}$ is random since it is the XOR of the output of $\mathcal{O}^{(3)}$ with another value, and $C[g]$ is the XOR of this key with the appropriate output from $F_{k_b^{v_b}}$. Thus, this is also exactly as in Step 2(a)iiiB of the simulator.

Case 2 – the ith gate is an AND gate: As before, for wires a and b, $\mathcal{A}_i$ has two keys $k_a^{v_a}, k_b^{v_b}$, two signal bits λ_a, λ_b and the bits v_a, v_b that are on the wires. Then it does the following:

1. Compute the values $K_0, \ldots, K_3$:

$$\begin{aligned}
K_{2\lambda_a+\lambda_b} &:= F_{k_a^{v_a}}(g\|\lambda_a\lambda_b) \oplus F_{k_b^{v_b}}(g\|\lambda_a\lambda_b) \\
K_{2\lambda_a+\overline{\lambda}_b} &:= F_{k_a^{v_a}}(g\|\lambda_a\overline{\lambda}_b) \oplus \mathcal{O}^{(3)}(g\|\lambda_a\overline{\lambda}_b) \\
K_{2\overline{\lambda}_a+\lambda_b} &:= \mathcal{O}^{(1)}(g\|\overline{\lambda}_a\lambda_b) \oplus F_{k_b^{v_b}}(g\|\overline{\lambda}_a\lambda_b) \\
K_{2\overline{\lambda}_a+\overline{\lambda}_b} &:= \mathcal{O}^{(1)}(g\|\overline{\lambda}_a\overline{\lambda}_b) \oplus \mathcal{O}^{(3)}(g\|\overline{\lambda}_a\overline{\lambda}_b)
\end{aligned}$$

2. Set the output wire keys and permutation bits:
   (a) Compute: $\pi_a = v_a \oplus \lambda_a$ and $\pi_b = v_b \oplus \lambda_b$
   (b) If $\pi_a = \pi_b = 1$, set $k_c^0\|\pi_c \leftarrow \{0,1\}^{n+1}$ and $k_c^1 := K_0[1..n]$
   (c) Else, set $k_c^0\|\pi_c := K_0$ and $k_c^1 \leftarrow \{0,1\}^n$

   Denote $K_c^0 := k_c^0\|\pi_c$ and $K_c^1 := k_c^1\|\overline{\pi}_c$

3. Compute the ciphertexts: For $\alpha \in \{1,2,3\}$,
   (a) If $\alpha = 2\overline{\pi}_a + \overline{\pi}_b$, then $T_\alpha := K_\alpha \oplus K_c^1$
   (b) Else: $T_\alpha := K_\alpha \oplus K_c^0$

   Set $C[g] \leftarrow \{T_1, T_2, T_3\}$

As in the previous case, when $\sigma = 0$, the code is identical to real garbling scheme. When $\sigma = 1$, the answers of the oracles are random strings, and therefore all the rows in the garbled table are random as well, except for the row that is pointed to by the signal bits of the active keys (the row $T_{2\lambda_a+\lambda_b}$ where the adversary computes the value of $K_{2\lambda_a+\lambda_b}$ directly using the keys it holds). Thus, the gate is garbled as in the simulation.

We conclude that when $\sigma = 0$, the ith gate is garbled as in the real garbling scheme, while when $\sigma = 1$ the ith gate is garbled as in the simulator procedure. However, to complete the construction of the garbled circuit, $\mathcal{A}_i$ needs to construct all the gates $\ell > i$. For a gate $\ell > i$ with input-wires that are output from the ith gate and greater, $\mathcal{A}_i$ has both keys on the wires and so can compute the gate just like in the real garbling procedure. If a gate $\ell > i$ has an input-wire that is output from a gate $j < i$ that does not equal a or b, then $\mathcal{A}_i$ simply chooses the (inactive) key at random, like in the hybrid definition. Finally, for a gate that has an input wire a or b, the gate is constructed used

oracles $\mathcal{O}^{(2)}$ and $\mathcal{O}^{(4)}$ and the same code for gate i (except with these oracles instead of $\mathcal{O}^{(1)}$ and $\mathcal{O}^{(3)}$). Since these oracles always use the pseudorandom functions, it follows that the computation of the gate is always according to the real garbling method. (Note that when garbling the ℓth gate, each of the queries to these oracles includes the gate's number. Thus we are guaranteed that these queries were not sent to $\mathcal{O}^{(1)}$ and $\mathcal{O}^{(3)}$ when $\mathcal{A}_i$ garbled the ith gate, as required in $\mathsf{Expt}^{2PRF}_{\mathcal{F},\mathcal{A}}(n,\sigma)$ experiment.)

Concluding the proof, when $\sigma = 0$, $\mathcal{A}_i$ constructs the hybrid $H_{i-1}(x)$, while when $\sigma = 1$, $\mathcal{A}_i$ constructs the hybrid $H_i(x)$. We therefore construct a single adversary $\mathcal{A}'$ for $2PRF$ who chooses a random i and then runs $\mathcal{A}_i$ with adversary $\mathcal{A}$. By a standard hybrid argument, if $\mathcal{A}$ succeeds with non-negligible probability in $\mathsf{Expt}^{\text{priv}}$ then $\mathcal{A}'$ distinguishes between $\mathsf{Expt}^{2PRF}_{\mathcal{F},\mathcal{A}}(n,0)$ and $\mathsf{Expt}^{2PRF}_{\mathcal{F},\mathcal{A}}(n,1)$, with non-negligible probability. This contradicts the assumption that $\mathcal{F}$ is a family of pseudorandom functions. ■

Achieving obliviousness and authenticity. In order to satisfy the obliviousness requirement, we need to construct a simulator that outputs (C, X) given only c as an input. Note that the simulator $\mathcal{S}$ constructed above for the privacy requirement outputs the triple (C, X, d). However, $\mathcal{S}$ uses c only for generating (C, X), and in particular the output $c(x)$ is used only for generating d. Thus, we can simply remove the generation of the decoding information from $\mathcal{S}$'s instruction, and we obtain a simulator that generates only (C, X) as required. Proving that this simulator's output is indistinguishable from (C, X) generated by the real scheme, is the same as in the proof of privacy.[8]

Regarding authenticity, we need to show that a probabilistic-polynomial time adversary $\mathcal{A}$ that is given (C, X) as input can output $\tilde{Y}$ such that $\mathsf{Decode}(\tilde{Y}, d) \notin \{c(x), \perp\}$ with at most negligible probability. Note that if we give $\mathcal{A}$ the pair (C, X) generated by our simulator, it can succeed only with probability at most 2^{-n}. This is due to the fact that in the simulated garbled circuit, for each output wire j corresponding to the jth output bit, $d[j, \overline{c(x)}_j]$ is a random string. Now, if given the real (C, X), the adversary can output such a $\tilde{Y}$ with non-negligible probability, then it could be used by an adversary given (C, X, d), to break the privacy property, in contradiction to Theorem 3.4. Observe that since the adversary in the privacy experiment is given all of the decoding information d, it can efficiently verify if $\mathcal{A}$ output a $\tilde{Y}$ with the property that $\mathsf{Decode}(\tilde{Y}, d) \notin \{c(x), \perp\}$.

3.6 XOR Gates with Only Three PRF Computations

Our garbling method requires four calls to the pseudorandom function for garbling XOR gates, where each call uses a different key. In this section we show that it is possible to remove one of these calls by leaving one of the input keys unchanged. Recall that our ciphertext for a XOR gate g with input wires i, j and output wire ℓ is:

$$C[g] = F_{k_i^{\pi_i}}(g\|0)[1..n] \oplus F_{k_i^{\overline{\pi}_i}}(g\|1)[1..n] \oplus F_{k_j^{\pi_j}}(g\|0)[1..n] \oplus F_{k_j^{\overline{\pi}_j}}(g\|1)[1..n] \tag{4}$$

Assume the evaluator has the keys $k_i^{v_i}, k_j^{v_j}$ and the signal bits λ_i, λ_j when computing the gate. Then, using the ciphertext $C[g]$ it computes $C[g] \oplus F_{k_i^{v_i}}(g\|\lambda_i)[1..n] \oplus F_{k_j^{v_j}}(g\|\lambda_j)[1..n]$ and obtains $F_{k_i^{\overline{v}_i}}(g\|\overline{\lambda}_i)[1..n] \oplus F_{k_j^{\overline{v}_j}}(g\|\overline{\lambda}_j)$, which is the XOR of two pseudorandom values. If we leave, for

[8] Recall that in the reduction the adversary $\mathcal{A}_i$ knows the input x. However, in the experiment in the proof of obliviousness, $\mathcal{A}$ also outputs x and so it is known. Thus, the same reduction works.

example, the value of $k_j^{\overline{v}_j}$ unchanged – i.e., use it in Eq. (4) instead of $F_{k_j^{\overline{v}_j}}(g\|\overline{\lambda}_j)$ – the evaluator will be able to compute $F_{k_i^{\overline{v}_i}}(g\|\overline{\lambda}_i)[1..n] \oplus k_j^{\overline{v}_j}$. Observe that the evaluator still cannot learn anything since one of the two values is a new pseudorandom value that does not appear anywhere else in the circuit (taking the gate index g as an input to F ensures that if a wire i or j enters multiple gates, then we compute a different value for each gate). Therefore, the ciphertext is pseudorandom as required. In addition, since the two keys on wire i are still translated to new keys, the output wire keys, generated in the same way as before, are guaranteed to obtain new fresh values. (See Footnote 7 as to why we cannot use the same method to remove one of the pseudorandom function calls on wire i as well.)

The modified garbling scheme. Denote the modified scheme where only three pseudorandom function calls are made by G'. Figure 8 presents the modifications in G' compared to our base scheme; only the items in GbXOR and Eval that were changed appear, and the actual changes appear in bold. The procedure GbXOR is changed by not changing the key on wire j that represents the bit v_j when $v_j \oplus \pi_j = 0$; i.e., $\tilde{k}_j^0$ is set to k_j^0 instead of $F_{k_j^0}(g\|0)[1..n]$. Consequently, in the Eval procedure, the evaluator uses the signal bit it holds to decide whether to translate the key on wire j into a new key or not.

Procedure $\mathsf{GbXOR}(k_i^0, k_i^1, k_j^0, k_j^1, \pi_i, \pi_j)$**:**

4. Compute translated new keys for wire j and the garbled value for this gate:
   (a) If $\pi_j = 0$, set $\boldsymbol{\tilde{k}_j^0 := k_j^0}$, $\tilde{k}_j^1 := \tilde{k}_j^0 \oplus \Delta_\ell$, and $T := F_{k_j^1}(g\|1)[1..n] \oplus \tilde{k}_j^1$
   (b) If $\pi_j = 1$, set $\boldsymbol{\tilde{k}_j^1 := k_j^1}$, $\tilde{k}_j^0 := \tilde{k}_j^1 \oplus \Delta_\ell$, and $T := F_{k_j^0}(g\|1)[1..n] \oplus \tilde{k}_j^0$

Procedure $\mathsf{Eval}(C, X)$**:**

2. For each gate g in c, in topological order:
   (a) If g is a XOR gate with input wires i, j and output wire ℓ:
      i. Compute the output wire key:
         A. **If $\boldsymbol{\lambda_j = 0}$, set $\boldsymbol{k_\ell := F_{k_i}(g\|\lambda_i)[1..n] \oplus k_j \oplus \lambda_j C[g]}$**
         B. **Else,** set $k_\ell := F_{k_i}(g\|\lambda_i)[1..n] \oplus F_{k_j}(g\|1)[1..n] \oplus \lambda_j C[g]$
      ii. Compute the output wire signal bit: $\lambda_\ell := \lambda_i \oplus \lambda_j$

Figure 8: The Improved Garbling Scheme G'

Security proof. We now prove security of the modified scheme.

Theorem 3.5 *If $\mathcal{F}$ is a family of pseudorandom functions, then the garbling scheme G' achieves privacy.*

Proof Sketch: The proof is very similar to the proof of Theorem 3.4 for G. We describe the main changes that are needed in order to make the proof valid for our modified scheme G'. First, let $\mathcal{S}'$ be a simulator that is identical to the simulator $\mathcal{S}$ from the proof of Theorem 3.4, except that when simulating XOR gates in step 2.(a).iii, when $\lambda_j = 0$, it sets $\tilde{k}_j = k_j$ (instead of as $F_{k_j}(g\|0)[1..n]$). In order to prove that the output of $\mathcal{S}'$ is indistinguishable (C, X, d) generated by the garbling scheme G', we reduce the security to Expt^{2PRF}, as in Theorem 3.4. Specifically, the same hybrid

distribution $H_i(x)$ that was defined in the proof of theorem 3.4 is used here. Then, we define a probabilistic-polynomial time adversary $\mathcal{A}_i$ for Expt^{2PRF}.

$\mathcal{A}_i$ garbles the first $i-1$ gates using the instructions of simulator $\mathcal{S}'$. When $\mathcal{A}_i$ needs to construct the ith gate with input wires a, b and output wire c, $\mathcal{A}_i$ holds two active keys $k_a^{v_a}, k_b^{v_b}$, two signal bits λ_a, λ_b and the actual bits that are on the wire v_a, v_b. If g is an AND gate, then $\mathcal{A}_i$ proceeds exactly as in the proof of Theorem 3.4. If g is a XOR gate, then $\mathcal{A}_i$ proceeds differently, depending on the value of λ_b. If $\lambda_b = 0$ (i.e, the key that $\mathcal{A}_i$ holds on wire b has the signal bit '0'), then in order to generate the ciphertext $C[g]$, the adversary $\mathcal{A}_i$ needs to use its oracle (see Step 4a of $\mathcal{A}_i$ in the case that the ith gate is a XOR gate). In this case, it sets $\tilde{k}_b^{v_b} := k_b^{v_b}$ and sets the other key $\tilde{k}_b^{\overline{v}_b}$ and the ciphertext $C[g]$ exactly as in the proof of Theorem 3.4. However, if $\lambda_j = 1$, then $\mathcal{A}_i$ does not use the oracle anymore in order to generate $\tilde{k}_b^{\overline{v}_b}$ (see Step 4b of $\mathcal{A}_i$ in the case that the ith gate is a XOR gate) since this value is not translated. Thus, $\mathcal{A}_i$ just chooses $k_b^{\overline{v}_b}$ randomly and uses this instead of the call to $\mathcal{O}^{(3)}$. This is the only difference to $\mathcal{A}_i$ (note that when constructing gates for $\ell > i$ where b is an input wire, $\mathcal{A}_i$ does not use $\mathcal{O}^{(3)}$ or $\mathcal{O}^{(4)}$ but rather uses $\tilde{k}_b^{\overline{v}_b}$ as chosen above).

The remainder of the proof is the same. ■

4 Simple and Fast 4-2 GRR for non-XOR gates

4.1 Overview

Abstractly, gate garbling typically works by generating four pseudorandom masks K_0, K_1, K_2, K_3, corresponding to the four possible input combinations (in some permuted order). The evaluator of the circuit is able to compute one of these four masks, and can also use the signal bits to identify the index of that mask. Namely, it computes a pair (i, K_i) (but is unable to identify the real input combination corresponding to the value that it computed).

In our base scheme described in the previous section, we garbled non-XOR gates with three ciphertexts for each garbled gate. One of the ciphertexts was "removed" by setting one of the keys on the output wire to actually be K_0 rather than using K_0 to mask the key (this is called *garbled row reduction*, or GRR for short). In this section we improve on this by applying a 4-2 row reduction technique on these gates in order to remove an additional ciphertext. There are two known such techniques: The 4-to-2 reduction technique method of [20] and the new "Half-Gates" approach of [22]. The "Half-Gates" technique was designed to be compatible with the free-XOR technique and actually *requires* free-XOR; as such, it is based on the circularity assumption and so is not suitable for this paper. In contrast, the 4-2 GRR technique of [20] does not require free-XOR; it has been proved relying on a standard assumption only and can be incorporated into our scheme. However, in this technique, the generation of the garbled table by the circuit garbler, as well as the computation of the output wire key given two ciphertexts of the gate table and the K value, are carried out by interpolating a degree 2 polynomial. We describe here a different 4-to-2 garbling method where the garbling and evaluation of the gate use only simple XOR operations. This is preferable for two major reasons:

- *Efficiency:* Polynomial interpolation uses three finite field multiplications and two additions (after the Lagrange coefficients are precomputed). The overhead of computing the multiplications is rather high, even when implemented in $GF(2^{128})$. For example, our implementation of this task, which used the PCLMULQDQ Intel instruction, needed about half as many cycles as AES encryption.

- *Simpler coding:* Efficient implementation of polynomial interpolation, especially over $GF(2^{128})$, and using machine instructions rather than calling a software library, requires some expertise and is significantly harder to code than a few XOR operations.

Gate evaluation. We first describe the process of evaluating a gate. We will then describe the garbling procedure which enables this gate evaluation procedure. Although this is somewhat reversed (as one would expect a description of how garbling is computed first), we present it this way as we find it clearer.

The gate evaluator receives as input a gate table with two entries $[T_1, T_2]$, an index $i \in \{0, 1, 2, 3\}$, and a value K_i computed from the two garbled values of the input wires (note, T_1, T_2, K_i are all 128 bit strings). It computes the garbled output wire key k_{out} in the following way:

- If $i = 0$ then $k_{out} = K_0$
- If $i = 1$ then $k_{out} = K_1 \oplus T_1$
- If $i = 2$ then $k_{out} = K_2 \oplus T_2$
- If $i = 3$ then $k_{out} = K_3 \oplus T_1 \oplus T_2$

Garbling. We now show how to garble AND gates so that the evaluation described above provides correct evaluation. Due to the random permutation applied to the rows (via the permutation bit), the single output bit "1" of these gates might correspond to any of the masks K_0, K_1, K_2, K_3. Denote the index of that mask as $s \in \{0, 1, 2, 3\}$, and denote by k^0_{out}, k^1_{out} the output wire keys. We need to design a method for computing the garbled output key from the garbled table of this gate and the K_i values, such that

- The method applied to K_s outputs k^1_{out}, and when applied to any other K value it outputs k^0_{out}.
- Given K_s and the gate table, the value k^0_{out} pseudorandom. Similarly, given any other K value and the gate table, k^1_{out} value is pseudorandom.

Our starting point is the basic garbled gate procedure without row reduction, and so with a gate table of four entries $[T_0, T_1, T_2, T_3]$. We denote the garbled value associated with the ith entry of the table with $k_{out,i}$ (it holds that one $k_{out,i}$ value is equal to k^1_{out}, and the other three $k_{out,i}$ values are equal to k^0_{out}). The table contains the four entries $T_i = K_i \oplus k_{out,i}$.

In the 4-to-3 garble the gate entry T_0 is always 0, and therefore **(1)** there is no need to store and communicate that entry, and **(2)** it always holds that $k_{out,0} = K_0$. If $k_{out,0} = k^0_{out}$ then k^1_{out} can be defined arbitrarily, whereas if $k_{out,0} = k^1_{out}$ then k^0_{out} can be defined arbitrarily. (If the free-XOR method is used then the output wire key different than $k_{out,0}$ must be set to $k_{out,0} \oplus \Delta$ where Δ is constant for all gates, and cannot be set arbitrarily.)

In our new garbling method we use the freedom in choosing the second output wire key to always set it to $K_1 \oplus K_2 \oplus K_3$. As a result, and as will be explained below, the garbled table will have the property that entry T_3 of the table satisfies $T_3 = T_1 \oplus T_2$. Therefore T_3 can be computed in run-time by the evaluator and need not be stored or sent. In summary, garbling is carried out as follows:

- If $k_{out,0} = k^0_{out}$ then $k^0_{out} = K_0$ and $k^1_{out} = K_1 \oplus K_2 \oplus K_3$

- Else, $k^1_{out} = K_0$ and $k^0_{out} = K_1 \oplus K_2 \oplus K_3$

This fully defines the garbled table, as follows:

- If $k_{out,1} = k^0_{out} = K_0$ then $T_1 = K_0 \oplus K_1$ (since $K_0 = k_{out} = K_1 \oplus T_1$). Else, we have $k_{out,1} = K_1 \oplus K_2 \oplus K_3$ implying that $T_1 = K_2 \oplus K_3$.
- If $k_{out,2} = k^0_{out} = K_0$ then $T_2 = K_0 \oplus K_2$ (since $K_0 = k_{out} = K_2 \oplus T_2$). Else, we have $k_{out,2} = K_1 \oplus K_2 \oplus K_3$ implying that $T_2 = K_1 \oplus K_3$.

See Table 3 for the full definition of the garbled table $[T_1, T_2]$ and the definition of the output wires, depending on the permutation (recall that s is the index such that $K_s = k^1_{out}$). It is easy to verify correctness by tracing the computation in each case according to the table.

s	truth table	T_1	T_2	k^0_{out}	k^1_{out}
3	0001	$K_0 \oplus K_1$	$K_0 \oplus K_2$	K_0	$K_1 \oplus K_2 \oplus K_3$
2	0010	$K_0 \oplus K_1$	$K_1 \oplus K_3$	K_0	$K_1 \oplus K_2 \oplus K_3$
1	0100	$K_2 \oplus K_3$	$K_0 \oplus K_2$	K_0	$K_1 \oplus K_2 \oplus K_3$
0	1000	$K_2 \oplus K_3$	$K_1 \oplus K_3$	$K_1 \oplus K_2 \oplus K_3$	K_0

Table 3: Garbling the gate table

An alternative way to verify that the new scheme is correct is to observe that the output wire key computed for K_3 is always

$$\begin{aligned} k_{out,3} &= K_3 \oplus T_1 \oplus T_2 \\ &= K_3 \oplus (k_{out,1} \oplus K_1) \oplus (k_{out,2} \oplus K_2) \\ &= K_1 \oplus K_2 \oplus K_3 \oplus k_{out,1} \oplus k_{out,2} \end{aligned}$$

If $k_{out,1} \neq k_{out,2}$ then $k_{out,1} \oplus k_{out,2} = K_0 \oplus (K_1 \oplus K_2 \oplus K_3)$. In this case, $k_{out,3}$ should equal K_0 (since one of $k_{out,1}, k_{out,2}$ equals k^1_{out} and thus $k^3_{out} = k^0_{out}$), and this indeed follows from the equation.

If $k_{out,1} = k_{out,2}$, then by the equation we have that $k_{out,3} = K_1 \oplus K_2 \oplus K_3$. If $k_{out,3} = k^1_{out}$ then this is correct since $k^0_{out} = K_0$. Furthermore, if $k_{out,3} = k^0_{out}$ then since $k_{out,1} = k_{out,2}$ they both also equal k^0_{out}. This implies that $k_{out,0} = k^1_{out} = K_0$ and so $k_{out,3} = K_1 \oplus K_2 \oplus K_3$, as required. Intuitively, the first case (where $k_{out,3} = k^1_{out}$) corresponds to the case that the 0-key is K_0 and the 1-key is $K_1 \oplus K_2 \oplus K_3$, whereas the second case (where $k_{out,3} = k^0_{out}$) corresponds to the case that the 0-key is $K_1 \oplus k_2 \oplus K_3$ and the 1-key is K_0.

Encoding the permutation bits. The permutation bits can be encoded in a similar way to that suggested in [20]. Two changes are applied to the basic garbling scheme:

- The garbled values are only $n-1$ bits long, whereas the values K_i are still n bits long (concretely here, we use $n = 128$). Therefore, the function used for generating the K_i inputs has $n-1$-bit inputs and an n-bit output. We denote the least significant bit of K_i by m_i. Only $n-1$ bits of K_i are used for computing the garbled key of the output wire, using the procedure described above. Consequently, the values T_1, T_2 of the garbled table are also only $n-1$ bits long.
- We add 4 bits to the table. The ith of these bits is the XOR of m_i with the permutation bit of the corresponding output value.

The total length of a gate table is now $2(n-1)+4 = 2n+2$ bits (concretely 258 bits). The evaluation of a gate is performed by computing K_i; using its most significant $n-1$ bits for computing the corresponding garbled output value; and using its least significant bit m_i for computing the corresponding signal bit.

As for security, note that the m_i bits are pseudorandom, and are used only for the encryption of the permutation/signal values.

Intuition for Security. Recall that the 4-to-3 garbled row reduction scheme enables an arbitrary choice of the output wire key that is not $k_{out,0}$. The new 4-to-2 garbled row reduction scheme that we present is a special case, where we define that output wire key to be equal to $K_1 \oplus K_2 \oplus K_3$. Note that the evaluator can compute one of the K_i values using the two keys it holds, and can obtain two of the other three using T_1, T_2. However, in order to learn the *other* output wire key it needs the one K_i value that it cannot compute. Thus, from the point of view of the evaluator, the other output wire key, is a random string as required.

4.2 The Garbling Scheme

The changes need to be made at the Garble and Eval procedures in order to incorporate our 4-2 GRR technique are presented in Figure 9. We denote the improved scheme by G''.

4.3 Proof of Security

Next, we prove that the G'' satisfies the privacy requirement. As before, we present the modifications needed to the proof of Theorem 3.4.

Theorem 4.1 *If $\mathcal{F}$ is a family of pseudorandom functions, then the garbling scheme G'' achieves privacy.*

Proof Sketch: Let $\mathcal{S}''$ be a simulator that is identical to the simulator $\mathcal{S}$ from the proof of Theorem 3.4 (or to $\mathcal{S}'$ from Theorem 3.5 if the XOR gates are computed as in G'), except that when $\mathcal{S}''$ needs to simulate the garbling of an AND gate, holding the active keys k_i, k_j and signal bits λ_i, λ_j, it does the following:

1. $\mathcal{S}''$ computes $K||m := F_{k_i}(g||\lambda_i\lambda_j) \oplus F_{k_j}(g||\lambda_i\lambda_j)$
2. $\mathcal{S}''$ sets the output wire active key and signal bit:
   (a) $\lambda_\ell \leftarrow \{0,1\}$
   (b) If $2\lambda_i + \lambda_j = 0$, then set $k_\ell := K$
   (c) Else, set $k_\ell \in \{0,1\}^n$
3. Set T_1, T_2:
   (a) If $2\lambda_i + \lambda_j = 0$, set $T_1, T_2 \leftarrow \{0,1\}^n$
   (b) If $2\lambda_i + \lambda_j = 1$, set $T_1 := K \oplus k_\ell$ and $T_2 \leftarrow \{0,1\}^n$
   (c) If $2\lambda_i + \lambda_j = 2$, set $T_1 \leftarrow \{0,1\}^n$ and $T_2 := K \oplus k_\ell$
   (d) If $2\lambda_i + \lambda_j = 3$, set $T_1 \leftarrow \{0,1\}^n$ and $T_2 := K \oplus k_\ell \oplus T_1$

4. Compute the additional 4 bits: set $t_{2\lambda_i+\lambda_j} := m \oplus \lambda_\ell$, and for $\alpha \in \{0,1,2,3\} \setminus (2\lambda_i + \lambda_j)$ set $t_\alpha \leftarrow \{0,1\}$

5. Set $C[g] \leftarrow T_1, T_2, t_0, t_1, t_2, t_3$

Note that in the AND gates generated by $\mathcal{S}''$ code, the ciphertexts are computed so that the result of Eval will always be k_ℓ. (For example, according to Eval, if $2\lambda_i+\lambda_j = 1$ then k_ℓ is computed as $K \oplus T_1$. In such a case, $\mathcal{S}''$ sets $T_1 := K \oplus k_\ell$ and thus indeed $K \oplus T_1 = k_\ell$.) Beyond this constraint, the values are uniformly random. In particular, if $2\lambda_i + \lambda_j = 0$ then both ciphertexts are random, and otherwise the single ciphertext not used in Eval is random. In addition, the 4 bits that mask the output wire permutation bits are chosen randomly except for the the bit that is pointed to by the input wire's active signal bits.

We now define a hybrid H_i as in the proof of Theorem 3.4, and construct an adversary $\mathcal{A}_i$ for the experiment Expt^{2PRF}. Adversary $\mathcal{A}_i$ garbles the first $i-1$ gates in topological order using the instructions of $\mathcal{S}''$. When $\mathcal{A}_i$ reaches the ith gate with input wires $\boldsymbol{a}, \boldsymbol{b}$ and output wire $\boldsymbol{c}$, it holds two active keys $k_a^{v_a}, k_b^{v_b}$, two signal bits λ_a, λ_b and the actual bits that are on the wire v_a, v_b. Now, if g is a XOR gate, the $\mathcal{A}_i$ garbled the gate exactly as in the proof of Theorem 3.4 (or Theorem 3.5). If g is an AND gate, then $\mathcal{A}_i$ works as follows:

1. $\mathcal{A}_i$ computes:
$$\begin{aligned}
K_{2\lambda_a+\lambda_b}||m_{2\lambda_a+\lambda_b} &:= F_{k_a^{v_a}}(g||\lambda_a\lambda_b) \oplus F_{k_b^{v_b}}(g||\lambda_a\lambda_b)\\
K_{2\lambda_a+\overline{\lambda}_b}||m_{2\lambda_a+\overline{\lambda}_b} &:= F_{k_a^{v_a}}(g||\lambda_a\overline{\lambda}_b) \oplus \mathcal{O}^{(3)}(g||\lambda_a\overline{\lambda}_b)\\
K_{2\overline{\lambda}_a+\lambda_b}||m_{2\overline{\lambda}_a+\lambda_b} &:= \mathcal{O}^{(1)}(g||\overline{\lambda}_a\lambda_b) \oplus F_{k_b^{v_b}}(g||\overline{\lambda}_a\lambda_b)\\
K_{2\overline{\lambda}_a+\overline{\lambda}_b}||m_{2\overline{\lambda}_a+\overline{\lambda}_b} &:= \mathcal{O}^{(1)}(g||\overline{\lambda}_a\overline{\lambda}_b) \oplus \mathcal{O}^{(3)}(g||\overline{\lambda}_a\overline{\lambda}_b)
\end{aligned}$$

2. $\mathcal{A}_i$ computes the location of '1' in the truth table: $s := 2\overline{\pi}_a + \overline{\pi}_b = 2(\overline{v_a \oplus \lambda_a}) + (\overline{v_a \oplus \lambda_a})$

3. $\mathcal{A}_i$ runs steps (3)–(5) from procedure GbAND in G''

4. $\mathcal{A}_i$ outputs the garbled table $C[g] \leftarrow T_1, T_2, t_0, t_1, t_2, t_3$

It clear that when $\sigma = 0$ in Expt^{2PRF}, the result is identical to the real scheme G''. In contrast, when $\sigma = 1$, the answers of the oracles are random strings, and we have that all of the K values are independent random strings except for the value of $K_{2\lambda_a+\lambda_b}$ which $\mathcal{A}_i$ computes by itself using the keys it holds. (Observe that in each of the K values except for $K_{2\lambda_a+\lambda_b}$, oracles $\mathcal{O}^{(3)}$ and $\mathcal{O}^{(1)}$ are invoked on different inputs, resulting in independent random outputs.)

The output from the garbling of the gate by $\mathcal{S}''$ is k_c, λ_c (the active key used in garbling gates with wire c along with its signal) and the garbled table $T_1, T_2, t_0, t_1, t_2, t_3$. In contrast, the output of $\mathcal{A}_i$ is k_c^0, k_c^1, π_c along with $T_1, T_2, t_0, t_1, t_2, t_3$. Since the actual value v_c on the output wire is given, we can compute the actual signal bit λ_c (which equals $\pi_c \oplus v_c$) and the active wire $k_c^{v_c}$. Thus, we need to show that the *joint distribution* over $(k_c, \lambda_c, T_1, T_2, t_0, t_1, t_2, t_3)$ generated by $\mathcal{S}''$ in this case of $\sigma = 1$ is identical to the joint distribution over $(k_c^{v_c}, \lambda_c, T_1, T_2, t_0, t_1, t_2, t_3)$ generated by $\mathcal{A}_i$. In order to understand the following, we remark that given $2\lambda_a + \lambda_b$ (the row pointed to by the signal bits) and s (the row in which the '1'-key is "encrypted"), the active key on the output wire can be determined. This is because if $s = 2\lambda_a + \lambda_b$ then the active key on the output wire is k_c^1 (since the signal bits point to the 1-key), and otherwise it is k_c^0 (since the signal bits point to the 0-key).

Garble$(1^n, c)$**:**
Procedure GbAND$(k_i^0, k_i^1, k_j^0, k_j^1, \pi_i, \pi_j)$**:**

1. Compute: $K_0||m_0 := F_{k_i^{\pi_i}}(g||00) \oplus F_{k_j^{\pi_j}}(g||00)$ $\quad K_2||m_2 := F_{k_i^{\overline{\pi}_i}}(g||10) \oplus F_{k_j^{\pi_j}}(g||01)$
   $K_1||m_1 := F_{k_i^{\pi_i}}(g||01) \oplus F_{k_j^{\overline{\pi}_j}}(g||01)$ $\quad K_3||m_3 := F_{k_i^{\overline{\pi}_i}}(g||11) \oplus F_{k_j^{\overline{\pi}_j}}(g||11)$
2. Compute the location of '1' in the truth table: $s := 2\overline{\pi}_i + \overline{\pi}_j$
3. Set the output wire keys and permutation bits:
   (a) Choose the permutation bit for the wire: $\pi_\ell \leftarrow \{0,1\}$
   (b) If $s \neq 0$, set $k_\ell^0 := K_0$ and $k_\ell^1 := K_1 \oplus K_2 \oplus K_3$
   (c) Else, (if $s = 0$), set $k_\ell^0 := K_1 \oplus K_2 \oplus K_3$ and $k_\ell^1 := K_0$
4. Compute T_1, T_2:
   (a) If $s = 3$, set $T_1 := K_0 \oplus K_1$ and $T_2 := K_0 \oplus K_2$
   (b) If $s = 2$, set $T_1 := K_0 \oplus K_1$ and $T_2 := K_1 \oplus K_3$
   (c) If $s = 1$, set $T_1 := K_2 \oplus K_3$ and $T_2 := K_0 \oplus K_2$
   (d) If $s = 0$, set $T_1 := K_2 \oplus K_3$ and $T_2 := K_1 \oplus K_3$
5. Compute the additional 4 bits: set $t_s := m_s \oplus \overline{\pi}_\ell$, and for $\alpha \in \{0,1,2,3\} \setminus \{s\}$ set $t_\alpha := m_\alpha \oplus \pi_\ell$
6. Return $(k_\ell^0, k_\ell^1, \pi_\ell, T_1, T_2, t_0, t_1, t_2, t_3)$

Note that GbAND returns 2 ciphertexts and 4 bits (instead of 3 ciphertexts as in G).

Procedure Eval(C, X)**:**

2. (b) If g is an AND gate with inputs wires i, j and output wire ℓ (and table $T_1, T_2, t_0, t_1, t_2, t_3$):
   i. Compute: $K||m := F_{k_i}(g||\lambda_i\lambda_j) \oplus F_{k_j}(g||\lambda_i\lambda_j)$
   ii. Compute the output wire key:
      A. If $2\lambda_i + \lambda_j = 0$, set $k_\ell := K$
      B. If $2\lambda_i + \lambda_j = 1$, set $k_\ell := K \oplus T_1$
      C. If $2\lambda_i + \lambda_j = 2$, set $k_\ell := K \oplus T_2$
      D. If $2\lambda_i + \lambda_j = 3$, set $k_\ell := K \oplus T_1 \oplus T_2$
   iii. Compute the output wire signal bit: $\lambda_\ell := m \oplus t_{2\lambda_i + \lambda_j}$

Figure 9: The Improved Garbling Scheme G''

We consider four cases:

1. *Case 1* – $2\lambda_a + \lambda_b = 0$*:* In this case, $\mathcal{A}_i$ computes K_0 using keys $k_a^{v_a}, k_b^{v_b}$ whereas K_1, K_2, K_3 are independent random strings. Now, in this case, $\mathcal{S}''$ sets $k_c := K$ where K is computed exactly like K_0 by $\mathcal{A}_i$. In addition, $\mathcal{S}''$ chooses $T_1, T_2 \leftarrow \{0,1\}^n$ at random. Since K_1, K_2, K_3 are independent and random, it follows that all four ways of setting T_1, T_2 depending on s that are described in Step 4 of GbAND of G'' yield two independent keys. Thus, K_0, T_1, T_2 generated by $\mathcal{A}_i$ are distributed identically to K_0, T_1, T_2 generated by $\mathcal{S}''$. Now, if $s \neq 0$, then $\mathcal{A}_i$ sets $k_c^0 = K_0$ while if $s = 0$ then $\mathcal{A}_i$ sets $k_c^1 = K_0$. Since $2\lambda_a + \lambda_b = 0$ it follows that if $s \neq 0$ then the active key is k_c^0 and if $s = 0$ then the active key is k_c^1. Thus, in both cases the active output key is K_0, exactly like $\mathcal{S}''$.

2. *Case 2 – $2\lambda_a + \lambda_b = 1$:* In this case, $\mathcal{A}_i$ computes K_1 using keys $k_a^{v_a}, k_b^{v_b}$ whereas K_0, K_2, K_3 are independent random strings. When $s \in \{2,3\}$, the active key k_c^0 on the output wire is set by $\mathcal{A}_i$ in Step 3 to be equal to K_0, and $T_1 = K_0 \oplus K_1$. When $s \in \{0,1\}$ the active key on the output wire equals $K_1 \oplus K_2 \oplus K_3$ (because when $s = 0$, the active key is $k_c^0 := K_1 \oplus K_2 \oplus K_3$ while when $s = 1$, the active key is $k_c^1 := K_1 \oplus K_2 \oplus K_3$), and $T_1 = K_2 \oplus K_3$. In both cases, we have that $K_1 \oplus T_1$ equals the active key on the output wire. In addition, in all cases T_2 is computed by $\mathcal{A}_i$ by XORing two strings, of which at least one of them is random and independent of T_1 and K_1. (To be exact, T_2 is actually the XOR of one of the output-wire keys with K_2. However, since K_2 is random, and since there is at least one random string that appears in T_1 or T_2 but not in both, we have that T_2 is completely independent of all other values.) In summary, k_c, T_1, T_2 are all random strings under the constraint that $k_c = T_1 \oplus K_1$.

   In contrast, $\mathcal{S}''$ sets k_c to be random, sets $T_1 = K \oplus k_c$ and T_2 to be random. Thus, $K \oplus T_1$ equals the active key on the output wire, and we have that k_c, T_1, T_2 are also all random under the constraint that $k_c = T_1 \oplus K$. Thus, the distributions are identical.

3. *Case 3 – $2\lambda_a + \lambda_b = 2$:* In this case, $\mathcal{A}_i$ computes K_2 using keys $k_a^{v_a}, k_b^{v_b}$ whereas K_0, K_1, K_3 are independent random strings. Using the same analysis as the previous case, we obtain that when $s \in \{0,2\}$ the active key on the output wire is set by $\mathcal{A}_i$ to be $K_1 \oplus K_2 \oplus K_3$ (because when $s = 2$, the active key is k_c^1, while when $s = 0$ the active key is k_c^0; in both cases it equals $K_1 \oplus K_2 \oplus K_3$), and $T_2 = K_1 \oplus K_3$. In contrast, when $s \in \{1,3\}$, the active key k_c^0 on the output wire is set to be K_0, and $T_2 = K_0 \oplus K_2$. Denoting the active key by k_c in all cases, we have that k_c and T_2 are random under the constraint that $k_c \oplus T_2 = K_2$. In all cases, as in the previous case with T_2, ciphertext T_1 is random and independent of k_c, T_2 since it involves an independent random value each time (K_0, K_1 or K_3). Thus, k_c, T_1, T_2 are independent random strings, under the constraint that $k_c \oplus T_2 = K_2$.

   Regarding $\mathcal{S}''$, it chooses k_c and T_1 uniformly at random, and sets $T_2 = K \oplus k_c$. Thus, k_c, T_1, T_2 have exactly the same distribution as that generated by $\mathcal{A}_i$.

4. *Case 4 – $2\lambda_a + \lambda_b = 3$:* In this case, $\mathcal{A}_i$ computes K_3 using keys $k_a^{v_a}, k_b^{v_b}$ whereas K_0, K_1, K_2 are independent random strings. In this case, if $s \in \{0,3\}$ then the active key k_c on the output wire is set by $\mathcal{A}_i$ to be $K_1 \oplus K_2 \oplus K_3$ (since if $s = 0$ the active key is k_c^0 whereas if $s = 3$ the active key is k_c^1), and $T_1 \oplus T_2 = K_1 \oplus K_2$ (see Step 4 in GbAND). Furthermore, if $s \in \{1,2\}$ then the active key k_c on the output wire is K_0 and $T_1 \oplus T_2 = K_0 \oplus K_3$. In both cases, $k_c \oplus T_1 \oplus T_2 = K_3$. Apart from this constraint, the values are random. Thus, we have that k_c, T_1, T_2 are random under the constraint that $k_c \oplus T_1 \oplus T_2 = K_3$.

   Regarding $\mathcal{S}''$, in this case it chooses k_c and T_1 independently at random and sets $T_2 = K \oplus k_c \oplus T_1$. Thus, as above, k_c, T_1, T_2 are random under the constraint that $k_c \oplus T_1 \oplus T_2 = K_3$.

We conclude that k_c, T_1, T_2 is identically distributed when generated by the adversary $\mathcal{A}_i$ in the case of $\sigma = 1$ and when generated by the simulator $\mathcal{S}''$ (note that $K_{2\lambda_a+\lambda_b}$ is always the exact same value since it is fixed by the incoming keys). In addition, in A_i's code all of the m values, except for the value of $m_{2\lambda_a+\lambda_b}$ are random. Thus, the bits t_1, t_2, t_3, t_4 are random except for $t_{2\lambda_a+\lambda_b}$, and the distribution over their values is the same as when they are generated by $\mathcal{S}''$. We conclude that when $\sigma = 1$, adversary $\mathcal{A}_i$ constructs gate i exactly according to $\mathcal{S}''$.

The remaining gates $j > i$ are garbled using the real garbling scheme G'', with $\mathcal{A}_i$ using its oracles $O^{(2)}, O^{(4)}$ to garble the other gates which wires a and b enters. We conclude that when

$\sigma = 0$, adversary $\mathcal{A}_i$ constructs the H_{i-1} hybrid, while when $\sigma = 1$ it constructs the H_i hybrid. The rest of the proof is the same as the proof of Theorem 3.4. ■

5 Garbling With Related-key Security

5.1 Background

When using the free-XOR technique, a constant difference is used between the garbled values on every wire (i.e., there exists a random Δ such that for every wire i, $k_i^0 \oplus k_i^1 = \Delta$). As a result, the keys used for encryption in non-XOR gates are correlated with each other, and also with the plaintext that they encrypt (observe that Δ appears in the garbled values on both the input and output wires). Thus, a strong circularity related-key assumption is needed for proving that the technique is secure. As we have seen, if we want to rely on a pseudorandom function assumption only, then the keys on the wires have to be uniformly and independently chosen. In this section, we consider garbling schemes that rely on related keys, but do *not* require the stronger circularity assumption. In order to achieve this, keys on the input wires of each gate are allowed to be related, but no relation is allowed between input wire keys and the output wire keys they encrypt. This relaxation allows us to garble some of the XOR gates for free, and yields results that are better than when garbling under a pseudorandom function assumption only, but worse than garbling all the XOR gates for free which requires circularity. The work in this section builds strongly on the fleXOR technique of [13], and provides a more complete picture regarding the trade-off between efficiency and the security assumptions used in circuit garbling. (Specifically, we consider the cost of garbling under the hierarchy of assumptions, from ideal-cipher to circular related-key security to related-key security to a pseudorandom function assumption).

In the work of [13], they showed that in order to avoid circularity it suffices to apply a monotone rule on the *wire ordering* of the circuit. This monotone rule states that when a certain difference value Δ is used on the input wires to a non-XOR gate, then the Δ on the output wire must be different (actually, it has to be a Δ that has not appeared previously in the garbling of gates that are in the path to the current gate). Denote L different difference values by $\Delta_1, ..., \Delta_L$. Then, a wire ordering is defined to be a function ϕ that takes a wire as its input and returns an element of the group $\{1, .., L\}$ (with the interpretation that on wire i, the difference between the garbled values is $\phi(i)$). We formally define a monotone ordering as follows.

Definition 5.1 *Let C be a garbled circuit, and let I be the set of circuit wires. A wire ordering function $\phi : I \to \{1, .., L\}$ is called* monotone *if:*

1. *For every non-XOR gate with input wires i, j and output wire ℓ: $\phi(\ell) > max(\phi(i), \phi(j))$*
2. *For every XOR gate with input wires i, j and output wire ℓ: $\phi(\ell) \geq max(\phi(i), \phi(j))$*

Now, assume that a wire ordering was fixed, and consider a XOR gate g. If $\phi(\ell) = \phi(i) = \phi(j)$ then the gate is garbled and computed using the free-XOR technique. However, if $\phi(\ell) \neq \phi(i)$ (or likewise if $\phi(\ell) \neq \phi(j)$) then wire i's keys are translated into new keys $\tilde{k}_i^0, \tilde{k}_i^1$ such that $\tilde{k}_i^0 = F_{k_i^0}(g)$ and $\tilde{k}_i^1 = \tilde{k}_i^0 \oplus \Delta_{\phi(\ell)}$, yielding a garbled gate entry $F_{k_i^1}(g) \oplus \tilde{k}_i^1$ (to be more exact, the way of computing $\tilde{k}_i^0, \tilde{k}_i^1$ can be reversed, depending on the permutation bit). Once the input and output wires all have difference $\Delta_{\phi(\ell)}$, the free XOR technique can once again be used. It

follows that a translation from $\Delta_{input\ wire}$ to $\Delta_{output\ wire}$ can be carried out with one ciphertext and two encryptions. Thus XOR gates can be garbled using 0, 1 or 2 ciphertexts and using 0, 2 or 4 encryptions (for the cases where no translation is needed, where one translation is needed and where two translations are needed, in respectively), depending on the wire ordering that was chosen for the circuit. This is a "flexible approach" since many different wire orderings can be chosen, and hence its name "fleXOR".[9] Since the specific ordering determines the cost, this introduces a new *algorithmic goal* which is to find a monotone wire ordering that is optimal; i.e., that minimizes the size of the circuit while satisfying the monotone property.

Unfortunately, it is NP-hard to find an optimal monotone wire ordering [13]. Thus, [13] described heuristic techniques for finding a good monotone ordering. Briefly, their heuristic is based on the observation that only non-XOR gates increase the wire ordering number. They therefore define the non-XOR-depth of a wire i to be the maximum number of non-XOR gates on all directed paths from i to an output wire. Then, they set the wire ordering so that $\phi(i) + \text{non-XOR-depth}(i)$ is constant for all wires. Algorithmically, they set the wire ordering value of each XOR gate's output wire to be equal to the maximal ordering value of its input wires, and they make the wire ordering value of each AND gate's output to equal a value that maintains the constant. For more details, see [13].

5.2 Safe and Monotone Wire Orderings

The goal of constructing a good monotone wire ordering is to assign, whenever possible, the same wire ordering number to input wires and output wires of XOR gates, so that the communication and computation cost at XOR gates will be minimized. However, such a strategy is not compatible with 4-2 row reduction techniques (the technique in this paper and in [20] require that both output values be arbitrary unlike here, and the half-gates method of [22] works only under a circularity assumption which is exactly what we are trying to avoid here). Thus, an optimized monotone wire ordering may result in most AND gates being garbled with 3 ciphertexts (4-2 row reduction could be used in AND gates where the difference on the output wire is "new"). In circuits with many XOR gates relative to AND gates, such a strategy may be worthwhile. However, in circuits where there are more AND gates than XOR gates (like the SHA256 circuit), the result may be a larger circuit than that obtained by using our scheme based on pseudorandom functions alone that costs 2 ciphertexts per AND gate and 1 ciphertext per XOR gate.

This motivates the search for wire orderings that enable 4-2 row reduction in AND gates. Such a wire ordering is called *safe* and was defined by [13] for this purpose; intuitively, a wire ordering is safe if the values on the output wires of AND gates can be determined arbitrarily. Formally, we require that the Δ in the output of a non-XOR gate different to all previous gates, implying that it is not yet determined:

Definition 5.2 *Let C be a garbled circuit, and let I be the set of circuit wires. A wire ordering function $\phi : I \to \{1, .., L\}$ is called* safe *if for every non-XOR gate g with output wire ℓ, it holds that for every wire i that precedes it in the topological order of the circuit $\phi(i) < \phi(\ell)$.*

[9] This is the real difference between the fleXOR approach and our standard assumption based scheme: our scheme is not flexible; all XOR gates are garbled in the same way such that for each wire there will be an independent new offset that is set pseudorandomly when garbling each gate, and not in advance as in the fleXOR approach. As a result, our scheme requires 4 (or 3) encryptions even though only one ciphertext is required.

Note that a wire ordering that is safe does not necessarily avoid circularity; thus free-XOR together with half-gates will always be preferable (note that the notion of a safe ordering was introduced *before* the half-gates construction was discovered, and this made it redundant). Nevertheless, in order to both avoid circularity and potentially reduce the number of ciphertexts in AND gates, we are interested in wire orderings that are simultaneously *safe* and *monotone*. Such wire orderings were not considered in the work of [13], and we will dedicate the rest of this section to introducing two simple heuristics that satisfy these two properties.[10]

Safe and monotone heuristics. Our goal is to find a "good" safe and monotone wire ordering heuristic that will allow us to garble AND gates using our 4-2 row-reduction technique, and to garble XOR gates using the fleXOR approach. When we say a "good" heuristic, we mean that it minimizes the *average number of ciphertexts per XOR gate*. Recall that in the fleXOR approach, XOR gates are garbled using 0,1 or 2 ciphertexts. Thus, a wire ordering will only be reasonable if the average number of ciphertexts per XOR gate is lower than 1; otherwise, it is better to use the scheme presented earlier that garbles XOR gates with 1 ciphertext and under a pseudorandom function assumption only.

Observe that in a safe and monotone wire ordering, if there are L non-XOR gates in the circuit then there are $L+1$ different delta values $\{\Delta_0, \Delta_1, .., \Delta_L\}$, where Δ_0 is a *random* value that is set at the beginning of the garbling process and is assigned to the input wires of the circuit, and the rest of the Δ values are assigned to each non-XOR gate in topological order as determined by the garbling row-reduction method in the associated gate. We define two variables $\phi_i^{min}, \phi_i^{max}$ for every wire i in the circuit, where ϕ_i^{min} (resp., ϕ_i^{max}) is the minimal (resp., maximal) value that $\phi(i)$ can have in *any* safe and monotone wire ordering in the circuit. In Figure 10 we present an algorithm that computes the *exact* value of ϕ_i^{min} and ϕ_i^{max} for each wire.

Initialization algorithm:

1. Initialize $AND\_index := 0$
2. For every input wire i, set: $\phi_i^{min} = \phi_i^{max} := 0$
3. For every gate g in topological order (with input wires i, j and output wire ℓ):
   (a) If g is an AND gate:
      i. Set $gate\_index := AND\_index + 1$
      ii. Set $\phi_\ell^{min} = \phi_\ell^{max} := AND\_index$
   (b) If g is a XOR gate:
      i. Set $\phi_\ell^{min} := \max\{\phi_i^{min}, \phi_j^{min}\}$
      ii. Set $\phi_\ell^{max} := AND\_index$

Figure 10: Initialization algorithm for the safe and monotone wire ordering heuristic

To understand why the algorithm computes ϕ_i^{min} and ϕ_i^{max} correctly, recall that for each AND gate the wire ordering number is fixed in a safe wire ordering, and is equal to its index in the order of AND gates in the circuit. Thus, $\phi_\ell^{min} = \phi_\ell^{max} := AND\_index$ as set in step 3(a). For XOR

[10] We remark that the proof of [13] that the problem of finding an optimal monotone ordering is NP-hard does not go through for monotone and safe. We do not know if the problem of finding an optimal safe *and* monotone ordering is NP hard.

gates, note that in our algorithm, setting the value of ϕ_ℓ^{min} for each XOR gate's output wire to be the maximum ϕ of its inputs, means that it is increased only by AND gates that are in a path from a circuit input wire to the gate. Therefore, if there exists a wire ordering that assigns ϕ_ℓ^{min} a smaller value than our algorithm, it will assign it a wire ordering number that is smaller than a wire ordering number of an AND gate's output wire that is in a path that leads to it, thereby breaking monotonicity (as in Definition 5.1). In addition, since we cannot assign a wire with a Δ whose value will be determined at a later stage (due to the safe condition), the highest possible wire-ordering value that a XOR gate's output wire ℓ can have equals the number of AND gates that appears before it in the topological order of the circuit. This maximum value is exactly the value of the variable *AND_index* in the algorithm, which is assigned to ϕ_ℓ^{max} in step 3(b).

Next, observe that a heuristic that assigns each wire the value ϕ_i^{min}, as well as a heuristic that assigns each wire the value ϕ_i^{max}, both yield valid safe and monotone wire orderings. Moreover, taking ϕ_i^{min} at every gate ensures that each XOR gate will have at most one ciphertext (because the output-wire value equals at least one of the input-wire values). Thus, this heuristic – that we call the pure min-heuristic – guarantees that the average number of ciphertexts per XOR gate is *less than or equal to* 1.[11] This means that, not surprisingly, heuristics that yield a more efficient garbling scheme than our scheme based only on a pseudorandom function assumption do exist. However, our aim is to do better than the pure min-heuristic, and we suggest two heuristics that use ϕ_i^{min} and ϕ_i^{max} as initialization values for the wire ordering values, and then improve upon them by traversing the circuit gate by gate from the output to the input, and setting the value of each gate-output wire based on the existing values given so far.

The idea behind both heuristics is that, starting with the output wires and going backwards, we try to group as many wires as possible to have the same wire ordering number. We do this by trying to assign a wire i the same value as one of the output wires of a gate that it enter (specifically, we try give it the minimal value among all the values on the output wires that it enter; taking the minimal ensures monotonicity). When this fails – measured by the fact this yields a value not between ϕ_i^{min} and ϕ_i^{max} – we set its wire ordering number to be the maximum between the initialization values of its input wires.

In the first heuristic, called SafeMon1, each wire i is given the initial ordering value $\phi(i) = \phi_i^{min}$. Then, starting with the circuit's output wires and going backwards in reverse topological order, we compute for every wire i that is not a circuit-output wire:

$$\overline{\phi}_i := \min\left\{\phi(k) \mid \exists \text{ gate } g \text{ with input wire } i \text{ and output wire } k\right\}$$

This value is in fact the maximal value $\phi(i)$ can have without breaking monotonicity since the input wire i to a gate cannot have a higher value than its output wire ℓ. Then, we set $\phi(i) = \overline{\phi}_i$ if $\overline{\phi}_i \leq \phi_i^{max}$, and set $\phi(i) = \phi_i^{min}$ if $\overline{\phi}_i > \phi_i^{max}$ (this ensures that we don't break the safe property).

The second heuristic, called SafeMon2, works in the same way except that the wires are initialized with ϕ_i^{max} instead of ϕ_i^{min}. (Observe that in the initialization, $\phi_\ell^{min} = \max(\phi_i^{min}, \phi_j^{min}$ and thus in SafeMon1 setting $\phi_\ell = \phi_\ell^{min}$ is the same as setting it to be $\max\{\phi_i^{min}, \phi_j^{min}\}$ as in SafeMon2.) The full description of the heuristic appears in Figure 11.

[11] Note that taking ϕ_i^{max} at every gate does *not* guarantee at most 1 ciphertext per XOR gate.

1. Run the initialization heuristic to obtain for each wire i the values $\phi_i^{min}, \phi_i^{max}$
2. For every circuit output wire i set: $\phi(i) := \begin{cases} \phi_i^{min} & \text{for SafeMon1} \\ \phi_i^{max} & \text{for SafeMon2} \end{cases}$
3. For every wire g in **reverse** topological order (with input wires i, j and output wire ℓ):
   If ℓ is not a circuit output wire, then:
   (a) Compute $\overline{\phi}_\ell := \min\left\{\phi(k) \mid \exists \text{ gate } g \text{ with input wire } \ell \text{ and output wire } k\right\}$
   (b) If $\overline{\phi}_\ell \leq \phi_\ell^{max}$ then, set: $\phi(\ell) := \overline{\phi}_\ell$
   Else, set: $\phi(\ell) := \begin{cases} \phi_\ell^{min} & \text{for SafeMon1} \\ \max\left\{\phi_i^{max}, \phi_j^{max}\right\} & \text{for SafeMon2} \end{cases}$

Figure 11: Heuristics for finding safe and monotone wire orderings

5.3 Choosing the Best Heuristic

We have described three heuristics for generating wire orderings that yield garbled circuits that are secure based on a related-key assumption and without circularity (the monotone heuristic from [13] and two new heuristic SafeMon1 and SafeMon2). We ran these heuristics on three different circuits, and compared the results with our garbling scheme based only on pseudorandom functions and with the monotone heuristic of [13]. The circuits we tested the heuristics on are AES, SHA-256 and Min-Cut 250,000. The circuits have 6,800, 90,825 and 999,960 AND gates, respectively, and 25,124, 42,029 and 2,524,920 XOR gates, respectively [1]. The performance of each heuristic was measured by the size of the circuit it yields. Table 4 shows the results of the comparison. It can be seen that the monotone heuristic of [13] gives the best result for the AES circuit, while the SafeMon2 heuristic yields the smallest garbled circuit for the SHA-256 and Min-Cut circuits. Observe that in the SHA-256 circuit, which has a high percentage of AND gates, all the heuristics fail to significantly reduce the size of the garbled circuit, relative to the size of the circuit constructed under the pseudorandom function assumption only. This is due to the fact that in such circuits the high amount of AND gates impose many constraints on the wire ordering, and so we are forced to have many different deltas that are "spread" between a small amount of XOR gates. In such cases, not much is gained by using a related-key assumption, versus pseudorandom functions only.

As we described above, safe and monotone heuristics are expected to beat the pure monotone heuristic of [13] only when there are more AND gates than XOR gates. Indeed, if we measure only the effect on XOR gates (see the numbers in the parentheses in Table 4), then the monotone-only heuristic always results in less ciphertexts on average for the XOR gates. This is because it imposes less constraints on the wire ordering and focuses only on the XOR gates. In the AES circuit, where there are only 6800 AND gates and 25124 XOR gates, the average number of ciphertexts per XOR gates is only 0.15 (which is very impressive). Thus, even though the AND gates require 3 ciphertexts each, the overall result is the best. The safe and monotone heuristics that we present here also take into consideration the cost of AND gates (at the expense of XOR gates), and so achieve better results when there is a higher percentage of AND gates.

In Table 5, we show the computation cost of garbling XOR gates (number of pseudorandom function computations) when using the wire orderings that each heuristic generated. We only consider XOR gates in this computation, since all the methods used for garbling AND gates require the same computational work. Recall that in the fleXOR method, garbling XOR gates may need 0, 2 or 4 calls to the pseudorandom function. Thus, the computation cost is measured by the

	AES	**SHA-256**	**Min-Cut**
PRF only	1.21 (1)	1.68 (1)	1.28 (1)
Monotone [13]	**0.76 (0.15)**	2.26 (0.76)	1.13 (0.4)
SafeMon1	0.93 (0.64)	1.64 (0.97)	1.19 (0.87)
SafeMon2	1.19 (0.97)	**1.53 (0.86)**	**1.07 (0.7)**

Table 4: Comparison of the size of the garbled circuit that each heuristic generated (including a base comparison to the cost under a pseudorandom function assumption only). The main number in each cell shows the average number of ciphertexts per gate. The number in the parentheses shows the average ciphertexts per XOR gate only. The best result for each circuit is bolded.

average number of calls to the encryption function per XOR gate. Observe that for this measure, all heuristics are considerably better than the scheme relying only on a pseudorandom function assumption. This is because in the fleXOR approach, only two calls to the pseudorandom function are needed when garbling a XOR gate with one ciphertext, in contrast to three in the scheme based only on pseudorandom functions. However, we remark that when using AES-NI and pipelining, this actually makes little difference to the overall time. Also, observe that the monotone heuristic is better than the other heuristics when comparing computational cost. As explained before, this is because the monotone heuristic focuses solely on minimizing the cost at XOR gates, rather than minimizing the cost for the entire circuit, thus achieving better results when measuring the effect on XOR gates only.

	AES	**SHA-256**	**Min-Cut**
PRF only	3	3	3
Monotone [13]	**0.31**	**1.55**	**0.79**
SafeMon1	1.29	1.95	1.75
SafeMon2	1.96	1.73	1.41

Table 5: Comparison of the average number of calls to the pseudorandom function per XOR gate, for each heuristic

We conclude that in circuits with many XOR gates relative to AND gates, the use of a related-key assumption yields an improvement over the scheme relying on pseudorandom functions only. For example, in the AES circuit the smallest result is 24% smaller, and in the min-cut circuit the size of the circuit is approximately 16% smaller.

Optimal algorithms. We stress that we did not prove anything regarding the optimality of the heuristics we described. Indeed, adding the requirement that the ordering be safe is just a way to force the heuristic to take AND gates into account. However, it is possible that a better result can be achieved with an ordering that is not safe. However, as we have mentioned, finding an optimal monotone ordering is NP-hard. Thus, finding better heuristics or optimization algorithms is left for future work.

6 Experimental Results and Discussion

In the previous sections, we presented four tools that can optimize the performance of garbled circuits without relying on any additional cryptographic assumption beyond the existence of pseudorandom functions: (1) *pipelined garbling*; (2) *pipelined key-scheduling*; (3) *XOR gates with one ciphertext and three encryptions*; and (4) *improved 4-2 GRR for AND gates*. In this section, we present the results of an experimental evaluation of these methods – together and separately – and compare their performance to that of other garbling methods.

Table 6 shows the time it takes to run the *full Yao semi-honest protocol* [17, 21] on three different circuits of interest: AES, SHA-256 and Min-Cut 250,000. The circuits have 6,800, 90,825 and 999,960 AND gates, respectively, and 25,124, 42,029 and 2,524,920 XOR gates, respectively. The number of input bits for which OTs are performed are 128, 256 and 250,000, respectively [1]. We remark that our implementation of the semi-honest protocol of Yao utilizes the highly optimized OT extension protocol of [2].

We examined eight different schemes, described using the following notation: [**pipe-garble**] for the pipelined garbling method; [**pipe-garble+KS**] for pipelined garbling and pipelined key-scheduling; [**fixed-key**] where all PRF evaluations were performed using the fixed-key technique described in [5]; [**XOR-3**] where XOR gates were garbled using a simple 4-3 GRR method; [**XOR-1**] where XOR gates were garbled using our method of garbling with one ciphertext; [**free-XOR**] where the free-XOR technique was used; [**AND-3**] where AND gates were garbled using simple 4-3 GRR; [**AND-2**] where our 4-2 GRR method was used to garble AND gates; and finally, [**AND-HalfGates**] where the "half-gates" technique of [22] was used to garble AND gates. Note that the half-gates method is only used in conjunction with free-XOR since this is a requirement.

The first scheme in Table 6 is the most "naïve", where a simple 4-3 GRR was used for both AND and XOR gates and the garbling was pipelined but not the key-scheduling. In contrast, the last scheme is the most efficient as it uses fast fixed-key encryption and the half-gates approach to achieve two ciphertexts per AND gates and none for XOR gates. However, this scheme is based on the strongest assumption, that AES behaves like an "ideal-cipher", or to be more exact, like a "random permutation" with a fixed key. The third scheme in the table uses all our optimizations together, and thus it is the most efficient scheme that is based on a standard PRF assumption. The sixth scheme in the table shows the best that can be achieved while assuming circularity and related key security, but without resorting to the ideal cipher.

The experiments were performed on Amazon's c4.8xlarge compute-optimized machines (with

Assumption		**Scheme**	AES		SHA-256		Min-Cut
			VA-VA	VA-IRE	VA-VA	VA-IRE	VA-VA
PRF	1	Pipe-garbl; XOR-3; AND-3 (naïve)	20	203	68	303	1947
	2	Pipe-garbl+KS; XOR-1; AND-3	16	200	54	236	1195
	3	**Pipe-garbl+KS; XOR-1; AND-2**	**16**	**200**	**50**	**229**	**1047**
Circularity	4	Pipe-garbl; free-XOR; AND-3	16	198	45	222	753
	5	Pipe-garbl+KS; free-XOR; AND-3	16	198	36	221	701
	6	**Pipe-garbl+KS; free-XOR; HalfGates**	**16**	**196**	**27**	**206**	**546**
Ideal Cipher	7	Fixed-key; free-XOR; AND-3	16	196	27	214	596
	8	**Fixed-key; free-XOR; Halfgates**	**16**	**195**	**20**	**199**	**460**

Table 6: Summary of experimental results (times are for a full semi-honest execution in milliseconds). The first row is for naïve garbling. Rows 2,3,5 and 6 are based on our improvements. The rows marked in boldface highlight the best schemes under each set of assumptions.

Intel Xeon E5-2666 v3 Haswell processors) running Windows. The measurements include the time it takes to garble the circuit, send it to the evaluator and compute the output. Since communication is also involved, this measures improvements both in the encryption technique and in the size of circuit. Each scheme was tested on the three circuits in two different settings: the *Virginia-Virginia* (VA-VA) setting where the two parties running the protocol are located at the same data center, and the *Virginia-Ireland* (VA-IRE) setting where the physical distance between the parties is large. (We omitted the results of running the large min-cut circuit in the VA-IRE setting as they were not consistent and had a high variability.) Each number in the table is an average of 20 executions of the indicated specific scenario.

The table rows marked in boldface highlight the best schemes under each set of assumptions. Looking at the results, we derive the following observations:

- **Best efficiency:** As predicted, the fixed key + half-gates implementation (8) is the fastest and most efficient in all scenarios. (This seems trivial, but when using fixed-key AES, the Eval procedure at AND gates requires one more encryption than in a simple 4-3 GRR. Thus, this confirms the hypothesis that the communication saved is far more significant than an additional encryption, that is anyway pipelined.)

- **Small circuits:** In small circuits (e.g., AES) the running time is almost identical in all schemes and in both communication settings. In particular, using our optimizations (3) yields the same performance result as that of the most efficient scheme (8), in both the VA-VA and VA-IRE settings. This is due to the fact that in small circuits, running the OT protocol is the bottleneck of the protocol (even if, as in our experiments, optimize OT-extension [2] is used). This means that for small circuits there is no reason to rely on a non-standard cryptographic assumption.

- **Medium circuits:** In the larger SHA-256 circuit, where the majority of the gates are AND gates, there was a difference between the results in the two communication settings. In the VA-VA setting the best scheme based on PRF alone (3) has performance that is closer to that of the naïve scheme (1) than to that of the schemes based on the circularity or the ideal cipher assumptions (schemes 6 and 8). In contrast, in the VA-IRE setting the PRF based scheme performs close to schemes 6 and 8. This is explained by observing that when the parties are closely located, communication is less dominant and garbling becomes a bigger factor. Thus, garbling XOR gates for free improves the performance of the protocol. In contrast, when the parties are far from each other, communication becomes the bottleneck, thus the PRF based scheme (3) yields a significant improvement compared to the naïve case (1) and its performance is not much worse than that of the best fixed-key based scheme (and since there are fewer XOR gates, the overhead of an additional ciphertext per gate is reasonable).

- **Large circuits:** In the large Min-Cut circuit, the run time of our best PRF based scheme (3) is closer to the best result (8) than to the naïve result (1). This is explained by the fact that the circuit is very large and so bandwidth is very significant. This is especially true since the majority of gates are XOR gates, and so the reduction from 3 ciphertexts to 1 ciphertext per XOR gate has a big influence. (Observe that the number of ciphertexts sent in (8) is 2,000,000, the number of ciphertexts sent in (3) is 4,500,000, while the number of ciphertexts sent in (1) is 11,500,000.) Observe that schemes (6) and (8) have the same bandwidth; the difference in cost is therefore due to the additional cost of the AES key schedules and encryptions. Note,

however, that despite the fact that there are 1,000,000 AND gates, the difference between the running-times is 15%, which is not negligible but also not overwhelming.

- **Removing the ideal cipher assumption:** Comparing scheme (8), which is the most efficient, to scheme (6) which is the most efficient scheme that does not depend on the ideal cipher assumption, shows that in all scenarios removing the fixed-key technique causes only a minor increase in running time.

We conclude that strengthening security by removing the ideal-cipher assumption does not noticeably affect the performance of the protocol. Thus, in many cases, two-party secure computation protocols does not need to use the fixed-key method. Further security strengthening by not depending on a circularity assumption (i.e., "paying" for XOR gates) does come with a cost. Yet, in scenarios where garbling time is not the bottleneck (e.g.,, small circuits, large inputs, communication constraints), one should consider using a more conservative approach as suggested in this work. In any case, we believe that our ideas should encourage future research on achieving faster and more efficient secure two-party computation based on standard cryptographic assumptions.

Acknowledgements

We express our deepest gratitudes to Meital Levy for her great efforts in implementing the different methods and running the experiments.

References

[1] Circuits of Basic Functions Suitable For MPC and FHE, `http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC`.

[2] G. Asharov, Y. Lindell, T. Schneier and M. Zohner. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In the 20*th ACM Conference on Computer and Communications Security* (ACM CCS), pages 535–548, 2013.

[3] M. Bellare, V.T. Hoang and P. Rogaway. Foundations of garbled circuits. In the 19*th ACM Conference on Computer and Communications Security* (ACM CCS0, pages 784–796, 2012.

[4] J. Black. The Ideal-Cipher Model, Revisited: An Uninstantiable Blockcipher-Based Hash Function. In *FSE 2006*, Springer (LNCS 4047), pages 328–340, 2006.

[5] M. Bellare, V.T. Hoang, S. Keelveedhi and P. Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In the *IEEE Symposium on Security and Privacy 2013*, pages 478–492, 2013.

[6] A. Biryukov, D. Khovratovich and I. Nikolic. Distinguisher and Related-Key Attack on the Full AES-256. In *CRYPTO 2009*, Springer (LNCS 5677), pages 231–249, 2009.

[7] S.G. Choi, J. Katz, R. Kumaresan and H. Zhou. On the Security of the "Free-XOR" Technique. In the 9*th TCC*, Springer (LNCS 7194), pages 39–53, 2012.

[8] S. Gueron. Intel Advanced Encryption Standard (AES) Instructions Set, Rev 3.01. (2012) `https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set`

[9] S. Gueron. Intel's New AES Instructions for Enhanced Performance and Security. In the 16*th FSE* (FSE 2009), Springer (LNCS 5665), pages 51–66, 2009.

[10] S. Gueron. Optimized implementation of AES 128/192/256 key expansion. Software patch in `https://bugzilla.mozilla.org/show_bug.cgi?id=1122903` (2015).

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

[12] Y. Ishai, J. Kilian, K. Nissim and E. Petrank. Extending Oblivious Transfer Efficiently. In *CRYPTO 2003*, Springer (LNCS 2729), pages 145–161, 2003.

[13] V. Kolesnikov, P. Mohassel and M. Rosulek. FleXOR: Flexible Garbling for XOR Gates That Beats Free-XOR. In *CRYPTO 2014*, Springer (LNCS 8617), pages 440–457, 2014.

[14] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In the 35*th ICALP*, Springer (LNCS 5126), pages 486–498, 2008.

[15] V. Kolesnikov and T. Schneider. Secure Function Evaluation Techniques For Circuits Containing XOR Gates With Applications To Universal Circuits. Patent No. US 8,443,205 B2, 2013.

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

[17] Y. Lindell and B. Pinkas. A Proof of Yao's Protocol for Secure Two-Party Computation. In the *Journal of Cryptology*, 22(2):161–188, 2009.

[18] Y. Lindell, B. Pinkas and N. Smart. Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries. In the *6th Conference on Security and Cryptography for Networks,* Springer (LNCS 5229), pages 2–20, 2008.

[19] M. Naor, B. Pinkas and R. Sumner. Privacy Preserving Auctions and Mechanism Design. In the *ACM Conference on Electronic Commerce*, pages 129–139, 1999.

[20] B. Pinkas, T. Schneider, N.P. Smart and S.C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT 2009*, Springer (LNCS 5912), pages 250–267, 2009.

[21] A. Yao. How to Generate and Exchange Secrets. In the 27*th FOCS*, pages 162–167, 1986.

[22] S. Zahur, M. Rosulek and D. Evans. Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT 2015*, Springer (LNCS 9057), pages220–250, 2015. pages 220–250, 2015.