Faster LEGO-based Secure Computation without Homomorphic Commitments

Ruiyu Zhu and Yan Huang

Indiana University

Abstract. LEGO-style cut-and-choose is known for its asymptotic efficiency in realizing actively-secure computations. The dominant cost of LEGO protocols is due to *wire-soldering* — the key technique enabling to put independently generated garbled gates together in a bucket to realize a logical gate. Existing wire-soldering constructions rely on homomorphic commitments and their security requires the majority of the garbled gates in every bucket to be correct.
In this paper, we propose an efficient construction of LEGO protocols that does not use homomorphic commitments but is able to guarantee security as long as at least one of the garbled gate in each bucket is correct. Additionally, the faulty gate detection rate in our protocol doubles that of the state-of-the-art LEGO constructions. We have implemented our protocol and our experiments on several benchmark applications show that the performance of our approach is highly competitive in comparison with existing implementations.

1 Introduction

Since 1980s, significant effort has been devoted to making secure computation protocols practical. This include novel garbling schemes [8,7,54,19], software tools [38,20,23,21,30,36,37,49], and their applications [25,52,22,13,45,40,27,51]. While these works are restricted in the *passive* (honest-but-curious) threat model, which is fairly weak to model real-world adversaries, security against active adversaries is often more desirable.

The most practical approach for building actively-secure two-party computation protocols by far is the *cut-and-choose* paradigm. With cut-and-choose, roughly speaking, one party generates κ garbled circuits where κ depends on the statistical security parameter s; some fraction of those are "checked" by the other party—who aborts if any misbehavior is detected—and the remaining fraction are evaluated with the results being used to derive the final output. A rigorous analysis of the cut-and-choose paradigm was first given by Lindell and Pinkas [33], which required setting κ to roughly $3s$, and was later optimized to $\kappa = s$ [32,9,4] since it suffices to have only one honest circuit used for evaluation (hence we call them *SingleCut* protocols). More recently, it was shown that κ can be further reduced to $O(s/\log N)$[1] per execution over a batch of N executions of

[1] We will show in Section 6.4 that this should really be $2 + O(s/\log N)$.

the same function f between the same two parties (here we call them *BatchedCut* protocols) [34,24,35]. However, despite a prohibitive startup overhead, this technique is not applicable to many scenarios where secure computation is needed on an *ad hoc* basis, e.g., when either the function f or the peer is advised to change every so often due to the inevitable leakage through the results.

To attain similar efficiency in the single-execution setting, Nielsen and Orlandi proposed LEGO [42], which exploited the circuit evaluator's randomness to group individual NAND gates (as opposed to *circuits* in the batched-execution setting) to thwart active attacks. Later, the idea evolved to MiniLEGO [14], which is compatible with the more efficient AND-based garbling and the free-XOR technique. Their recent independent work [44] provides an implementation of their protocol [15], demonstrating the practical efficiency of LEGO approach. They use homomorphic commitments and the security of their protocol depends on the *majority* of the gadgets in every bucket being correct. In contrast, our work shows a different construction of LEGO protocols with highly competitive performance.

1.1 Contribution

Contributions of this work include:

New Techniques. We propose two new optimizations for constructing efficient LEGO protocols:

1. The main bottleneck of LEGO protocols is *wire-soldering*, which converts, in a privacy-preserving way, a wire-label of a logical-gate bucket to a wire-label on an garbled gate to enable combining multiple independently garbled gates to realize a logical gate. To achieve high performance wire-soldering, we introduce a new cryptographic primitive called *XOR-homomorphic interactive hash* (IHash) to replace the XOR-homomorphic commitments used in prior works. We propose a simple construction of IHash by integrating Reed-Solomon codes, pseudorandom generators (PRG), and a single invocation of a w-out-of-n oblivious transfer protocol (Section 4.2). We proved the security of our interactive hash construction (Section 4.3). IHash can be a primitive of independent interest, e.g., it may also be used to efficiently solder circuits in other BatchedCut protocols.
2. Using IHash, we are able to improve existing LEGO-based cut-and-choose mechanism in two more aspects:
   (a) Our protocols guarantee security assuming a *single* correctly garbled gate exists in every bucket. In contrast, existing LEGO-based protocols [42,14,15,44] require *majority correctness* in every bucket. This enhancement allows us to roughly reduce the number of gadgets in every bucket by 1/2 when offering 40-bit statistical security.
   (b) We are able to double the faulty gate detection rate in our LEGO protocol. That is, we allow detecting faulty gates with probability 1/2, as opposed to 1/4 in existing protocols [14,15,44].

Implementation and Evaluation. We have implemented our protocol and experimentally evaluated its performance with several representative computations. In particular, our protocol exhibits very attractive performance in handling the target function's input and output wires: 0.57 μs per garbler's input-wire and 8.24 μs per evaluator's input-wire, and 0.02 μs per output-wire, which are roughly 24x, 2.4x, and 600x faster than WMK [50]'s highly optimized designs (Fig. 8). Without exploiting parallelism, our protocol is able to execute 105.3M logical XOR gates per second and (when bucket size is 5) 45.5K logical AND per second on commodity hardware (two Amazon EC2 `c4.2xlarge` instances over LAN). We will open-source our implementation to benefit future exploration in this field.

2 Technical Overview

Notations. We assume P_1 and P_2, holding x and y respectively, want to securely compute a function $f(x, y)$. We use the standard definition of actively-secure two-party computation [18]. Throughout this paper, we assume P_1 is the circuit generator (who is also the IHash sender) and P_2 is the circuit evaluator (who is also the IHash receiver). For simplicity, we assume that only P_2 will receive the final result $f(x, y)$. We assume f can be represented as a circuit C containing N AND gates while the rest are all XORs.

We summarize the list of variables in Fig. 1.

s	The statistical security parameter.	ℓ_w	The symbol-length of wire-labels
k	The computational security parameter.	n_w	The symbol-length of wire-label encoding.
N	The number of logical AND gates (i.e., buckets) in the circuit.	w_w	The number of watched symbols in a wire-label encoding.
T	Total number of garbled AND gates generated by P_1.	σ_w	The bit-length of symbols used in wire-label encoding.
B	Bucket size, i.e., the number of garbled AND gates in a bucket.	ℓ_p	The symbol-length of permutation messages.
Δ	The global secret delta between a 0-label and its corresponding 1-label.	n_p	The symbol-length of wire-label encoding.
$\langle m \rangle$	The i-hash of a message m.	w_p	The number of watched symbols in a wire-label encoding.
ρ	The permutation message whose parity bit, p, is the permutation bit. Each wire has a freshly sampled ρ.	σ_p	The bit-length of symbols used in permutation message encoding.

Fig. 1: Variables and their meanings.

All vectors in this paper are by default column vectors.

2.1 LEGO Protocols

LEGO protocols belong to the BatchedCut category of cut-and-choose-based secure computation protocols [55]. For a Boolean circuit C of N logical gates, the high-level steps of a LEGO protocol to compute C are,

1. **Generate.** P_1 generates a total of T garbled gates.
2. **Evaluate.** P_2 randomly picks $B \cdot N$ gates and groups them into N buckets. Each bucket will realize a gate in C. P_2 evaluates every bucket by first translating wire-labels on the bucket's input-wires to wire-labels on individual garbled gate's input-wires, evaluating every garbled gates in the bucket, and then translating the obtained wire-labels on the garbled gates' output-wires back to a wire-label on the bucket's output-wire. (The wire-label translation, also called *wire-soldering*, is explained in more detail below.)
3. **Check.** P_2 checks each of the rest $T - BN$ garbled gates for correctness. If any of these gates was found faulty, P_2 aborts. Though, due to the randomized nature of the checks, P_2 will not always be able to detect it when checking a faulty gate.
4. **Output.** P_1 reveals the secret mapping on the circuit's final output-wires so that P_2 is able to map the final output-wire labels into their logical bit values.

The first construction [42] was based on NANDs and require public key operations for wire-soldering. Fredericksen et al. [14] later proposed a LEGO scheme that is compatible with the notable free-XOR optimization [29] using XOR-Homomorphic commitments as a black box. Under this paradigm, it suffices to assume all the garbled gates are ANDs, since all XORs can be securely computed locally and no extra treatment is needed to ensure correct behavior on processing XORs. However, due to the use of the global secret Δ for free-XOR, a garbled AND can't be fully opened for check purpose. Instead, a random one of the four possible pairs of inputs to a binary gate is picked to check correctness.

Wire-soldering. As depicted in Fig. 5, each bucket realizes a logical gate, thus has input and output wires like the logical gate it realizes. In order to evaluate an independently generated garbled gate assigned to a bucket, an input-wire of the bucket (with wire-labels w^0_{bucket} and $w^1_{\mathsf{bucket}} = w^0_{\mathsf{bucket}} \oplus \Delta$ denoting 0 and 1) needs to be connected to the corresponding input-wire (with wire-labels w^0_{gate} and $w^1_{\mathsf{gate}} = w^0_{\mathsf{gate}} \oplus \Delta$) of the garbled gate to evaluate. This is done by requiring P_1 to send $d = w^0_{\mathsf{bucket}} \oplus w^0_{\mathsf{gate}}$ and P_2 to xor d with the wire-label on the bucket (either w^0_{bucket} or w^1_{bucket}) he obtained from evaluating the previous bucket. To prevent a malicious P_1 to send a forged d, existing protocols used XOR-Homomorphic commitments to let P_1 commit Δ, w^0_{bucket}, and w^0_{gate} (which allows P_2 to derive the commitment of d homomorphically), so that P_2 can verify the validity of d from its decommitment without learning any extra information about $w^0_{\mathsf{bucket}}, w^0_{\mathsf{gate}}$.

2.2 Our Optimizations

Below we sketch the intuition behind our optimization ideas of LEGO protocols.

XOR-homomorphic Interactive Hash. XOR-homomorphic Interactive Hash (IHash) is a cryptographic protocol involving two participants, which we call the *sender* and the *receiver*, respectively. The design of IHash is directly motivated by the security goals of wire-soldering:

1. **Binding.** Every i-hash of a secret message uniquely identifies the message with all but a negligible probability, so that the message holder cannot modify a secret message once its i-hash is sent.
2. **Hiding.** The i-hash receiver does not learn any extra information about the secret message other than the i-hash itself. For a uniform-randomly sampled message, it is guaranteed that certain entropy remains after its i-hash is sent because by definition an i-hash needs to be shorter than the original message.
3. **XOR-Homomorphism.** Given the i-hashes of two messages m_1 and m_2, the receiver can locally compute the i-hash of $m_1 \oplus m_2$. This enables the receiver (circuit evaluator) to solder wire-labels from independently garbled gates using a verifiable label-difference supplied by the circuit generator.

Unlike a commitment scheme which requires the committer's cooperation to match messages with commitments, IHash allows the receiver *alone* to verify if any message matches with an i-hash (like with traditional hashes). In addition, a commitment hides every bit of its message whereas i-hashes allow leaking arbitrary information about its message through the i-hash itself, up to the length of the i-hash. Nevertheless, we find that this somewhat weaker primitive suffices to soldering wires in LEGO protocols.

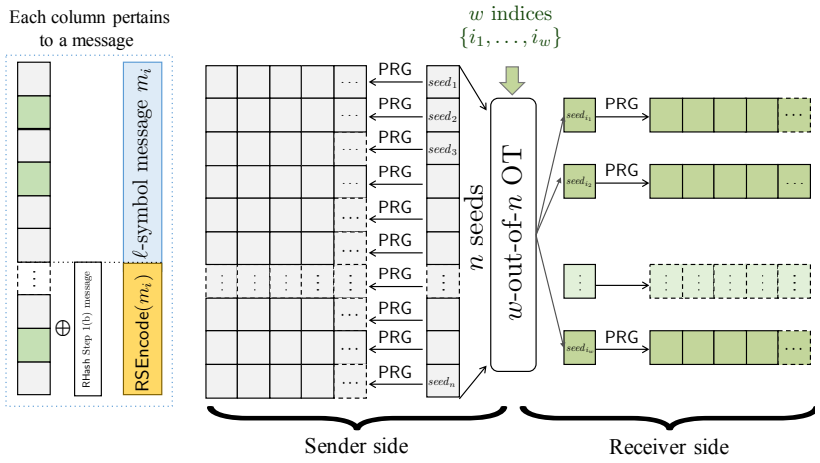

Fig. 2: Interactive Hash based on OT and Reed-Solomon Code

Figure 2 illustrates our construction of the XOR-homomorphic IHash scheme. The high-level idea is to let the IHash sender encode his/her secret message m using a $[n, \ell, n-\ell+1]_{2^\sigma}$ Reed-Solomon code and let the receiver secretly watch (soon we will detail how to watch secretly) w of the n symbols in $\mathsf{Encode}(m)$.

Recall that a $[n, \ell, n-\ell+1]_{2^\sigma}$-code is one that takes in an ℓ-symbol message and outputs an n-symbol encoding (where symbols are of σ-bit) so that the minimal distance between the codewords is $n-\ell+1$ symbols. The w watched symbols are the i-hash of m. The receiver can verify a particular message matches with its i-hash by encoding the message and making sure all values at the w watched positions on the encoding coincide with the i-hash it holds. As a result, with respect to binding, if the sender forges a message m', at least $n-\ell+1$ symbols in the codewords have to be different, hence, the i-hash receiver can detect the forgery with all but $\binom{\ell-1}{w} / \binom{n}{w}$ probability. With respect to hiding, although the i-hash reveals $w\sigma$ bits entropy in m to the receiver, if the original message has $\ell\sigma$ bits entropy, then the rest $(\ell-w)\sigma$ bits entropy remains perfectly hidden to the receiver. Therefore, to guarantee hiding, we can set ℓ sufficiently large based on the security parameter. Finally, the additive (i.e., XOR) homomorphic property of i-hashes is inherent in the linearity of Reed-Solomon codes.

The "secret watch" above can be realized by a w-out-of-n oblivious transfer protocol. Moreover, we only need to invoke this oblivious transfer *once*. The key idea is to let the sender pick n random seeds and obliviously transfer w seeds of the receiver's choice. Later, the sender sends correction messages $\mathsf{Encode}(m)_i \oplus \mathsf{PRG}(seed_i)$ to the receiver where $\mathsf{Encode}(m)_i$ is the i^{th} symbol of m's encoding and $i \in \{1, \ldots, n\}$. Thus, learning $seed_i$ allows the receiver to see the corresponding symbols in m's codeword. We also notice that the input-wire labels of all garbled gates are uniformly random. Therefore, setting the i^{th} symbol of the j^{th} wire-label to be $\mathsf{PRG}(seed_i, j)$ where $i \in \{1, \ldots, \ell\}$ while using a systematic code will reduce the work to only send the corrections on the last $n-\ell$ symbols, i.e., $m_i \oplus \mathsf{PRG}(seed_i, j)$ for $i \in \{\ell+1, \ldots, n\}$.

Fast Wire-Soldering. Wire-soldering is one of the most challenging efficiency barriers in LEGO protocols. Recall that P_1 garbles all the AND gates independently. Thus, in the circuit evaluation phase where B random AND gates are grouped into a bucket to evaluate a logical AND gate, P_2 needs to "translate" a left-input wire-label of a bucket to a left-input wire-label of a gate for each garbled gates in the bucket. To this end, we require, at the garbling stage, that, for every garbled gate, P_1 i-hash one wire-label on every wire of the garbled gate to P_2; and, at the gate evaluation stage, that P_1 send the `xor`-differences between every pair of the source and target wire-labels. The validity of the `xor`-differences can be verified against their i-hashes. Note that even if an i-hash leaks entropy, we can increase the length of the wire-labels to ensure enough entropy remain in the labels to guarantee the needed computational security.

Moreover, for benefits that will be clear soon, we require P_1 also to i-hash the global Δ (required by the free-XOR technique) to P_2. Recall that all the wire-labels at the bucket level needs also to be i-hashed to P_2. To prevent P_2 from learning logical values of the intermediate wire-labels, P_1 will i-hash either the 0-label or the 1-label of each wire with equal probability. Without extra treatment, however, this will allow a malicious P_1 to surreptitiously flip a wire-label's logical value. We fix this issue by adding a random permutation message ρ to each wire and use ρ's parity bit to bind the plaintext bit the i-hashed wire-

label represents. For integrity of ρ, we require P_1 to i-hash ρ to P_2 so that P_2 can verify the ρ values of each wire at the garbled gate checking stage. We stress that the value of ρ for all intermediate wires will never be revealed.

To achieve fast wire-soldering for practically efficient LEGO protocols, we found the following two optimizations indispensable.

1. The Reed-Solomon encoding process can be viewed as multiplying the public encoding matrix $A_{n\times\ell} = [a_{i,j}]_{1\le i\le n, 1\le j\le \ell}$ with the message vector $\boldsymbol{m}_{\ell\times 1} = [m_1, \ldots, m_\ell]$ where m_i's are σ-bit symbols. To ensure security of LEGO protocols, the (n, ℓ, σ) values would be $(n_w, \ell_w, \sigma_w) = (86, 32, 8)$ for wire-labels and $(n_p, \ell_p, \sigma_p) = (44, 20, 6)$ for permutation messages. A naïve implementation of the encoding process will require more than 2700 Galois Field (GF) multiplications per wire-label and 900 GF multiplications per permutation message, which amounts to more than 10K multiplications per garbled gate. Even if field multiplications are realized as table-lookups, 10K memory accesses per gate is already 40× slower than AESNI-based garbling itself, making LEGO approach noncompetitive in practice.

   Our key idea to speedup encoding process is to pack many symbols into operands (e.g., `__m128, __m256, __m512`) of vector instructions and leverage Intel Intrinsic instructions [2] to enable efficient message encoding. Below we illustrate the idea with an example where $n = 96, \ell = 32, \sigma = 8$), i.e., encoding a 32-symbol message $\boldsymbol{m}$ into a 96-symbol codeword where symbols are of 8-bit. First, we use a systematic code so that it suffices to compute the last $n - \ell = 64$ symbols of the codeword since the first $\ell = 32$ symbols are identical to the original message. Thus, we can restrict our attention to the last 64 rows of the encoding matrix A, call it A'. Let $\boldsymbol{a}_{\cdot,1} = [a_{33,1}, \ldots, a_{96,1}]^T$ be the first column of the matrix A' and we can store the 64 symbols of $\boldsymbol{a}_{\cdot,1}$ in a single `__m512` register. Let m_i be the i^{th} symbol of $\boldsymbol{m}$. The last 64 symbols in the encoding of $\boldsymbol{m}$ can be computed as $\boldsymbol{c} = \sum_{i=1}^{\ell} m_i \boldsymbol{a}_{\cdot,i} = \sum_{i=1}^{\ell} [m_i a_{33,i}, \ldots, m_i a_{96,i}]^T$. Since $m_i \in \mathrm{GF}(2^8)$ and all column vectors $\boldsymbol{a}_{\cdot,i}$ are publicly fixed, each $m_i \boldsymbol{a}_{\cdot,i}$ can thus be efficiently derived with a single lookup into a table of 256 entries of `__m512` values and the sum can be computed with $\ell - 1 = 31$ `__mm512_xor` instructions. This optimization would reduce $32 \times 96 = 3072$ field multiplications per encoding down to just 32 memory reads, about a hundredth cost of the implementation based on naïve table-lookups.
2. We observe that, although the hiding of i-hashes for wire-labels is *computational* (since an adversary could use the garbled truth table to search the "right" label offline), the hiding on the permutation message ρ is *perfect* because no additional constraints are provided to allow offline search for the permutation bit (i.e., the parity of ρ). Thus, it suffices to require only 1-bit of entropy remain in ρ after i-hashing each ρ for perfectly hiding which wire-labels on a wire was i-hashed. This observation allows us to select much more efficient parameters to i-hash ρ, i.e., $(n_p, \ell_p, \sigma_p) = (44, 20, 6)$ as opposed to $(86, 32, 8)$ for i-hashing wire-labels.

Doubling the Faulty Gate Rate. In state-of-the-art LEGO protocols [14,15,44], a faulty gate selected to be checked will be found faulty with probability 1/4. This is because the garbled gates are produced with respect to the global secret Δ (which is the `xor`-difference between the 0-label and 1-label on a wire) required by the free-XOR technique [29], hence only one out of the four garbled rows of a binary gate can be opened for checking. In contrast, our protocol allows a faulty gate to be detected with probability 1/2. We achieve this by integrating the Half-Gate garbling technique [54] which requires only two garbled rows per gate, so that each check allows to verify 50% of the garbled table without revealing Δ. We formally prove this result as Lemma 5.3.

Dealing with Faulty Gates Used for Evaluation. Our protocol is able to guarantee security as long as a single correctly garbled gate exists in every bucket. This improvement is due to a combination of the IHash and the free-XOR techniques. Denote the i-hash of a message $m \in \{0,1\}^*$ by $\langle m \rangle$. We let the circuit evaluator to learn $\langle \Delta \rangle$ where Δ is the global secret. On each wire, the evaluator also learns an i-hash $\langle w \rangle$ where w defines either the 0-label or the 1-label on that wire (but the evaluator doesn't know which). Recall that the evaluator can locally verify the validity of a wire-label using the IHash's Verify algorithm. Therefore, if at least one gate in a bucket is good, evaluating all the garbled gates in the bucket will give one or more valid output wire-labels. When translating these wire-labels to the bucket output wire-label, one of the following two cases has to happen:

1. They all match with the same i-hash, either $\langle w \rangle$ or $\langle w \rangle \oplus \langle \Delta \rangle$. Since all *valid* wire-labels are consistent on the plaintext bit they represent and one of them is known to be correct, the evaluator can directly proceed with this valid wire-label to evaluate the subsequent buckets.
2. Some of them translate to $\langle w \rangle$ whereas others translate to $\langle w \rangle \oplus \langle \Delta \rangle$. In this case, the evaluator can simply `xor` the two valid labels to recover Δ. Once Δ is known, the evaluator can use it to recover the circuit generator's private input x and locally computes and outputs $f(x, y)$.

Hence, in either case our protocol can be proved secure.

Entropy Extraction for Efficient Garbling. With fixed-key AESNI instructions, the state-of-the-art garbling technique is able to produce 20 million garbled rows per second, which is about 10× faster than a SHA256-based garbling scheme [39,54,7]. However, the wire-labels in our protocol need to be longer than 128 bits to ensure enough entropy (e.g., more than 80 bits) remains even part of a wire-label is leaked to the evaluator through its i-hash. Although it is straightforward to use SHA256 to implement garbling to accommodate longer wire-labels in the random oracle model, *a priori*, it is unclear how this can be efficiently realized only assuming fixed-key AES is an ideal cipher.

Our intuition is to compress a longer wire-label down to a 128-bit label while preserving as much entropy as possible, and then run existing fixed-key AES-based garbling with the compressed labels. As a concrete example, wire-labels in our protocol are 256-bit (i.e. 32 8-bit symbols). During i-hashing, a 32-symbol

GenAND$(i,\ \Delta,\ w_l^0,\ w_r^0)$

Require: $i \in \mathbb{N}.\ \Delta, w_l^0, w_r^0 \in \{0,1\}^{256}$.

$q_l \coloneqq \mathsf{lsb}\,(w_l^0)$
$q_r \coloneqq \mathsf{lsb}\,(w_r^0)$
$j \coloneqq 2i$
$j' \coloneqq 2i+1$
$T_G \coloneqq H(w_l^0, j) \oplus H(w_l^0 \oplus \Delta, j) \oplus q_r \Delta$
$w_g^0 \coloneqq H(w_l^0, j) \oplus q_l T_G$
$T_E \coloneqq H(w_r^0, j') \oplus H(w_r^0 \oplus \Delta, j') \oplus w_l^0$
$w_e^0 \coloneqq H(w_r^0, j') \oplus q_r(T_E \oplus w_l^0)$
$w_o^0 \coloneqq w_g^0 \oplus w_e^0$
return (w_o^0, T_G, T_E)

EvlAND$(i,\ w_l,\ w_r,\ T_G,\ T_E)$

Require: $i \in \mathbb{N}.\ w_l, w_r \in \{0,1\}^{256}$.

$s_l \coloneqq \mathsf{lsb}\,(w_l)$; $s_r \coloneqq \mathsf{lsb}\,(w_r)$
$j \coloneqq 2i$; $j' \coloneqq 2i+1$
$w_g \coloneqq H(w_l, j) \oplus s_l T_G$
$w_e \coloneqq H(w_r, j') \oplus s_r(T_E \oplus w_l)$
$w_o \coloneqq w_g \oplus w_e$
return w_o

$H(m,\ j)$

$w \coloneqq \mathsf{Compress}(m)$
$k \coloneqq 2j$; $k' \coloneqq 2j+1$
$w \coloneqq m \oplus k$; $w' \coloneqq m \oplus k'$
return $\mathrm{AES}(w) \oplus w \parallel \mathrm{AES}(w') \oplus w'$

Fig. 3: Garbling with 256-bit wire-labels. AES$(\cdot)$ denotes calling AES with a fixed, publicly-known key. $\mathsf{Compress}(m)$ essentially computes $A'm$ where A' is a rank-16, 16×32 matrix over $\mathrm{GF}(2^8)$. A' is randomly picked by the circuit generator after the evaluator chose its watch symbols.

wire-label will be encoded into a 86-symbol codeword; and the evaluator randomly picks 21 of the 86 symbols in the codewords to watch. Since the "watch" reveals $8 \times 21 = 168$ bits entropy, 88 bits of entropy remains in each wire-label. To compress a 256-bit wire-label m to a 128-bit m' while carrying over the entropy, our strategy is to randomly sample a 16×32, rank-16 matrix $A' = [a'_{i,j}]$ where $a'_{i,j} \in \mathrm{GF}(2^8)$ and compute $m' = A'm$ (in other words, A' represents a set of 16 linearly independent row-vectors in the vector space $\mathrm{GF}(2^8)^{32}$). Note that A' is sampled only *after* the evaluator has chosen its watched symbols. Intuitively, this compression preserves entropy because the chances are extremely low that any one of the 16 row-vectors happen to be a linear combination of a set of 21 row-vectors of $\mathrm{GF}(2^8)^{32}$ that are picked randomly and independently by the evaluator. We present a formal analysis of the entropy loss in Section 6.2.

Once the Compress algorithm (i.e., essentially a matrix multiplication as described above) is clear, we can formalize our garbling scheme based on that of Half-Gates [54]. The main difference lies in the function H. Our H, specified in Fig. 3, maps $\{0,1\}^{256} \times \mathbb{N}$ to $\{0,1\}^{256}$ and involves two calls to a fixed-key AES cipher to produce 256 bits of pseudorandom mask to encrypt the longer output wire-label.

2.3 Related Work

TinyLEGO. As an independent and concurrent work, TinyLEGO [15,43,44] explored ways to improve LEGO protocols in the single-execution setting. Our approach is different from TinyLEGO in several aspects:

1. To solder the wires, TinyLEGO used additive homomorphic commitment, whereas we propose the notion of IHash and give a highly efficient construction of IHash using PRG, Reed-Solomon code and Intel Intrinsics [2]. We show that, despite of IHash being more leaky than homomorphic commitments, it suffices the purpose of constructing efficient LEGO protocols. Our construction of IHash shares some similarity with that of XOR-Homomorphic commitments in [16]. However, the two schemes differ in the way OTs are used (1-out-of-2 OT versus w-out-of-n OT), the selection of error correcting codes, and the rationals behind picking the parameters.
2. TinyLEGO involves cut-and-choose two types of garbled gadgets (i.e., ANDs and wire-authenticators) and requires correct *majority* in the total number of garbled gadgets in each bucket, whereas our protocol only uses garbled ANDs and the security holds as long as a single correctly garbled AND exists in each bucket. In addition, the faulty gate detection rate in our protocol is twice of that in TinyLEGO.
3. Because of our optimizations, our protocol can run more than 2x faster in a LAN and being highly competitive over a WAN. However, their protocols are about 20–50 more efficient in bandwidth thus would be advantageous in some bandwidth-stringent network environments. See Section 7 for detailed performance comparisons.

NNOB [41] and SPDZ [12]. Both NNOB and SPDZ require linear number of rounds and expensive pre-processing, our protocol has only a small constant rounds and lightweight setup. Thus, ours performs better when the network latency cannot be ignored, or when resources are limited in the preparation phase.

Lindell-Riva [35] and Rindal-Rosulek [48]. In the offline/online setting, Lindell-Riva and Rindal-Rosulek provided very efficient prototypes of secure computation protocols. Although the high-level idea of cut-and-choose resembles that of LEGO, their results are not applicable if only one (or just very few) executions is needed. For example, with [35], one AES can be computed in about 74 ms amortized time assuming a 75,000 ms offline delay is acceptable for preparing 1024 executions. Since wire-soldering is much less of an issue, their technique would be far less efficient when carried out to cut-and-choose individual-gates.

Wang-Malozemoff-Katz [50]. Wang et al. recently designed and implemented by far the most efficient SingleCut secure computation protocol. Thanks to a (mostly) symmetric-key cryptography based garbler's input consistency enforcement mechanism and their careful use of SSE instructions for preventing selective failure attacks, a single AES instance can be computed in 65 ms (with

input/output wires processed at roughly 20 μs per wire). However, even compared with their optimized protocol, processing input/output wires in our protocol can still be much faster, hence will be competitive in computing shallow circuits with many input/output wires (see Section 7 for detailed performance comparisons). Moreover, due to the unique advantages of LEGO-based cut-and-choose in supporting actively secure RAM-based secure computations as was pointed out by [3], we believe it will be worthwhile and promising to plug faster IHash constructions into our framework to derive more efficient LEGO protocols in the future.

Parallelism. Many existing works [44,31,35] on secure computations have exploited parallelism for bigger performance improvements. However, noting the *embarrassingly parallelizable* nature of the protocols in this domain (including ours), we follow the convention of many other works [50,54] and restrict our attention to the single-threaded model and treat computation as an energy-consuming scarce resource.

3 Preliminaries

3.1 Oblivious Transfer

We use 1-out-of-2 oblivious transfers to send wire labels corresponding to the evaluator's input, and two k-out-of-n oblivious transfers for wire soldering. A k-out-of-n oblivious transfer protocol takes n messages $m_1, \ldots, m_n$ from the sender and a set of k indices $i_1, \ldots, i_k \in [1, n]$ from the receiver, and outputs nothing but the k messages $m_{i_1}, \ldots, m_{i_k}$ to the receiver. Composably secure 1-out-of-2 OT can be efficiently instantiated from dual-mode cryptosystems [46] and efficiently extended [28,5] from inexpensive symmetric operations plus a constant number of base OTs. Camenisch, Neven, and Shelat [10] proposed an efficient and simulatable k-out-of-n OT in the Random Oracle Model.

3.2 Garbled Circuits

First proposed by Yao [53], garbled circuits were later formalized as a cryptographic primitive of its own interest [8]. Bellare et al. have carved out three security notions for garbling: *privacy*, *obliviousness*, and *authenticity*. We refer readers to their paper for the formal definitions. In the past few years, many optimizations have been proposed to improve various aspects of garbled circuits, such as bandwidth [47,54], evaluator's computation [47], memory consumption [23], and using dedicated hardware [8]. Our protocol leverages Half-Gates garbling recently proposed by Zahur et al. [54] which offers the simulation-based definition of privacy, obliviousness, and authenticity under a *circular correlation robustness* assumption of the hash function H. We summarize their garbling algorithms GenAND and EvlAND in Fig. 3.

More formally, a *garbling scheme* $\mathcal{G}$ is a 5-tuple $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De}, f)$ of algorithms, where Gb is an efficient randomized *garbler* that, on input $(1^k, f)$, outputs (F, e, d); En is an *encoder* that, on input (e, x), outputs X; Ev is an *evaluator*

that, on input (F, X), outputs Y; De is a *decoder* that, on input (d, Y), outputs y. The *correctness* of $\mathcal{G}$ requires that for every $(F, e, d) \leftarrow \mathsf{Gb}(1^k, f)$ and every x,

$$\mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x))) = f(x).$$

Let Φ be a prefixed function modeling the acceptable information leak and "$\approx$" symbolizes *computational indistinguishability*. *Privacy* of $\mathcal{G}$ implies that there exists an efficient simulator $\mathcal{S}$ such that for every x,

$$\left\{(F, X, d) : \begin{matrix}(F, e, d) \leftarrow \mathsf{Gb}(1^k, f), \\ X \leftarrow \mathsf{En}(e, x).\end{matrix}\right\} \approx \{\mathcal{S}(1^k, f, \Phi(f))\}.$$

Obliviousness of $\mathcal{G}$ implies that there exists an efficient simulator $\mathcal{S}$ such that for every x,

$$\{(F, e, d) \leftarrow \mathsf{Gb}(1^k, f), X \leftarrow \mathsf{En}(e, x) : (F, X)\} \approx \{\mathcal{S}(1^k, f)\}.$$

4 Homomorphic Interactive Hash

In this section, we describe XOR-homomorphic interactive hash, a new primitive that enables multiple enhancements in our LEGO protocol.

4.1 Definition

It involves two parties, known as the *sender* (P_1) and the *receiver* (P_2), to *compute* an interactive hash (i-hash) while the receiver can locally *verify* a message against an i-hash that it holds. We formally define its ideal functionality $\mathcal{F}_{\text{XorIHash}}$ in Fig. 4. Hash is an efficient two-party probabilistic algorithm that takes a message m from P_1 and outputs an i-hash of m (denoted as $\langle m \rangle$) to P_2 without revealing any additional information to either party. Verify is an efficient algorithm (locally computable by P_2) that takes an i-hash $\langle m \rangle$ and a message m' and outputs a bit b indicating whether $m = m'$. Like conventional hashes, we require $|\langle m \rangle| < |m|$ and that for any two distinct messages m_1 and m_2, $\langle m_1 \rangle \neq \langle m_2 \rangle$ except for a negligible probability. Finally, we require the hashes to be XOR-homomorphic, i.e., $\langle m_1 \rangle \oplus \langle m_2 \rangle = \langle m_1 \oplus m_2 \rangle$.

Interactive hash offers certain "*hiding*" and "*binding*" properties. That is, with all but negligible probability,

- **Hiding.** The receiver of $\langle m \rangle$ learns nothing about m except for what can be efficiently computed from $\langle m \rangle$;
- **Binding.** The sender of $\langle m \rangle$ can't claim a different message m' to be the preimage of $\langle m \rangle$.

However, unlike cryptographic commitments, with i-hash, (1) some entropy in m can be leaked to the receiver yet the rest remains; (2) the message owner can't compute the hash on its own; (3) the hash receiver can verify *on its own* whether a message matches with a hash.

– Hash. Upon receiving $(\texttt{Hash}, m_1, \ldots, m_\nu)$ ($\nu \geq 1$) from P_1: for every i, if there is a recorded value (cid, m_i), generate a delayed output $(\texttt{Receipt}, cid_i, \langle m_i \rangle)$ to P_2; otherwise, pick a fresh number cid_i, record (cid_i, m_i) and generate a delayed output $(\texttt{Receipt}, cid_i, \langle m_i \rangle)$ to P_2.
– Verify. Upon receiving $(\texttt{Verify}, cid_1, \ldots, cid_\nu, d)$ from P_2: if there are recorded values $(cid_1, m_1), \ldots, (cid_\nu, m_\nu)$ (otherwise do nothing), set $z = 1$ if $m_1 \oplus \cdots \oplus m_t = d$, and $z = 0$, otherwise; generate a delayed output $(\texttt{VerifyResult}, z)$ to R.

Fig. 4: Ideal functionality of XOR-Homomorphic Interactive Hash. (P_1 is the hash sender and P_2 the hash receiver. "Send a delayed output x to party P" reflects a standard treatment of fairness, i.e., "send (x, P) to the adversary; when receiving `ok` from the adversary, output x to P.")

4.2 Construction

Fig. 2 illustrates the high-level idea behind our construction. Let OT_n^w be an ideal functionality for a w-out-of-n oblivious transfer. $\mathsf{Encode}_{\ell,n,d}(\cdot)$ denotes the encoding algorithm of $[n, \ell, n-\ell+1]_{2^\sigma}$ Reed-Solomon *systematic* code, i.e., (over σ-bit symbols) ℓ-symbol messages are encoded into n-symbol *codeword*s with minimal distance of $n - \ell + 1$ symbols. Let PRG be a pseudorandom generator and s, k are the statistical and computational security parameters.

To allow the receiver to obliviously watch the set of w positions on every message's n-symbol codeword without invoking an OT instance per message, we let the sender generate n secret seeds and call *only once* a w-out-of-n OT to allow the receiver learn w of these seeds (Step 1 of IHash.Setup). These seeds are then used as keys to a PRG to create n *rows* of pseudorandom symbols, of which the receiver is able to recover w rows. When a message m is ready to be i-hashed, the sender simply encode m and sends the `xor`-difference between m's codeword and the next *column* of n pseudorandom symbols generated from the seeds (Step **??** of IHash.Hash) so that the receiver can record the symbols for which it watched the corresponding keys.

To obtain active-security, our protocol actually generates $n+\xi$ i-hashes when i-hashing n messages, then uses the extra ξ i-hashes to verify that the sender followed the protocol honestly (Step 2 of IHash.Hash). In our protocol, we set $\xi = \left\lceil \frac{1}{\sigma} \log \left(2^{-s} - \binom{\ell-1}{w} / \binom{n}{w} \right) \right\rceil$ to bound the failure probability of our simulator $\mathcal{S}$ in the security proof of Theorem 4.1 by 2^{-s}.

The detailed construction steps are as follows.

– $\mathsf{IHash}_{\ell,\sigma}.\mathsf{Setup}(\{seed_1, \ldots, seed_n\};\ \{i_1, \ldots, i_w\})$
Note $\{seed_1, \ldots, seed_n\}$ is P_1's secret input and $\{i_1, \ldots, i_w\}$ is P_2's secret inputs.

1. P_1 and P_2 run OT_n^w where P_1 is the sender with inputs $seed_1, \ldots, seed_n$, and P_2 is the receiver with inputs $i_1, \ldots, i_w$. At the end of this step, P_2 learns $seed_{i_1}, \ldots, seed_{i_w}$.

– $\mathsf{IHash}_{\ell,\sigma}.\mathsf{Hash}(\boldsymbol{m}_1, \ldots, \boldsymbol{m}_\nu)$

1. For $t = 1, \ldots, \nu + \xi$,
   (a) For $1 \le i \le \ell$, P_1 computes $x_i = \mathsf{PRG}(seed_i, t)$, where $x_i \in \{0,1\}^\sigma$, and sets $\boldsymbol{m'}_t \coloneqq x_1 \| \ldots \| x_\ell$.

   (b) For $\ell < i \le n$, P_1 sends to P_2

$$x'_i \coloneqq \mathsf{PRG}(seed_i, t) \oplus \mathsf{Encode}(\boldsymbol{m'}_t)[i]$$

   where $\mathsf{Encode}(\boldsymbol{m'}_t)[i]$ denotes the i^{th} symbol of $\boldsymbol{m'}_t$'s *systematic* codeword.

   (c) $\forall i \in \{i_1, \ldots, i_w\}$, P_2 computes

$$x_i \coloneqq \begin{cases} \mathsf{PRG}(seed_i, t), & \text{if } 1 \le i \le \ell \\ \mathsf{PRG}(seed_i, t) \oplus x'_i, & \text{if } \ell < i \le n \end{cases}.$$

   Then P_2 sets $\langle \boldsymbol{m'}_t \rangle = (x_{i_1}, \ldots, x_{i_w})$.

2. For $t = 1, \ldots, \xi$,
   (a) P_2 randomly picks $y \leftarrow \{0,1\}^{\sigma \cdot \nu}$ and sends it to P_1.

   (b) P_1 sends $\hat{\boldsymbol{m}'}_t \coloneqq \sum_{i=1}^{\nu} y_i \boldsymbol{m'}_i + \boldsymbol{m'}_{\nu+t}$ where $y_i \in \{0,1\}^\sigma$ to P_2.

   (c) P_2 runs $\mathsf{IHash}_{\ell,\sigma}.\mathsf{Verify}\left(\sum_{i=1}^{\nu} y_i \langle \boldsymbol{m'}_i \rangle + \langle \boldsymbol{m'}_{\nu+t} \rangle, \hat{\boldsymbol{m}'}_t\right)$ where $y_i \in \{0,1\}^\sigma$ and aborts if it fails.

3. For $i = 1, \ldots, \nu$, P_1 sends $\boldsymbol{x}_i \coloneqq \boldsymbol{m}_i \oplus \boldsymbol{m'}_i$ to P_2, who then computes

$$\langle \boldsymbol{m}_t \rangle \coloneqq \bigparallel_{i=i_1}^{i_w} (\langle \boldsymbol{m'}_t \rangle[i] \oplus \mathsf{Encode}(\boldsymbol{x}_t)[i])$$

   where $\|_{i=1}^n a_i$ means $a_1 \| \cdots \| a_n$ and $\mathsf{Encode}(\boldsymbol{x}_t)[i]$ denotes the i^{th} symbol of $\boldsymbol{x}_t$'s codeword. P_1 outputs nothing and P_2 outputs $\langle \boldsymbol{m}_1 \rangle, \ldots, \langle \boldsymbol{m}_\nu \rangle$.

– $\mathsf{IHash}_{\ell,\sigma}.\mathsf{Verify}\left(\langle \boldsymbol{m}_1 \rangle, \ldots, \langle \boldsymbol{m}_t \rangle, \boldsymbol{m'}_1, \ldots, \boldsymbol{m'}_t\right)$

1. P_2 computes $\langle \boldsymbol{m} \rangle \coloneqq \bigoplus_{i=1}^t \langle \boldsymbol{m}_i \rangle$ and $\boldsymbol{m'} = \bigoplus_{i=1}^t \boldsymbol{m'}_i$.

2. P_2 parses $\langle \boldsymbol{m} \rangle$ into $(x_{i_1}, \ldots, x_{i_w}) \in \{0,1\}^{\sigma \cdot w}$ and returns 1 if for all $i \in \{i_1, \ldots, i_w\}$, $\mathsf{Encode}(\boldsymbol{m'})[i] = x_i$; and 0, otherwise.

(Setting $t = 1$ allows P_2 to verify any single messages.)

Optimization. If the goal is only to i-hash a random message, it suffices to treat the x_i's (generated by calling PRG in Step 1a) as the random messages to i-hash, hence no need to execute Step 1b (which sends the `xor`-differences between the specified messages and the random messages), saving $(n - \ell)\sigma$ bits per i-hashed message.

4.3 Proof of Security

Theorem 4.1 *The protocol described in Section 4.2 securely realizes the ideal functionality of XOR-homomorphic interactive hash defined in Figure 4.*

Proof (Sketch) We examine all three interfaces of interactive hash in a hybrid model where an ideal OT functionality is available.

1. Hash: For a corrupted sender P_1, the simulator $\mathcal{S}$ interacts with P_1 like P_2 in a OT-hybrid model where it can extracts all the seeds thus recover the messages $\boldsymbol{m}_1, \ldots, \boldsymbol{m}_\nu$ and sends them to the ideal interactive hash functionality in Step 3 of Hash and outputs whatever P_1 outputs.
   For a corrupted P_2, it is trivial to construct $\mathcal{S}$ as P_2 has no secret input. The indistinguishability of the two executions can be derived from the linearity and threshold secret sharing properties of Reed-Solomon codes.
2. Verify: This case does not involve interaction. The security proof can be derived from the linearity of Reed-Solomon codes and a corollary of Lemma 4.3. □

Lemma 4.2 *If* $\mathsf{IHash}_{\ell,\sigma}.\mathsf{Hash}(\boldsymbol{m}_1, \ldots, \boldsymbol{m}_\nu)$ *succeeds, with all but* 2^{-s} *probability, the* ν *random messages being hashed must be* $\{\boldsymbol{m}_1, \ldots, \boldsymbol{m}_\nu\}$.

Lemma 4.3 *Let* $\mathbf{H}_{min}$ *be the min-entropy function. For* $\mathbf{H}_{min}(\boldsymbol{m}) = \sigma \cdot \ell$ *where* $\boldsymbol{m} \in \{0,1\}^{\sigma \cdot \ell}$ *and* $\langle \boldsymbol{m} \rangle \in \{0,1\}^{\sigma \cdot w}$,

1. $\mathbf{H}_{min}(\boldsymbol{m} | \langle \boldsymbol{m} \rangle) = (\ell - w)\sigma$. *I.e.,* $(\ell - w)\sigma$ *entropy remains even if* P_2 *learns* $\langle \boldsymbol{m} \rangle$.
2. $\forall \boldsymbol{m} \neq \boldsymbol{m}'$, $\Pr\left[\mathsf{IHash}_{\ell,\sigma}.\mathsf{Verify}(\langle \boldsymbol{m} \rangle, \boldsymbol{m}') = 1\right] \leq \binom{\ell-1}{w} / \binom{n}{w}$, *with probability taken over the randomness of* P_1 *and* P_2.

Due to page limit, we prove these lemmas in Appendix A.1 and A.2.

5 The Main Protocol

5.1 Protocol Description

Assume P_1 (the generator) and P_2 (the evaluator) wish to compute f over secret inputs x, y, where f is realized as a boolean circuit C that has only AND and XOR gates. The protocol proceeds as follows.

0. **Setup.** The parties decide the public parameters $\ell_w, \sigma_w, \ell_p, \sigma_p$ from the security parameters s, k (see Section 6.1 and 6.2 for the detailed discussion).
   (a) P_1 randomly picks $seed_1, \ldots, seed_{n_w}$; P_2 randomly picks $i_1, \ldots, i_{n_w}$. P_1 (as the sender using $seed_1, \ldots, seed_{n_w}$) and P_2 (as the receiver using $i_1, \ldots, i_{n_w}$) run $\mathsf{IHash}_{\ell_w,\sigma_w}.\mathsf{Setup}$ to initialize the IHash scheme for i-hashing wire-labels.
   (b) P_1 randomly picks $\Delta \in \{0,1\}^{\lambda_w}$ where $\lambda_w = \ell_w \sigma_w$ and calls $\mathsf{IHash}_{\ell_w,\sigma_w}.\mathsf{Hash}$ to send $\langle \Delta \rangle$ to P_2.

(c) P_1 (using seeds $H(\Delta, 1), \ldots, H(\Delta, n_p)$ where H is a random oracle) and P_2 (using freshly sampled indices $i'_1, \ldots, i'_{n_p}$) run $\mathsf{IHash}_{\ell_p,\sigma_p}.\mathsf{Setup}$ to initialize the IHash scheme for wire permutation strings.

(d) P_1 sends $H(H(\Delta, 1)), \ldots, H(H(\Delta, n_p))$ to P_2.

Then, P_1 randomly select 16 linearly-independent vectors $a_1, \ldots, a_{16}$ from $\mathrm{GF}(8)^{\ell_w}$, which will be row vectors of the matrix to be left multiplied with a wire-label to realize the $\mathsf{Compress}$ function (compressing a 256-bit wire-label into 128-bit, see Fig. 3). P_1 sends $a_1, \ldots, a_{16}$ to P_2.

1. **Circuit Initialization.** Let n_w be the total number of wires in C. P_1 picks $m_1, \ldots, m_{n_w} \in \{0,1\}^{\lambda_w}$ where $\lambda_w = \ell_w \sigma_w$; then run $\mathsf{IHash}_{\ell_w,\sigma_w}.\mathsf{Hash}$ with P_2 to send $\langle m_1 \rangle, \ldots, \langle m_{n_w} \rangle$ to P_2. Then, P_1 samples $\rho^1, \ldots, \rho^{n_w} \in \{0,1\}^{\lambda_p}$ where $\lambda_p = \ell_p \sigma_p$, then run $\mathsf{IHash}_{\ell_p,\sigma_p}.\mathsf{Hash}$ with P_2 to send $\langle \rho^{i_1} \rangle, \ldots, \langle \rho^{i_w} \rangle$ to P_2. For all $1 \le i \le n_w$, P_1 sets $p_i = \rho^i_1 \oplus \cdots \oplus \rho^i_{\lambda_p}$ (where ρ^i_j denotes the j-th bit of ρ^i) and $w^0_i \coloneqq m_i \oplus p_i \Delta$. Let $w^1_i = m_i \oplus \bar{p}_i \Delta$, hence $m_i = w^{p_i}_i$. Then, P_1 and P_2 process the initial input-wires as follows.
   (a) For $1 \le i \le n_I^{P_1}$, let (w^0_i, w^1_i) be the pair of wire labels on the wire associated with x_i, P_1 sends $w^{x_i}_i$ to P_2.
   (b) For every input-wire W_i associated with P_2's private input y_i:
      i. W_i is $\oplus$-split into s wires $W_{i,1}, \ldots, W_{i,s}$.
      ii. P_1 picks $m_1, \ldots, m_s$ and run $\mathsf{IHash}_{\ell_w,\sigma_w}.\mathsf{Hash}$ with P_2 to send $\langle m_1 \rangle, \ldots, \langle m_s \rangle$ to P_2. For $1 \le j \le s$, P_1 sets $w^0_{i,j} = m_j$ and $w^1_{i,j} = w^0_{i,j} \oplus \Delta$.
      iii. P_2 samples $y_{i,1} \leftarrow \{0,1\}, \ldots, y_{i,s} \leftarrow \{0,1\}$ such that $y_{i,1} \oplus \cdots \oplus y_{i,s} = y_i$.
      iv. For $1 \le j \le s$, P_2 retrieves $w^{y_{i,j}}_{i,j}$ from P_1 through oblivious transfer, and verifies $w^{y_{i,j}}_{i,j}$ against $\langle w^{y_{i,j}}_{i,j} \rangle$ (note P_2 can compute $\langle w^{y_{i,j}}_{i,j} \rangle \coloneqq \langle w^0_{i,j} \rangle \oplus y_{i,j} \langle \Delta \rangle$). Any verification failure will result in P_2's delayed abort at Step 5.
      v. P_2 sets $w^{y_i}_i \coloneqq w^{y_{i,1}}_{i,1} \oplus \cdots \oplus w^{y_{i,s}}_{i,s}$, $\langle w^{y_i}_i \rangle \coloneqq \langle w^{y_{i,1}}_{i,1} \rangle \oplus \cdots \oplus \langle w^{y_{i,s}}_{i,s} \rangle$, and $p_i = 0$.

2. **Generate.** P_2 randomly picks $\mathcal{J} \in \{0,1\}^k$ and commits it to P_1. $\mathcal{J}$ will be used as the randomness for cut-and-choose later.
   (a) P_1 picks $2T$ random λ_w-bit messages ($\lambda_w = \ell_w \sigma_w$) and run $\mathsf{IHash}_{\ell_w,\sigma_w}.\mathsf{Hash}$ with P_2 to send the i-hashes of the $2T$ random messages. Denote the $2T$ messages by $\{m_{i,l}, m_{i,r}\}_{i=1}^T$, and their i-hashes by $\{\langle m_{i,l} \rangle, \langle m_{i,r} \rangle\}_{i=1}^T$.
   (b) P_1 picks $3T$ random λ_p-bit ($\lambda_p = \ell_p \sigma_p$) messages and run $\mathsf{IHash}_{\ell_p,\sigma_p}.\mathsf{Hash}$ with P_2 to send the i-hashes of these $3T$ random messages. Denote these messages and i-hashes by $\{\rho^{i,l}, \rho^{i,r}, \rho^{i,o}\}_{i=1}^T$ and $\{\langle \rho^{i,l} \rangle, \langle \rho^{i,r} \rangle, \langle \rho^{i,o} \rangle\}_{i=1}^T$. P_1 computes $p_{i,l} = \rho^{i,l}_1 \oplus \cdots \oplus \rho^{i,l}_{\lambda_p}$, where $\rho^{i,l}_j$ denotes the j^{th} bit of $\rho^{i,l}$. Similarly, P_1 derives $p_{i,r}$ and $p_{i,o}$ from $\rho^{i,r}$ and $\rho^{i,o}$, respectively. ($p_{i,l}, p_{i,r}, p_{i,o}$ will be used as the i-hash permutation bits on the three wires connected to a garbled AND gate.)
   (c) For $i = \{1, \ldots, T\}$, P_1 sets $w^0_{i,l} \coloneqq m_{i,l} \oplus p_{i,l} \Delta$ and $w^0_{i,r} \coloneqq m_{i,r} \oplus p_{i,r} \Delta$, then runs the garbling algorithm GenAND (Fig. 3) to create T garbled

AND gates:

$$(w^0_{i,o}, T_{i,G}, T_{i,E}) \leftarrow \mathsf{GenAND}(i, \Delta, w^0_{i,l}, w^0_{i,r})$$

where $w^0_{i,l}, w^0_{i,r}, w^0_{i,o}$ are the wire labels representing 0's on the left input-wire, the right input wire, and the output-wire, respectively; $T_{i,G}$ is the single garbled row in the generator half-gate and $T_{i,E}$ the single row in the evaluator half-gate.

(d) Let $w^1_{i,o} \coloneqq w^0_{i,o} \oplus p_{i,o}\Delta$ for all $1 \le i \le T$. P_1 and P_2 run

$$\mathsf{IHash}_{\ell_w,\sigma_w}.\mathsf{Hash}\left(w^{p_{1,o}}_{1,o}, \ldots, w^{p_{T,o}}_{T,o}\right)$$

so that P_2 learns $\left\langle w^{p_{1,o}}_{1,o}\right\rangle, \ldots, \left\langle w^{p_{T,o}}_{T,o}\right\rangle$.

3. **Evaluate.** P_2 opens to P_1 the cut-and-choose randomness $\mathcal{J}$, which is used to select and group $B \cdot N$ garbled ANDs into N buckets.

Recall that for every logical gate in C, P_2 has obtained from step 1 two wire-labels w^a_l, w^b_r, which correspond to secret values a, b on the input-wires and i-hashes $\langle\rho^l\rangle, \langle w^{p_l}_l\rangle, \langle\rho^r\rangle, \langle w^{p_r}_r\rangle, \langle\rho^o\rangle, \langle w^{p_o}_o\rangle$. P_1 and P_2 follow an identical topological order to process the logical gates as follows: For every XOR, P_2 sets $w_o \coloneqq w^a_l \oplus w^b_r$; For every logical AND (Fig. 5), we denote the B garbled AND gates by $g_1, \ldots, g_B$,

(a) P_2 sets $\mathcal{O}$ to an empty set and executes the following for $1 \le i \le B$ (note that P_2 always continues execution until Step 5 even if any check failed),

i. Let $p_{i,l}$ be the i-hash permutation bit of the left input-wire of g_i, i.e., $p_{i,l} = g_i.p_l$. Let $p_{i,r}, p_{i,o}, \rho^{i,l}, \rho^{i,r}, \rho^{i,o}$ be similarly defined. P_1 sends $\rho^l \oplus \rho^{i,l}$, $\rho^r \oplus \rho^{i,r}$ and $\rho^o \oplus \rho^{i,o}$ to P_2, who verifies them against their i-hashes and computes

$$p_l \oplus p_{i,l} \coloneqq \bigoplus_{1\le j\le\lambda_p} (\rho^l_j \oplus \rho^{i,l}_j)$$
$$p_r \oplus p_{i,r} \coloneqq \bigoplus_{1\le j\le\lambda_p} (\rho^r_j \oplus \rho^{i,r}_j)$$
$$p_o \oplus p_{i,o} \coloneqq \bigoplus_{1\le j\le\lambda_p} (\rho^o_j \oplus \rho^{i,o}_j).$$

ii. For $b \in \{0,1\}$, define $w^b_{i,l}$ be the wire-label representing signal b on gate g_i's left input-wire, and let $w^b_{i,r}$, $w^b_{i,o}$ be similarly defined with g_i's right input-wire and output-wire. P_1 sends

$$\delta_l \coloneqq w^{p_l}_l \oplus w^{p_{i,l}}_{i,l} \oplus (p_l \oplus p_{i,l})\Delta$$
$$\delta_r \coloneqq w^{p_r}_r \oplus w^{p_{i,r}}_{i,r} \oplus (p_r \oplus p_{i,r})\Delta$$
$$\delta_o \coloneqq w^{p_o}_o \oplus w^{p_{i,o}}_{i,o} \oplus (p_o \oplus p_{i,o})\Delta$$

to P_2, who verifies them against their hashes and computes $w^a_{i,l} \coloneqq w^a_l \oplus \delta_l$ and $w^b_{i,r} \coloneqq w^b_r \oplus \delta_r$.

iii. Recall that $T_{i,G} = g_i.T_G, T_{i,E} = g_i.T_E$. P_2 runs $w_{i,o} \coloneqq \mathsf{EvlAND}(w_l^a, w_r^b, T_{i,G}, T_{i,E})$, and sets $w_o \coloneqq w_{i,o} \oplus \delta_o$.

iv. P_2 verifies w_o against $\langle w_o^{p_o} \rangle$ and $\langle w_o^{p_o} \rangle \oplus \langle \Delta \rangle$. If either verification succeeds, P_2 adds w_o to $\mathcal{O}$.

(b) If $\mathcal{O}$ contains two different labels, say w and w'. P_2 computes $\Delta^* \coloneqq w \oplus w'$, and uses Δ^* to recover P_1's private inputs x and computes $f(x, y)$. Otherwise, $\mathcal{O} = \{w\}$ so P_2 sets $w_o = w$.

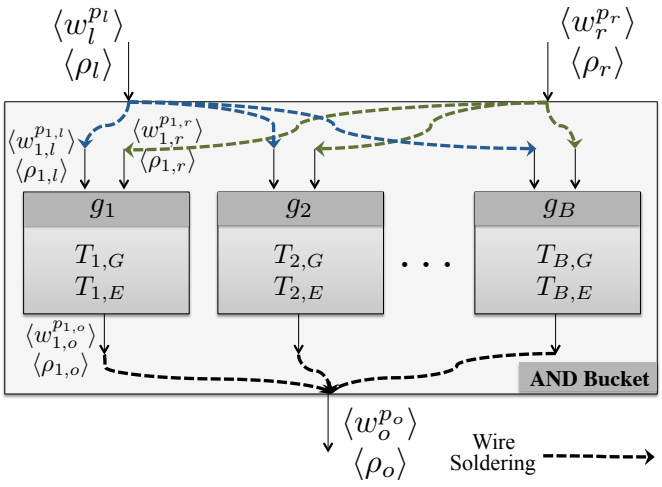

Fig. 5: A bucket of B garbled gates. (Wire labels and hashes exist at both the bucket-level (e.g. $w_l^{p_l}$) and gate-level (e.g. $w_l^{p_{i,l}}$).)

4. **Check.** P_2 verifies the correctness of the rest $T - BN$ garbled AND gates. For every check-gate parsed into

$$\left(\langle \rho^l \rangle, \langle w_l^{p_l} \rangle, \langle \rho^r \rangle, \langle w_r^{p_r} \rangle, \langle \rho^o \rangle, \langle w_o^{p_o} \rangle, T_G, T_E\right),$$

(a) P_2 samples $a \leftarrow \{0,1\}, b \leftarrow \{0,1\}$, sends them to P_1.

(b) P_1 sends ρ^l, ρ^r, ρ^o to P_2. P_2 verifies them with $\langle \rho^l \rangle, \langle \rho^r \rangle$, and $\langle \rho^o \rangle$. Let $p_l = \rho_1^l \oplus \cdots \oplus \rho_{\lambda_p}^l$, $p_r = \rho_1^r \oplus \cdots \oplus \rho_{\lambda_p}^r$, $p_o = \rho_1^o \oplus \cdots \oplus \rho_{\lambda_p}^o$,

i. P_1 sends $w_l^a = w_l^{p_l} \oplus (a \oplus p_l)\Delta$ to P_2, who verifies it against $\langle w_l^{p_l} \rangle \oplus (a \oplus p_l)\langle \Delta \rangle$.

ii. P_1 sends $w_r^b = w_r^{p_r} \oplus (b \oplus p_r)\Delta$ to P_2, who verifies it against $\langle w_r^{p_r} \rangle \oplus (b \oplus p_r)\langle \Delta \rangle$.

iii. Let $z = a \wedge b$. P_1 sends $w_o^z = w_o^{p_o} \oplus (z \oplus p_o)\Delta$ to P_2, who verifies it against $\langle w_o^{p_o} \rangle \oplus (z \oplus p_o)\langle \Delta \rangle$.

(c) P_2 checks $w_o^z = \mathsf{EvlAND}(w_l^a, w_r^b, T_G, T_E)$.

5. **Output determination.**

(a) If any check failed in steps 3 and 4, P_2 aborts.

(b) P_1 proves in zero knowledge that it executes the Step 0b and Step 0c honestly. Namely, the double-hashes received in Step 0d, the i-hash $\langle \Delta \rangle$ hold by P_2, and the watched seeds of $\mathsf{IHash}_{\ell_p,\sigma_p}.\mathsf{Setup}$ are all respect to the same Δ. P_2 aborts if the ZK proof fails.

(c) Otherwise, P_2 outputs $f(x, y)$, either from a recovered Δ or interpreted final output labels.

Remarks In practice, Step 5b is very efficient — using recent ZK proof techniques [27,54,17], it requires only $2n_p$ semi-honest garbled circuit executions of SHA256.

5.2 Proof of Security

First, we show that for any N and security parameter s, k, it is possible to set parameters T, B, n, ℓ, w such that our protocol securely computes $f(x, y)$ (except with probability 2^{-s}). For concrete values of s, k, N, we detail how to optimize T, B, n, ℓ, w for performance in Section 6.3.

Lemma 5.1 *For any N, s, k, there exist T, B such that if P_2 does not abort at Step 5, then with all but 2^{-s} probability every bucket has at least one correctly garbled gate.*

Lemma 5.1 is essentially a side-product of Section 6.3.

Lemma 5.2 *If every bucket has at least one correctly garbled gate, P_2 will output $f(x, y)$ at Step 5 except with negligible probability.*

Proof. Evaluating an incorrectly garbled gate will yield an output wire-label that either matches $\langle w_o^{p_o} \rangle$, or $\langle w_o^{p_o} \rangle \oplus \langle \Delta \rangle = \langle w_o^{p_o} \oplus \Delta \rangle$, or none of them.

1. If the output label matches with neither i-hashes, the evaluation result will be ignored (Step 3(a)iv);
2. If the output label matches with an i-hash and represents the same plain-text value as the output label obtained from evaluating the correctly garbled gate in the same bucket, the corrupted garbled gate does not affect the evaluation.
3. If the output label matches with an i-hash but represents the opposite plain-text value as the output label obtained from evaluating the correctly garbled gate, P_2 learns Δ at step 3b. Then P_2 will be able to learn P_1's input x by examining the buckets that take bits of x as immediate inputs. Note that P_2 can derive all the wire permutation string from Δ, hence, along with the wire-labels representing P_1's input (whose validity is guaranteed by the corresponding i-hashes), P_2 is able to precisely infer every bits of x.

In all cases, P_2 can correctly output $f(x, y)$ with all but negligible probability. □

Lemma 5.3 *Given $\langle \Delta \rangle$, a garbled AND gate (T_G, T_E), and the i-hashes of its wire-labels and permutation messages $\big(\langle \rho^l \rangle, \langle w_l^{p_l} \rangle, \langle \rho^r \rangle, \langle w_r^{p_r} \rangle, \langle \rho^o \rangle, \langle w_o^{p_o} \rangle\big)$, where $p_l = \rho_1^l \oplus \ldots \oplus \rho_{\lambda_p}^l$, $p_r = \rho_1^r \oplus \ldots \oplus \rho_{\lambda_p}^r$, $p_o = \rho_1^o \oplus \ldots \oplus \rho_{\lambda_p}^o$. If any of the following is not satisfied (see Fig. 3 for* EvlAND*),*

$$\mathsf{EvlAND}(w_l^0, w_r^0, T_G, T_E) = w_o^0;\ \mathsf{EvlAND}(w_l^0, w_r^1, T_G, T_E) = w_o^0;$$
$$\mathsf{EvlAND}(w_l^1, w_r^0, T_G, T_E) = w_o^0;\ \mathsf{EvlAND}(w_l^1, w_r^1, T_G, T_E) = w_o^1,$$

then P_2 will detect this with probability at least 1/2 at Step 4.

Proof. By the definition of EvlAND, the four equations above are essentially

$$H(w_l^0) \oplus \mathsf{lsb}(w_l^0) T_G \oplus H(w_r^0) \oplus \mathsf{lsb}(w_r^0)(T_E \oplus w_l^0) = w_o^{p_o}$$
$$H(w_l^1) \oplus \mathsf{lsb}(w_l^1) T_G \oplus H(w_r^0) \oplus \mathsf{lsb}(w_r^0)(T_E \oplus w_l^1) = w_o^{p_o}$$
$$H(w_l^0) \oplus \mathsf{lsb}(w_l^0) T_G \oplus H(w_r^1) \oplus \mathsf{lsb}(w_r^1)(T_E \oplus w_l^0) = w_o^{p_o}$$
$$H(w_l^1) \oplus \mathsf{lsb}(w_l^1) T_G \oplus H(w_r^1) \oplus \mathsf{lsb}(w_r^1)(T_E \oplus w_l^1) = w_o^{p_o} \oplus \Delta.$$

That is,

$$\mathsf{lsb}(w_l^0) T_G \oplus w_o^{p_o} \oplus \mathsf{lsb}(w_r^0) T_E = H(w_l^0) \oplus H(w_r^0) \oplus \mathsf{lsb}(w_r^0) w_l^0$$
$$\mathsf{lsb}(w_l^1) T_G \oplus w_o^{p_o} \oplus \mathsf{lsb}(w_r^0) T_E = H(w_l^1) \oplus H(w_r^0) \oplus \mathsf{lsb}(w_r^0) w_l^1$$
$$\mathsf{lsb}(w_l^0) T_G \oplus w_o^{p_o} \oplus \mathsf{lsb}(w_r^1) T_E = H(w_l^0) \oplus H(w_r^1) \oplus \mathsf{lsb}(w_r^1) w_l^0$$
$$\mathsf{lsb}(w_l^1) T_G \oplus w_o^{p_o} \oplus \mathsf{lsb}(w_r^1) T_E = H(w_l^1) \oplus H(w_r^1) \oplus \mathsf{lsb}(w_r^1) w_l^1 \oplus \Delta,$$

which can be viewed as a linear system of four equations over three variables T_G, T_E, and $w_o^{p_o}$. Note that all coefficients on the left-hand side and all constants on the right-hand side of the equations are fixed by the seven i-hashes known to P_2. Also note that any three out of the four equations are linearly independent except with negligible probability, because H is modeled as a random oracle and $\mathsf{lsb}(w_l^0) \oplus \mathsf{lsb}(w_l^1) = 1$, $\mathsf{lsb}(w_r^0) \oplus \mathsf{lsb}(w_r^1) = 1$, $w_l^0 \oplus w_l^1 = w_r^0 \oplus w_r^1 = \Delta$. Thus, if any three of the four equations hold, the fourth one will be automatically satisfied as it is simply a linear combination of the other three. In addition, we know there must be one solution to the system of four equations if P_1 follows the specification of GenAND. Therefore, if $T_G, T_E, w_o^{p_o}$ take some corrupted values such that any one equation does not hold, there has to be at least one other equation that does not hold (otherwise, $T_G, T_E, w_o^{p_o}$ have to satisfy all four equations). Therefore, if the gate is corrupted, at least two of the equations fail to hold, allowing P_2 to detect it with probability at least $1/2$ when it randomly checks one equations at step 4c. □

Theorem 5.4 *Under the assumptions outlined in Section 3, the protocol in Section 5.1 securely computes f in the presence of malicious adversaries.*

Proof (Sketch) We analyze the protocol in a hybrid world where the parties have access to ideal functionalities for 1-out-of-2 oblivious transfer, commitment, and interactive hash. The standard composition theorem [11] implies security

when the sub-routines are instantiated with secure implementations of these functionalities.

If P_1 is corrupted. We construct a polynomial-time simulator $\mathcal{S}$ that interacts with the corrupted P_1 as P_2 with input $y = 0$ in the protocol of Section 5.1, except for the following changes:

1. All invocations of interactive hash protocol is replaced with calls to the ideal interactive hash functionality simulated by $\mathcal{S}$.
2. At Step 1a, on receiving $w_i^{x_i}$, $\mathcal{S}$ learns P_1's input x_i for all $1 \leq i \leq n_I^{P_1}$, using its knowledge of $\rho^i, p_i, w_i^{p_i}$ extracted from the ideal interactive hash functionality.
3. At Step 5 of **Output determination**, if $\mathcal{S}$ does not abort, (instead of outputs $f(x, 0)$), $\mathcal{S}$ sends x to the trusted party and receives in return $z = f(x, y)$. $\mathcal{S}$ rewrites P_1's output $f(x, 0)$ with z.

We can infer $\mathbb{REAL}^{P_1,P_2}(x, y) \approx \mathbb{IDEAL}^{\mathcal{T},\mathcal{S},P_2}(x, y)$ for every x, y, where $\mathbb{REAL}$ and $\mathbb{IDEAL}$ are defined canonically as in [18, Definition 7.2.6], from two basic observations:

1. If P_2 aborts in the real execution, $\mathcal{S}$ will also abort in the ideal execution as the changes in $\mathcal{S}$ from P_2 does not affect their abort behavior.
2. If P_2 does not abort in the real execution, by Lemma 5.1 and Lemma 5.2, it will output $f(x, y)$ in the ideal execution except for negligible probability. Over exactly identical inputs and random tapes, $\mathcal{S}$ will not abort either and will send the extracted x to the trusted party in the ideal world to obtain the same outcomes as the real execution.

If P_2 is corrupted. An efficient simulator $\mathcal{S}$ can be constructed that interacts with the corrupted P_2 as P_1 with input $x = 0$ using the Section 5.1 protocol, except for the following changes:

1. At Step 1b, an ideal 1-out-of-2 OT functionality is used for P_2 to obtain wire labels on the split input-wires. This allows $\mathcal{S}$ to extract $y_{i,1}, \ldots, y_{i,s}$, thus learn P_2's effective input $y_i \coloneqq y_{i,1} \oplus \cdots \oplus y_{i,s}$ for all $1 \leq i \leq n_I^{P_2}$.
2. At Step 2, an ideal commitment functionality (simulated by $\mathcal{S}$) is used to commit $\mathcal{J}$. This allows $\mathcal{S}$ to extract the cut-and-choose string $\mathcal{J}$. $\mathcal{S}$ sends the y to the trusted party and receive in return $z = f(x, y)$. Then, $\mathcal{S}$ generates the T garbled AND gates such that all gates to be checked are correct; and all gates filling the bucket of the i^{th} final output-wire evaluate to constant z_i.

We can infer $\mathbb{REAL}^{P_1,P_2}(x, y) \approx \mathbb{IDEAL}^{\mathcal{T},P_1,\mathcal{S}}(x, y)$ for every x, y from the following observations:

1. If P_1 aborts in the real execution, $\mathcal{S}$ will also abort in the ideal execution. The real model P_2 learns (at the best) a transcript of garbled circuits, which, by the privacy and obliviousness properties of garbling scheme, is computationally indistinguishable from that generated by $\mathcal{S}$.

2. If P_1 does not abort in the real execution, P_1 will output $f(x, y)$ in the ideal execution as well (except with negligible probability). Over exactly identical inputs and random tapes, $\mathcal{S}$ will not abort either and will extract y to obtain $f(x, y)$ from the trusted party.

□

6 Parameters and Bounds

6.1 IHash-ing Permutation Messages

Here the goal is to decide the best parameters $n_p, \ell_p, w_p, \sigma_p$ that are used to i-hash the wire permutation messages, i.e., the ρ's used in the main protocol, to achieve the necessary binding and hiding properties. This can be framed into a constrained optimization problem:

$$(n_p, \ell_p, w_p, \sigma_p) = \arg\min \ \mathsf{cost}(n, \ell, w, \sigma)$$

subject to:

$$\binom{\ell-1}{w} \Big/ \binom{n}{w} \leq 2^{-s} \tag{1}$$

$$\sigma(\ell - w) \geq k \tag{2}$$

$$2^{\sigma} \geq n$$

$$n, \ell, \sigma, w \in \mathbb{Z}^+$$

where inequality (1) ensures s-bit statistical binding, inequality (2) ensures k-bit hiding, and the target cost function can depend on a number of deployment-specific tradeoffs between bandwidth and computation.

We stress that hiding for the permutation messages is *perfect* because there is no additional information revealed to allow a malicious evaluator to verify its guess on the hidden bits (comparing to the fact that garbled truth table can be used to verify guesses about the wire-labels). Once the cost function is fixed, an efficient solver through aggressive pruning can be constructed. In our experiment, we set $n_p = 44, \ell_p = 20, w_p = 19, \sigma_p = 6$, which provides 40-bit statistical binding (verify this by plugging them into (1)) and 6-bit perfect hiding since $(\ell_p - w_p) \cdot \sigma_p = 6$ (although only 1-bit perfect hiding is needed).

6.2 IHash-ing and Compress-ing Wire-labels

Here our goal is to determine the best parameters $n_w, \ell_w, w_w, \sigma_w$ to process the wire-labels so that s-bit statistical security and k-bit computational security can be guaranteed for the main protocol. Note the entropy hiding on the wire-labels downgrades to *computational* because a malicious evaluator could run *offline* tests on its guesses of the wire-labels using the garbled rows.

To ensure 40-bit binding and at least 80-bit hiding, $\ell_w \cdot \sigma_w$ (the wire-labels' length) has to be more than 128 bits. This poses a challenge to efficient garbling using fixed-key AES assembly instructions since AES only works on 128-bit blocks. We solve this challenge by making $(\ell_w - w_w)\sigma_w$ slightly larger (i.e., by a

factor of $1+\varepsilon$) than k, followed by a linear Compress function to derive 128-bit compressed labels that each carries more than 80-bit entropy from the original, watched wire-labels. Namely, for wire-labels, we replace constraint (2) by

$$\sigma_w(\ell_w - w_w) \geq (1+\varepsilon)k$$

where $\varepsilon > 0$ compensates the entropy loss during the compression. We choose $n_w = 86, \ell_w = 32, w_w = 21$ and $\sigma_w = 8$.

To compress wire-labels, the generator samples $128/\sigma_w = 16$ linear-independent vectors (over $\mathrm{GF}(2^8)^{32}$) once and left-multiply them to 256-bit wire-labels to obtain 128-bit compressed labels.

Recall that these 16 linear-independent vectors are declared only after the evaluator chose its 21 watch symbols. The entropy analysis of this Compress function can be done by considering the following experiment:

1. P_1 randomly samples 32 symbols, $m_1, \ldots, m_{32} \in \mathrm{GF}(2^8)$.
2. P_2 randomly chooses 21 linear-independent vectors $\boldsymbol{W} = (W_1, \ldots, W_{21})$ from $\mathrm{GF}(2^8)^{32}$ and thus learns $(m_1, \ldots, m_{32}) \cdot W_i$ for all $1 \leq i \leq 21$.
3. P_1 randomly chooses and sends 16 linear-independent vectors $\boldsymbol{T} = (T_1, \ldots, T_{16}) \in \mathrm{GF}(2^8)^{32}$, then outputs $v_1, \ldots, v_{16}$, where $v_i = (m_1 \ \ldots \ m_{32}) \cdot T_i$ for all $1 \leq i \leq 16$.

The question is: how much entropy in the output $v_1, \ldots, v_{16}$ remains hidden to P_2? In other words, for every $\mathcal{A}$, every rank-21 matrix $\boldsymbol{W} \in \mathrm{GF}(2^8)^{32\times 21}$ and every rank-16 matrix $\boldsymbol{T} \in \mathrm{GF}(2^8)^{32\times 16}$, define

$$Q = \Pr\left(\boldsymbol{m} \leftarrow \mathrm{GF}(2^8)^{32} \ : \ \mathcal{A}(\boldsymbol{W}, \boldsymbol{T}, \boldsymbol{m}\cdot\boldsymbol{W}) = \boldsymbol{m}\cdot\boldsymbol{T} \right).$$

We want to know the *min-entropy* of $\boldsymbol{m}\cdot\boldsymbol{T}$, which is essentially $\log(1/Q)$. We answer this question with Lemma 6.1 and elaborate the analysis in its proof in Appendix A.4.

Lemma 6.1 *Setting $n_w = 86, \ell_w = 32, w_w = 21$ and $\sigma_w = 8$ ensures 40-bit statistical binding and more than 87.999 bits hiding in the compressed wire-labels; while setting $n_w = 88, \ell_w = 48, w_w = 32$ and $\sigma_w = 8$ ensures 40-bit statistical binding and more than 127 bits hiding in the compressed wire-labels.*

Remark. As an alternative, the generic Leftover Hash Lemma [26,6] could be used to solve the wire-label entropy extraction problem. However, the potentially large entropy loss ($2\log(1/2^{-80}) + O(1) = 160 + O(1)$ bits) makes it unsuitable in our case. In contrast, less than 10^{-11} bit of entropy (out of the $8\cdot(32-21) = 88$ bits remaining entropy before Compress-ing) will actually be lost due to our Compress function (see Lemma 6.1's proof)!

6.3 LEGO Cut-and-Choose Parameters

While existing analysis of LEGO cut-and-choose rely on empirical point trials in a likely area of the parameter space, we show that the search can be fully guided to efficiently identify the best cut-and-choose parameters in practical scenarios.

Recall the goal is to determine the best T, B to ensure s-bit statistical security for computing a circuit of N gates. Assume out of the total T garbled gates, b of them are faulty gates. Let P_{c} be the probability of selecting a particular set of $T - BN$ gates in which t gates are faulty. Then $P_{\mathrm{c}} = \binom{b}{t}\binom{T-b}{T-NB-t} \Big/ \binom{T}{T-NB}$. Let P_{e} be the probability that at least one bucket is filled entirely by faulty gates, then $P_{\mathrm{e}} \leq N\binom{b}{B} \Big/ \binom{NB}{B}$ where the equality is approached from below when $N \gg B$. Since the overall failure probability $P_{\mathrm{overall}} \leq \max_b \sum_{t=1}^{b} 2^t P_{\mathrm{c}} P_{\mathrm{e}}$, it suffices to find the smallest T such that $\max_b \sum_{t=1}^{b} 2^{-t} P_{\mathrm{c}} P_{\mathrm{e}} \leq 2^{-s}$. Note that we only need to consider b values up to a upper bound slightly larger than s because $P_{\mathrm{c}} P_{\mathrm{e}} < 1$ and $\sum 2^{-t}$ converges as $t \to \infty$. In addition, we observe that the smallest T for any fixed B (we call T_B) can be quickly determined since P_{overall} strictly decreases when T grows. Thus, T, B can be efficiently determined through pruning when examining $B = 2, 3, ...\lceil T_B/N \rceil$.

6.4 A Fallacy and A Tight Bound

Define $\kappa = T/N$. Prior works claimed $O(\kappa) = O(T/N) = O((skN/\log N)/N) = O(sk/\log N)$, implying $\kappa \to 0$ when $N \to +\infty$ [14,42,15]. However, we found that this is not the case. More precisely, T should be $O(\kappa \cdot N + skN/\log N)$ and *2 is a tight bound of* κ. That is, $\kappa \leq 2$ cannot be achieved without compromising security while any $\kappa > 2$ is securely achievable (using our protocol) if N is large enough. However, the formal proof of this seemingly intuitive result is nontrivial.

Theorem 6.2 *Let* $\kappa = T/N$ *and* $\Pr_{overall}$ *be the overall success rate of* P_1 *attacking LEGO-style cut-and-choose.*

1. *If* $\kappa \leq 2$*, there exists a constant* $c > 0$ *and an integer* N_0 *such that* $\Pr_{overall} > c$ *for all* $N > N_0$*.*
2. *For every statistical security parameter* s *and computational security parameter* k*, there exists an integer* N_0 *such that the protocol of Section 5.1 with a* $\kappa > 2$ *securely computes all circuits of size* $N > N_0$*.*

The proof of Theorem 6.2 is given in Appendix B.

7 Evaluation

Measurement Methodology. We ran experiments on Amazon EC2 (instance type: `c4.2xlarge`) running Ubuntu Linux in both the LAN (2.5 Gbps, < 1 ms latency) and WAN (200 Mbps, 20 ms latency) network settings. Our IHash parameters are listed in Fig. 6. As our comparison baseline, we chose WMK [50] and TinyLEGO [15,44], two implementations of single-execution setting protocols representing the state-of-the-art. All comparisons are aligned on the same hardware and network environment, based on single-threaded executions. We

	n_w	l_w	σ_w	w_w	n_p	l_p	σ_p	w_p
$s = 40, k = 88$	86	32	8	21	44	20	6	19
$s = 40, k = 127$	88	48	8	32	44	20	6	19

Fig. 6: IHash parameters.

	Garble		**Check**		**Evaluate**		**Solder**	
	CPU	BW	CPU	BW	CPU	BW	CPU	BW
$k = 88$	1.93	327	1.16	127	0.75	106	0.37	159
$k = 127$	2.37	333	1.40	159	0.79	138	0.48	207

(a) Cost per garbled gate.

	Per P_1's Input		**Per P_2's Input**		**Per Output**	
	CPU	BW	CPU	BW	CPU	BW
$k = 88$	0.29	86	4.11	5360	0.028	32
$k = 127$	0.34	88	4.18	6400	0.031	48

(b) Cost per input-/output-wire.

Fig. 7: Microbenchmarks. (The unites are either microsecond or byte. CPU timings do not include network time. Timings are averaged over 10^6 executions.)

include results for both 88- and 127-bit computational security for our protocols, but anticipate the performance numbers for 127-bit computational security would drop significantly if processors with `AVX512` instructions become available.

Microbenchmarks. We first measured the performance of our protocol over seven basic tasks (see Fig. 7):

1. **Garble**: This includes generating three wires (i.e. the wire-labels, wire permutation messages, and their i-hashes) and a garbled truth table for an AND gate.
2. **Check**: This includes evaluating a garbled gate and verifying the results with the three revealed wire permutations bits.
3. **Evaluate**: This includes evaluating a garbled truth table.
4. **Solder**: This includes solder three wires of a garbled AND gate.
5. **P_1's Input**: This includes generating a fresh wire and sending the wire-label associated to an input bit of P_1.
6. **P_2's Input**: This includes generating 40 fresh wires without the wire permutation messages and running 40 extended OTs.
7. **Output**: This includes revealing the wire permutation messages of an output wire if P_2 should learn the output (or sending the output wire-label obtained from evaluation if P_1 should learn the output).

We have also compared the performance of the basic procedures between our approach and WMK's [50] (Fig. 8). While our speed of processing logical AND gates is about 2.5–5x slower, we can outperform WMK's highly optimized circuit input/output handling mechanism by 2–200x in LAN setting and 3–75 in the WAN setting, which demonstrates a clear advantage of LEGO approach over traditional cut-and-choose protocols. Note that as the communication cost becomes the bottleneck in the WAN setting, the performance gap of a task will approach the bandwidth requirement ratio of the task between the two protocols.

	P_1**'s Input**		P_2**'s Input**		**Output**		**Logical AND**	
	WMK	**Ours**	WMK	**Ours**	WMK	**Ours**	WMK	**Ours**
LAN	13.8	**0.57**	19.7	**8.24**	12.3	**0.02**	4.13	**21**
WAN	158.3	**3.8**	111.3	**36.4**	105.1	**1.4**	51.6	**135**

Fig. 8: Microbenchmark comparisons with WMK [50]. The units are either microsecond/wire or microsecond/gate. Timings are wall-clock time. Security parameters are aligned at $s = 40, k = 127$, and assuming $B = 5$ in our protocol.

Applications. Fig. 9 shows how our protocol compares to the baselines over several end-to-end oblivious applications running with 2^{-40} statistical security. These include

1. **AES.** It encrypts one block using AES. The circuit takes a 128-bit input from each party and computes $N = 6800$ AND gates. We set $B = 5$, $T = 39535$.
2. **DES.** It encrypts one block using DES. The circuit takes a 768-bit key from P_1 and a 64-bit message from P_2. Since $N = 18175$, we set $B = 5$, $T = 97593$.
3. **Comparison.** It compares two 10K-bit integers so it takes 10K bits from P_1 and 10K bits from P_2 and outputs 1 bit to indicate which number is larger. It involves $N = 10$K AND gates, so we set $B = 5$ and $T = 55973$.
4. **Hamming Distance.** This computes the Hamming distance between two 2048-bit strings. The circuit takes a 2048-bit input from each party and computes $N = 4087$ AND gates, so we set $B = 5$, $T = 25432$.

Our approach is more than 2x faster than TinyLEGO [44] in the LAN setting. Our measurements of their AES and DES protocols are in line with the numbers reported in their paper, though we noticed that theirs run much slower on applications with longer inputs and outputs such as Compare and Hamming (whose performance numbers were not included in their paper). We suspect (and get confirmed by one of the TinyLEGO implementers) that this is probably due to some implementation issues and the use of *input authenticators* mechanism required in TinyLEGO. On the flip side, we note that TinyLEGO uses 20–50% less bandwidth.

In comparison with WMK, our protocol is about 4–5x slower when running AES and DES over the LAN. However, for input/output intensive applications

like Compare and Hamming, it is not hard to see our overall performance can be very close or even 30–80% faster than WMK, especially in the WAN setting.

	AES		**DES**		**Compare**		**Hamming**	
	LAN	WAN	LAN	WAN	LAN	WAN	LAN	WAN
WMK [50]	39	580	66	1200	266	3479	70	946
TinyLEGO [44]	241	1055	561	1883	1345	3927	353	1459
Ours ($k = 88$)	**97**	**827**	**254**	**2182**	**170**	**1705**	**61.5**	**733**
Ours ($k = 127$)	**119**	**1060**	**300**	**2589**	**202**	**1921**	**75**	**858**

(a) Time. (Numbers are in millisecond. Timings are end-to-end wall-clock time excluding that for *one-time* setup work such as the base OTs.)

	AES	**DES**	**Compare**	**Hamming**
WMK [50]	10.2	27.2	78.9	18.8
TinyLEGO [44]	15.3	30.5	65.6	21.4
Ours ($k = 88$)	**22.8**	**55.6**	**84.7**	**25.2**
Ours ($k = 127$)	**26.5**	**64.6**	**89.8**	**27.6**

(b) Bandwidth. (Numbers are in MB.)

Fig. 9: Applications performance. (Measurements are averaged over 10 executions. The numbers don't include the setup cost, i.e., for the base OTs and ZK proof of Step 5b.)

References

1. Circuits of Basic Functions Suitable For MPC and FHE. http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/.
2. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide.
3. A. Afshar, Z. Hu, P. Mohassel, and M. Rosulek. How to efficiently evaluate ram programs with malicious security. *EUROCRYPT*, 2015.
4. A. Afshar, P. Mohassel, B. Pinkas, and B. Riva. Non-interactive secure computation based on cut-and-choose. *EUROCRYPT*, 2014.
5. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. *EUROCRYPT*, 2015.
6. B. Barak, Y. Dodis, H. Krawczyk, O. Pereira, K. Pietrzak, F.-X. Standaert, and Y. Yu. Leftover hash lemma, revisited. In *CRYPTO*, 2011.
7. M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy*, 2013.

8. M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Conference on Computer and Communications Security*, 2012.
9. L. T. A. N. Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique. *ASIACRYPT*, 2013.
10. J. Camenisch, G. Neven, and a. shelat. Simulatable adaptive oblivious transfer. In *EUROCRYPT*, 2007.
11. R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 2000.
12. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. *CRYPTO*. 2012.
13. C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: an efficient and scalable protocol. *CCS*, 2013.
14. T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. *EUROCRYPT*, 2013.
15. T. Frederiksen, T. Jakobsen, J. Nielsen, and R. Trifiletti. TinyLEGO: An interactive garbling scheme for maliciously secure two-party computation. *Cryptology ePrint Archive 2015/309*, 2015. http://eprint.iacr.org/2015/309.
16. T. Frederiksen, T. Jakobsen, J. Nielsen, and R. Trifiletti. On the complexity of additively homomorphic UC commitments. *TCC*, 2016.
17. T. K. Frederiksen, J. B. Nielsen, and C. Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In *EUROCRYPT*, 2015.
18. O. Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, 2004.
19. S. Gueron, Y. Lindell, A. Nof, and B. Pinkas. Fast garbling of circuits under standard assumptions. In *CCS*, 2015.
20. W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. *CCS*, 2010.
21. A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. *CCS*, 2012.
22. Y. Huang, D. Evans, and J. Katz. Private set intersection: are garbled circuits better than custom protocols? *NDSS*, 2012.
23. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. *USENIX Security Symposium*, 2011.
24. Y. Huang, J. Katz, V. Kolesnikov, R. Kumaresan, and A. Malozemoff. Amortizing garbled circuits. *CRYPTO*, 2014.
25. Y. Huang, L. Malka, D. Evans, and J. Katz. Efficient privacy-preserving biometric identification. *NDSS*, 2011.
26. R. Impagliazzo, L. A. Levin, and M. Luby. Pseudo-random generation from one-way functions (extended abstracts). In *STOC*, 1989.
27. M. Jawurek, F. Kerschbaum, and C. Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. *CCS*, 2013.
28. M. Keller, E. Orsini, and P. Scholl. Actively secure OT extension with optimal overhead. *CRYPTO*, 2015.
29. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, 2008.
30. B. Kreuter, B. Mood, A. Shelat, and K. Butler. PCF: A portable circuit format for scalable two-party secure computation. *USENIX Security Symposium*, 2013.
31. Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-Gate Secure Computation with Malicious Adversaries. In *USENIX Security Symposium*, 2012.

32. Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. *CRYPTO*, 2013.
33. Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. *EUROCRYPT*, 2007.
34. Y. Lindell and B. Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. *CRYPTO*, 2014.
35. Y. Lindell and B. Riva. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. *CCS*, 2015.
36. C. Liu, Y. Huang, E. Shi, J. Katz, and M. W. Hicks. Automating efficient RAM-model secure computation. *IEEE Symposium on Security and Privacy*, 2014.
37. C. Liu, X. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. *IEEE Symposium on Security and Privacy*, 2015.
38. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. *USENIX Security Symposium*, 2004.
39. X. Wang and A. Malozemoff and J. Katz. EMP-Toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.
40. K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel secure computation made easy. *IEEE Symposium on Security and Privacy*, 2015.
41. J. Nielsen, P. Nordholt, C. Orlandi, and S. Burra. A new approach to practical active-secure two-party computation. *CRYPTO*, 2012.
42. J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. *TCC*, 2009.
43. J. B. Nielsen and C. Orlandi. Cross and Clean: Amortized garbled circuits with constant overhead. *TCC*, 2016.
44. J. B. Nielsen, T. Schneider, and R. Trifiletti. Constant round maliciously secure 2pc with function-independent preprocessing using lego. *IACR Cryptology ePrint Archive*, 2016/1069, 2017.
45. V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. *IEEE Symposium on Security and Privacy*, 2013.
46. C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. *CRYPTO*, 2008.
47. B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure two-party computation is practical. *ASIACRYPT*, 2009.
48. P. Rindal and M. Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *USENIX Security Symposium*, 2016.
49. E. Songhori, S. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. *IEEE Symposium on Security and Privacy*, 2015.
50. X. Wang, A. J. Malozemoff, and J. Katz. Faster two-party computation secure against malicious adversaries in the single-execution setting. *EUROCRYPT*, 2017.
51. X. Wang, Y. Huang, T.-H. Chan, A. Shelat, and E. Shi. SCORAM: Oblivious RAM for secure computation. *CCS*, 2014.
52. X. Wang, Y. Huang, Y. Zhao, H. Tang, X. Wang, and D. Bu. Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. *CCS*, 2015.
53. A. C.-C. Yao. How to generate and exchange secrets (extended abstract). *FOCS*, 1986.

54. S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. *EUROCRYPT*, 2015.
55. R. Zhu, Y. Huang, and J. Katz. The cut-and-choose game and its application to cryptographic protocols. *USENIX Security Symposium*, 2016.

A Some Missing Proofs

A.1 Proof of Lemma 4.2

Assume there exists a nonempty set I such that

$$\boldsymbol{m}'_i \neq \mathsf{PRG}(seed_1, i) \| \ldots \| \mathsf{PRG}(seed_\ell, i), \text{ if } i \in I.$$

Hence, with y randomly picked from $\{0,1\}^\nu$ and a fixed t, the probability that $\sum_{i\in I} y_i \boldsymbol{m}'_i = \hat{\boldsymbol{m}'}_t - \sum_{i\in[\nu]/I} y_i \boldsymbol{m}'_i - \boldsymbol{m}'_t$ is $2^{-\sigma}$. That is, $\sum_{i\in I} y_i \boldsymbol{m}'_i + \sum_{i\in[\nu]/I} y_i \boldsymbol{m}'_i + \boldsymbol{m}'_t = \sum_{i\in[\nu]} y_i \boldsymbol{m}'_i + \boldsymbol{m}'_t = \hat{\boldsymbol{m}'}_t$ holds with probability $2^{-\sigma}$. By part 2 of Lemma 4.3,

$$\sum_{i\in[\nu]} y_i \langle \boldsymbol{m}'_i \rangle + \langle \boldsymbol{m}'_t \rangle = \left\langle \hat{\boldsymbol{m}'}_t \right\rangle \Longleftrightarrow \sum_{i\in[\nu]} y_i \boldsymbol{m}'_i + \boldsymbol{m}'_t = \hat{\boldsymbol{m}'}_t$$

except for $\binom{\ell-1}{w} / \binom{n}{w}$ probability. Because Step 2c checks $\sum_{i\in[\nu]} y_i \langle \boldsymbol{m}'_i \rangle + \langle \boldsymbol{m}'_t \rangle = \left\langle \hat{\boldsymbol{m}'}_t \right\rangle$, it can pass with probability at most $2^{-\sigma} + (1-2^{-\sigma}) \cdot \binom{\ell-1}{w} / \binom{n}{w}$. Repeating step 2c t times renders the probability passing all checks below $2^{-t\sigma} + (1-2^{-t\sigma}) \cdot \binom{\ell-1}{w} / \binom{n}{w}$. Since Step 2c repeats $\lceil \log \left(2^{-s} - \binom{\ell-1}{w} / \binom{n}{w} \right) / \sigma \rceil$ times, a bad $\boldsymbol{m}'$ can be i-hashed without triggering a failure with probability at most 2^{-s}. □

A.2 Proof of Lemma 4.3

1. Recall $\mathbf{H}_{min}(\boldsymbol{m}) = \ell \cdot \sigma$ and $\langle \boldsymbol{m} \rangle$ is the product of $\boldsymbol{m}$ and w linearly independent vectors. Thus, in P_2's view, for any fixed $\boldsymbol{m}_0$, $\Pr(\boldsymbol{m} = \boldsymbol{m}_0 | \langle \boldsymbol{m} \rangle = \langle \boldsymbol{m}_0 \rangle) = 2^{-(\ell-w)\sigma}$. Therefore, $\mathbf{H}_{min}(\boldsymbol{m} | \langle \boldsymbol{m} \rangle) = -\sum_{\boldsymbol{m}_0} \Pr[\boldsymbol{m} = \boldsymbol{m}_0 | \langle \boldsymbol{m} \rangle = \langle \boldsymbol{m}_0 \rangle] \cdot \log \Pr[\boldsymbol{m} = \boldsymbol{m}_0 | \langle \boldsymbol{m} \rangle = \langle \boldsymbol{m}_0 \rangle] = -\sum_{\boldsymbol{m}_0} 2^{-(\ell-w)\sigma} \cdot [-(\ell-w)\sigma] = 2^{-(\ell-w)\sigma} \cdot [(\ell-w)\sigma] \cdot 2^{(\ell-w)\sigma} = (\ell-w)\sigma$.
2. If $\boldsymbol{m} \neq \boldsymbol{m}'$, their $[n, \ell, n-\ell+1]_{2^\sigma}$ Reed-Solomon codewords can have at most $\ell - 1$ identical symbols. Since w (out of n) symbols of the encoding are randomly selected to watch, $\mathsf{IHash}_{\ell,\sigma}.\mathsf{Verify}(\langle \boldsymbol{m} \rangle, \boldsymbol{m}')$ will pass with probability at most $\binom{\ell-1}{w} / \binom{n}{w}$.

□

A.3 Proof of Lemma 5.1

A.4 Proof of Lemma 6.1

Proof. Following lemma 4.3, 40-bit statistical binding for both settings can be verified by computing $-\log\left[\binom{\ell_w-1}{w_w}\Big/\binom{n_w}{w_w}\right] = -\log\left[\binom{31}{21}/\binom{86}{21}\right] > 40$ and $-\log_2\left[\binom{\ell_w-1}{w_w}\Big/\binom{n_w}{w_w}\right] = -\log_2\left[\binom{47}{32}/\binom{88}{32}\right] > 40$.

Next, we examine the hiding aspect. Let $\boldsymbol{T} = (T_1, T_2, \cdots, T_{16})$ be an 32×16 matrix over $\mathrm{GF}(2^8)$ of rank 16, and $\boldsymbol{W} = (W_1, W_2, \cdots, W_{21})$ be an 32×21 matrix over $\mathrm{GF}(2^8)$ of rank 21. We want to show that for every adversary $\mathcal{A}$,

$$-\log \Pr\left(\boldsymbol{m} \leftarrow \mathrm{GF}(2^8)^{32} \ : \ \mathcal{A}(\boldsymbol{W}, \boldsymbol{T}, \boldsymbol{m}\cdot\boldsymbol{W}) = \boldsymbol{m}\cdot\boldsymbol{T}\right) > 87.999.$$

Define $D = \dim(\boldsymbol{T} \oplus \boldsymbol{W}) - \dim(\boldsymbol{W})$ where "$\oplus$" denotes *direct sum* and "$\dim(\cdot)$" denotes the *dimension* of a given vector space. For every rank-t matrix $\boldsymbol{T}$ and every rank-w matrix $\boldsymbol{W}$, we note that

$$\Pr\left(\boldsymbol{m} \leftarrow \mathrm{GF}(2^8)^{32} \ : \ \mathcal{A}(\boldsymbol{W}, \boldsymbol{T}, \boldsymbol{m}\cdot\boldsymbol{W}) = \boldsymbol{m}\cdot\boldsymbol{T}\right) = 2^{-8D}.$$

Thus, our goal is to show that $-\log \mathbb{E}(2^{-8D}) > 87.999$. The concept of *expectation* is introduced because 2^{-8D} itself is a random variable over the random choices of picking $\boldsymbol{T}$ and $\boldsymbol{W}$.

Define $d_{i,j} = \dim(\mathrm{Span}(T_{i+1}, T_{i+2}, \cdots, T_t, \boldsymbol{W})) - \dim(\boldsymbol{W})$ under the condition that $\dim(\mathrm{Span}(T_1, T_2, \cdots, T_i) \cap \boldsymbol{W}) = j$. Thus, $D = d_{0,0}$ and we can derive $d_{i,j}$ from $d_{i+1,j+1}$ and $d_{i+1,j}$ using recursion. Vector T_{i+1} has to fall into one of the two cases:

1. $T_{i+1} \in \boldsymbol{W}$: Thus $\mathrm{Span}(T_{i+1}, T_{i+2}, \cdots, T_t, \boldsymbol{W}) = \mathrm{Span}(T_{i+2}, T_{i+3}, \cdots, T_t, \boldsymbol{W})$. In addition, because $T_{i+1} \notin \mathrm{Span}(T_1, T_2, \cdots, T_i)$ and $\mathrm{Span}(T_1, T_2, \cdots, T_{i+1}) \cap \boldsymbol{W} = (\mathrm{Span}(T_1, T_2, \cdots, T_i) \cap \boldsymbol{W}) \oplus \mathit{Span}(T_{i+1})$, we know,

$$\begin{aligned}\dim(\mathrm{Span}(T_1, T_2, \cdots, T_{i+1}) \cap \boldsymbol{W}) &= \dim(\mathrm{Span}(T_1, T_2, \cdots, T_i) \cap \boldsymbol{W}) + 1 \\ &= j + 1\end{aligned}$$

   Note the probability of $T_{i+1} \in \boldsymbol{W}$, conditioned on $T_{i+1} \notin \mathrm{Span}(T_1, \ldots, T_i)$, is $(2^{8\cdot(21-j)} - 1)/(2^{8\cdot(32-i)} - 1)$. This is because there are $2^{8\cdot(32-i)} - 1$ possible non-zero T_{i+1} that satisfy $T_{i+1} \notin \mathrm{Span}(T_1, \ldots, T_i)$; and $2^{8\cdot(21-j)} - 1$ non-zero choices of T_{i+1} such that $T_{i+1} \in \boldsymbol{W}$ but $T_{i+1} \notin \mathrm{Span}(T_1, \ldots, T_i) \cap \boldsymbol{W}$, since $\dim(\mathrm{Span}(T_1, \ldots, T_i) \cap \boldsymbol{W}) = j$.
2. $T_{i+1} \notin \boldsymbol{W}$: Thus $\mathrm{Span}(T_{i+1}, T_{i+2}, \cdots, T_t, \boldsymbol{W}) = \mathrm{Span}(T_{i+2}, T_{i+3}, \cdots, T_t, W) \oplus \mathrm{Span}(T_{i+1})$, and

$$\dim(\mathrm{Span}(T_1, T_2, \cdots, T_{i+1}) \cap \boldsymbol{W}) = \dim(\mathrm{Span}(T_1, T_2, \cdots, T_i) \cap \boldsymbol{W}) = j.$$

   This happens with probability $1 - (2^{8\cdot(21-j)} - 1)/(2^{8\cdot(32-i)} - 1)$.

Therefore,

$$d_{i,j} = \frac{2^{8\cdot(21-j)}-1}{2^{8\cdot(32-i)}-1} d_{i+1,j+1} + \left(1 - \frac{2^{8\cdot(21-j)}-1}{2^{8\cdot(32-i)}-1}\right)(d_{i+1,j}+1)$$

Moreover,

$$\begin{aligned}\mathbb{E}(2^{-8\cdot d_{i,j}}) =& \frac{2^{8\cdot(21-j)}-1}{2^{8\cdot(32-i)}-1}\mathbb{E}(2^{-8\cdot d_{i+1,j+1}}) + \left(1 - \frac{2^{8\cdot(21-j)}-1}{2^{8\cdot(32-i)}-1}\right)\mathbb{E}\left(2^{-8\cdot(d_{i+1,j}+1)}\right) \\ =& \frac{2^{8\cdot(21-j)}-1}{2^{8\cdot(32-i)}-1}\mathbb{E}(2^{8\cdot d_{i+1,j+1}}) + \left(1 - \frac{2^{8\cdot(21-j)}-1}{2^{8\cdot(32-i)}-1}\right)\frac{1}{2^8}\mathbb{E}(2^{-8\cdot d_{i+1,j}})\end{aligned}$$

Finally, the base cases for bootstrapping the recursive calculation are: (1) $d_{16,j} = 0$ for all j; and (2) $d_{i,21} = i$ for all i. It is easy to calculate $d_{0,0}$ in full precision with a computer program. Thus,

$$\begin{aligned}-\log_2 \mathbb{E}(2^{-8\cdot D}) =& \log_2 \mathbb{E}(2^{8\cdot d_{0,0}}) \\ =& \log_2 \frac{340282366920938463463374607431768211457}{1099511627777} \\ >& 87.9999999999986 > 88 - 10^{-11}.\end{aligned}$$

When setting $n_w = 88, \ell_w = 48, w_w = 32$ and $\sigma_w = 8$, we can derive a similar recurrence as above and solve it for a different $d_{0,0}$ and verify that $\log_2 \mathbb{E}(2^{8\cdot d_{0,0}}) > 127$. □

SUPPLEMENTARY MATERIAL

B Proof of Theorem 6.2

Here we generalize the bucket size B to decimals, implying buckets can be of several different integer sizes. Lemmas referenced in this section can be found in the supplemental materials.

Proof of Theorem 6.2 Part 1: We first show the theorem for the case $\kappa = 2$. The validity of Theorem 1 for $\kappa = 2$ can be established based on Lemma B.1 and B.2:

1. According to Lemma B.1, the protocol cannot be secure if the buckets are of sizes 1 or 2;
2. According to Lemma B.2, fixing $\kappa = 2$ and T, N, using buckets of size greater than 2 will only make it even more vulnerable to attacks.

For the case of $\kappa < 2$, we know that, compared to the case when $\kappa = 2$, T has to be reduced (for any fixed N). This implies one of the following three has to be true:

1. less garbled gates can be used for verification;
2. less garbled gates be used for evaluation;
3. both the above two happen.

Since any of above three facts will make it easier for the cheating P_1 to succeed, Part 1 of Theorem 6.2 holds for $\kappa < 2$ too.

Proof of Theorem 6.2 Part 2: The proof is the same as that of Theorem 5.4 except that the use of Lemma 5.1 and Lemma 5.2 are replaced by Lemma B.6. □

Remark. The conclusion that "$\kappa \leq 2$ can't be secure" is meaningful only in the *asymptotic* sense. A $\kappa \leq 2$ could still offer certain *concrete* security, e.g., a $\kappa \leq 2$ could provide 3 bits of statistical security. However, our proof shows that a $\kappa \leq 2$ cannot provide statistical security of more than 5 bits for any realistic application that needs more than three AND buckets. To see this, simply set $N_0 = 3$ and plugging in $c_2 = 0.48, c_3 = 0.13, c_1 = 0.85$ (so $c > 0.05$, that is, $\log 0.05^{-1} \approx 4.32$ bits of security at best) in the proof of Claim B.5.

Lemma B.1 *Assume $B \leq 2$. Let $\Pr_{overall}(N, x, b)$ be the probability that P_1, with b bad gates, succeeds in attacking a protocol computing a circuit of N buckets, of which x fraction is size 2 and $(1 - x)$ fraction is size 1. If $\kappa = 2$, there exists a constant $c > 0$ and an integer N_0 such that $\Pr_{overall}(N, x, b) > c$ for all $N > N_0$.*

Proof. Assume for simplicity that in the gate verification stage, if a faulty gate is indeed selected for verification, P_2 is able to detect it with probability 1, i.e., $\tau = 1$. (If $\tau < 1$, it is easier for P_1 to succeed.)

Let $\Pr_c(N,x,b)$ be the probability that P_1 survives the gate verification stage; and $\Pr_e(N,x,b)$ be the probability that P_1 succeeds in the evaluation stage given that it passes the verification stage. Because $\tau = 1$, in a successful attack, no bad gates are "consumed" in the verification stage. Therefore, there exists a positive constant c and a constant N_0 such that for all $N > N_0$,

$$\begin{aligned}
&\Pr_{overall}(N,x,b) = \Pr_c(N,x,b) \cdot \Pr_e(N,x,b) \\
\geq &\left(\frac{1+x}{2}\right)^b \left[1 - \frac{(1-x)b^2}{(1+x)(2N-b)}\right] \cdot \Pr_e(N,x,b) && \text{[Claim B.3]} \\
\geq &\left(\frac{1+x}{2}\right)^b \left[1 - \frac{(1-x)b^2}{(1+x)(2N-b)}\right] \left(1 - \left(\frac{2x}{1+x}\right)^b\right) && \text{[Claim B.4]} \\
> &c && \text{[Claim B.5]}
\end{aligned}$$

This completes the proof. □

Lemma B.2 *Fixing T, N and $\kappa = 2$, setting $B \leq 2$ leads to lower successful attacks rates than setting $B > 2$.*

Proof. First, we show that, if $B > 2$, a scheme S_1 that used a size-1 bucket and a size-B bucket performs no better (in terms of preventing attacks to cut-and-choose) than a scheme S_2 that replaces the two buckets by a size-2 bucket and a size-$(B-1)$ bucket (note this change preserves the total numbers of garbled gates and the only difference between S_1 and S_2 is this pair of buckets). We can derive this fact from counting the number of arrangements of b' bad gates over this pair of buckets that *foil* attacks to cut-and-choose:

1. If $b' < B-1$, S_1 has $\binom{B}{b'}$ foiling arrangements (as long as no bad gates go to the size-1 bucket) while S_2 has $\binom{B-1}{b'} + \binom{2}{1}\binom{B-1}{b'-1}$ (the bad gates either all go to the size-$(B-1)$ bucket, or only $b'-1$ of them go to the size-$(B-1)$ bucket and the rest goes to the size-2 bucket). Because $\binom{B-1}{b'} + \binom{2}{1}\binom{B-1}{b'-1} = \binom{B}{b'} + \binom{B-1}{b'-1} > \binom{B}{b'}$, S_2 works better.
2. If $b' = B-1$, the count is B for S_1 and $2(B-1)$ for S_2. Because $2(B-1) > B$, S_2 works better than S_1.
3. If $b' > B-1$, the counts are 0 for both S_1 and S_2. Hence, S_2 works no worse than S_1.

Through recursively applying the argument above, we always end up with a scheme that either uses (1) only buckets of size 2 or above, which implies $\kappa > 2$ because at least 1 gate needs to be used for verification, hence contracting the assumption $\kappa = 2$; Or (2) only buckets of size 1 and 2. □

Claim B.3 *Fix $\kappa = 2$ and N. Let x fraction of the buckets are of size 2 and the rest $(1-x)$ are of size 1. Let $\Pr_c(N,x,b)$ be the probability that P_1 survives the gate verification stage with b bad gates. Then*

$$\Pr_c(N,x,b) \geq \left(\frac{1+x}{2}\right)^b \left[1 - \frac{(1-x)b^2}{(1+x)(2N-b)}\right].$$

Proof. Since a total of $(1+x)N$ garbled gates are used in evaluation while the rest $T-(1+x)N$ gates are used for checking, we have

$$\begin{aligned}
&\Pr_c(N,x,b) = \binom{T-b}{T-(1+x)N} \Big/ \binom{T}{T-(1+x)N} \\
=&\binom{2N-b}{2N-(1+x)N} \Big/ \binom{2N}{2N-(1+x)N} && (3)\\
=&\frac{[(1+x)N]}{2N} \cdot \cdots \cdot \frac{[(1+x)N-b+1]}{[2N-b+1]} \geq \left[\frac{(1+x)N-b}{2N-b}\right]^b && (4)\\
=&\left(\frac{1+x}{2}\right)^b \left(1-\frac{(1-x)b}{(1+x)(2N-b)}\right)^b \geq \left(\frac{1+x}{2}\right)^b \left(1-\frac{(1-x)b^2}{(1+x)(2N-b)}\right) && (5)
\end{aligned}$$

where equality (3) holds because $\kappa = T/N = 2$; the inequality (4) holds because every of the b fractions is larger than or equal to $[(1+x)N-b]/(2N-b)$; and (5) can be derived from the binomial inequality (i.e., $\forall x \in \mathbb{R}, x > -1$, and $\forall n \in \mathbb{N}, (1+x)^n \geq 1+nx$) and the fact that $\dfrac{(1-x)b}{(1+x)(2N-b)} < 1$ when $0 \leq b \leq \kappa N = 2N$, $\dfrac{1}{N} \leq x \leq 1-\dfrac{1}{N}$. □

Claim B.4 *Fix N. Let x represents the fraction of the buckets are of size 2, so the rest $(1-x)$ are of size 1. Let $\Pr_e(N,x,b)$ be the probability of P_1 successfully cheats in the evaluation stage, with b bad evaluation gates. Then $\Pr_e(N,x,b) \geq 1-\left(\frac{2x}{1+x}\right)^b$.*

Proof. P_1's attack fails if every bucket has at least one good gate (achievable using our proposed protocol in Section 5). Throwing $(1+x)N$ gates (b of which is bad) into N buckets, with probability $2^b\binom{xN}{b}/\binom{(1+x)N}{b}$ every bucket will contain at least one good gate as $2^b\binom{xN}{b}$ is the number of ways to place b bad gates into the xN size-2 buckets subject to at most one bad gate per bucket, and $\binom{(1+x)N}{b}$ is the total number of ways to group all gates without any restriction. Therefore,

$$\Pr_e(N,x,b) = 1-\frac{2^b\binom{xN}{b}}{\binom{(1+x)N}{b}} = 1-\frac{2^b[xN-(b-1)][xN-(b-2)]\cdots[xN]}{[(1+x)N-(b-1)][(1+x)N-(b-2)]\cdots[(1+x)N]} \geq 1-\left(\frac{2x}{1+x}\right)^b$$

where the inequality holds because every of the b fractions is greater than or equal to $(2x)/(1+x)$. □

Claim B.5 *There exists a constant $c > 0$ and a constant N_0 such that for all integer $N > N_0, \forall x \in [\frac{1}{N}, 1-\frac{1}{N}]$, there exists a positive integer b such that*

$$\left(\frac{1+x}{2}\right)^b \left(1-\frac{(1-x)b^2}{(1+x)(2N-b)}\right)\left(1-\frac{2x}{1+x}\right)^b > c.$$

Proof. It suffices to show that there exists $c_1, c_2, c_3 > 0$ and $N_0 > 0$ such that $\forall N > N_0, \forall x \in [1/N, 1-1/N]$, there exists a positive integer b that satisfies all of the three inequality below,

$$[(1+x)/2]^b > c_1 \tag{6}$$

$$1-(1-x)b^2/[(1+x)(2N-b)] > c_2 \tag{7}$$

$$1-[2x/(1+x)]^b > c_3 \tag{8}$$

Because (6) holds as long as $b < \log c_1 \,/\log \frac{1+x}{2}$, (7) holds if

$$b < \sqrt{2N\frac{1+x}{1-x}(1-c_2)+\frac{1}{4}\left(\frac{1+x}{1-x}\right)^2(1-c_2)^2}-\frac{1}{2}\cdot\frac{1+x}{1-x}\cdot(1-c_2),$$

(8) holds as long as $b > \log(1-c_3)/\log\frac{2x}{1+x}$, and b needs to be a positive integer, it suffices to show that there exist positive c_1, c_2, c_3, and N_0 such that the following two inequalities hold for all $N > N_0$,

$$\frac{\log(1-c_3)}{\log\frac{2x}{1+x}}+1 < \frac{\log c_1}{\log\frac{1+x}{2}} \tag{9}$$

$$\frac{\log(1-c_3)}{\log\frac{2x}{1+x}}+1 < -\frac{1}{2}\cdot\frac{1+x}{1-x}\cdot(1-c_2)+\sqrt{2N\frac{1+x}{1-x}(1-c_2)+\frac{1}{4}\cdot\left(\frac{1+x}{1-x}\right)^2(1-c_2)^2} \tag{10}$$

We note that (9) is equivalent to

$$\frac{1}{\log c_1}\log\frac{1+x}{2x} > \frac{1}{\log(1-c_3)}\log\frac{2}{1+x}+\log\frac{1+x}{2x}\log\frac{2}{1+x},$$

which will always hold as long as

$$\frac{1}{\log c_1}-\frac{1}{\log(1-c_3)}-\log 2 > 0 \tag{11}$$

because $\frac{1+x}{2x} > \frac{2}{1+x}$ and $\log\frac{2}{1+x} < \log 2$ hold for all $x \in [1/N, 1-1/N]$. Since (11) doesn't involve x, it is easy to find a c_1 based on the value of c_3 such that (11) is satisfied.

Next, we note that (10) is equivalent to

$$\frac{2N\frac{1+x}{1-x}(1-c_2)}{\sqrt{2N\frac{1+x}{1-x}(1-c_2)+\frac{1}{4}(\frac{1+x}{1-x})^2(1-c_2)^2}+\frac{1}{2}\cdot\frac{1+x}{1-x}\cdot(1-c_2)} > 1+\frac{\log\frac{1}{1-c_3}}{\log\frac{1+x}{2x}},$$

which can be simplified, by defining $c_2' = 1-c_2$ and $y = (1+x)/(1-x)$, to $\left(\frac{2N}{\sqrt{\frac{2N}{yc_2'}+\frac{1}{4}}+\frac{1}{2}}-1\right)\log\left(1+\frac{1}{y-1}\right) > \log\frac{1}{1-c_3}$. Now we analyze this inequality in two cases:

Case I ($y < 3$)**:** If $y < 3$,

$$\left(\frac{2N}{\sqrt{\frac{2N}{yc_2'}+\frac{1}{4}}+\frac{1}{2}}-1\right)\log\left(1+\frac{1}{y-1}\right) \geq \left(\frac{2N}{\sqrt{\frac{2N}{c_2'}+\frac{1}{4}}+\frac{1}{2}}-1\right)\log\left(1+\frac{1}{y-1}\right)$$

$$\geq (\frac{2N}{\sqrt{\frac{2N}{c_2'}+1}}-1)\log(1+\frac{1}{y-1})$$
$$\geq \left(\frac{2N}{\sqrt{\frac{2N}{c_2'}+1}}-1\right)\log\frac{3}{2}.$$

Because there exists an integer N_0 such that for all $N > N_0$,

$$\left(\frac{2N}{\sqrt{\frac{2N}{c_2'}+1}}-1\right)\log\frac{3}{2} > \log\frac{1}{1-c_3} \tag{12}$$

(10) can also be satisfied when $N > N_0$, regardless of the values of c_2' and c_3.
Case II ($y \geq 3$): If $y \geq 3$, then $0 < 1/(y-1) < 1/2$, and (because $y = \log(1+x)$ is concave function when $x \in [0, 1/2]$) we have $\log[1+1/(y-1)] \geq [2\log(3/2)]/(y-1) > [2\log(3/2)]/y$. In addition, for all $N > N_0$ (where N_0 is defined as above), we have $\frac{2N}{\sqrt{\frac{2N}{yc_2'}+\frac{1}{4}}+\frac{1}{2}} - 1 > \frac{2N}{\sqrt{\frac{2N}{c_2'}+1}} - 1 > 0$. Thus, for all $N > N_0$, we know

$$\left(\frac{2N}{\sqrt{\frac{2N}{yc_2'}+\frac{1}{4}}+\frac{1}{2}}-1\right)\log\left(1+\frac{1}{y-1}\right) \geq 2\log(3/2)\left(\frac{2N}{\sqrt{\frac{2Ny}{c_2'}+\frac{1}{4}y^2}+\frac{1}{2}y}-\frac{1}{y}\right)$$
$$\geq 2\log(3/2)\left(\frac{2N}{\sqrt{\frac{4N^2}{c_2'}+N^2}+N}-\frac{1}{3}\right) = 2\log(3/2)\left(\frac{2}{1+\sqrt{\frac{4}{c_2'}+1}}-\frac{1}{3}\right),$$

where the second inequality holds because $3 \leq y \leq 2N$. Therefore, we can find c_3 based on (the value of) c_2' to satisfy

$$2\log(3/2)\left(\frac{2}{1+\sqrt{\frac{4}{c_2'}+1}}-\frac{1}{3}\right) > \log\frac{1}{1-c_3}, \tag{13}$$

which will guarantee (10) holds.

To sum up, we have shown that if we pick an arbitrary positive number c_2 then find c_3 based on (13) and c_2, find c_1 based on (11) and c_3, and finally find N_0 based on (12) and c_2, c_3, then for all $N > N_0$, all three inequalities, (6), (7), (8) should hold. This completes the proof. □

Lemma B.6 *Fix $B = 2$. Let $\Pr_{overall}(N, b)$ be the probability that a malicious P_1 succeeds when generating b bad gates. For any $\kappa > 2$ and any $\varepsilon > 0$, there exists N_0 such that*

$$\Pr_{overall}(N, b) < \varepsilon, \quad (\forall\, N > N_0)$$

Proof. Let $0 < \tau \leq 1$ be the probability that P_2 detects that g is faulty (through checking) conditioned that g is indeed faulty. We have

$$\Pr_{overall}(N,b) = \sum_{i=0}^{b}(1-\tau)^i \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} \Pr_e(N, b-i)$$

where $(1-\tau)^i \binom{b}{i}\binom{T-b}{T-2N-i} \Big/ \binom{T}{T-2N}$ is the probability that P_1, with b bad gates initially, surviving the checking stage losing i bad gates (due to verification, while P_2 detecting none of them). Because $\exists i_0$ such that $(1-\tau)^{i_0} < \varepsilon/2$,

$$\begin{aligned}
&\Pr_{overall}(N,b)\\
=&\sum_{i=0}^{b}(1-\tau)^i \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} \Pr_e(N, b-i)\\
=&\sum_{i=0}^{i_0}(1-\tau)^i \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} \Pr_e(N, b-i) + \sum_{i=i_0+1}^{b}(1-\tau)^i \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} \Pr_e(N, b-i)\\
\leq&\sum_{i=0}^{i_0}(1-\tau)^i \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} \Pr_e(N, b-i) + (1-\tau)^{i_0} \sum_{i=i_0+1}^{b} \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} \Pr_e(N, b-i)\\
\leq&\sum_{i=0}^{i_0}(1-\tau)^i \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} \Pr_e(N, b-i) + \frac{\varepsilon}{2} \sum_{i=i_0+1}^{b} \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}}\\
\leq&\sum_{i=0}^{i_0}(1-\tau)^i \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} \Pr_e(N, b-i) + \frac{\varepsilon}{2} \sum_{i=1}^{b} \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}}\\
\leq&\sum_{i=0}^{i_0}(1-\tau)^i \frac{\binom{b}{i}\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} \Pr_e(N, b-i) + \frac{\varepsilon}{2} \cdot 1\\
\leq&\sum_{i=0}^{i_0}(1-\tau)^i \binom{b}{i} \left(\frac{T-2N}{T}\right)^i \left(\frac{2N}{T-i}\right)^{b-i} \Pr_e(N, b-i) + \frac{\varepsilon}{2} \qquad \text{[Claim B.8]}\\
\leq&\sum_{i=0}^{i_0}(1-\tau)^i \binom{b}{i} \left(\frac{T-2N}{T}\right)^i \left(\frac{2N}{T-i}\right)^{b-i} \Pr_e(N, b) + \frac{\varepsilon}{2}\\
\leq&\sum_{i=0}^{i_0}(1-\tau)^i \binom{b}{i} \left(\frac{T-2N}{T}\right)^i \left(\frac{2N}{T-i_0}\right)^{b-i} \Pr_e(N, b) + \frac{\varepsilon}{2}\\
\leq&\sum_{i=0}^{b}(1-\tau)^i \binom{b}{i} \left(\frac{T-2N}{T}\right)^i \left(\frac{2N}{T-i_0}\right)^{b-i} \Pr_e(N, b) + \frac{\varepsilon}{2}\\
=&\left((1-\tau)\frac{T-2N}{T} + \frac{2N}{T-i_0}\right)^b \Pr_e(N, b) + \frac{\varepsilon}{2}.
\end{aligned}$$

holds for any $N > N_1$ (where N_1 is determined according to the proof of Claim B.7).

Since $T = \kappa N$, we have

$$\begin{aligned}&\lim_{N\to\infty} \frac{(1-\tau)(T-2N)}{T} + \frac{2N}{T-i_0}\\ =&\lim_{N\to\infty} \frac{(1-\tau)(\kappa-2)}{\kappa} + \frac{2}{\kappa - i_0/N} = 1 - \frac{\tau(\kappa-2)}{\kappa} < 1.\end{aligned}$$

Therefore, there exists N_1 such that for all $N > N_1$, $(1-\tau)(T-2N)/T + 2N/(T-i_0) < 1$. Hence, for every $\varepsilon > 0$, we can find a b_0 such that,

1. for all $b > b_0$,

$$\begin{aligned}&\Pr_{overall}(N,b)\\ \le&\big[(1-\tau)(T-2N)/T + 2N/(T-i_0)\big]^b \Pr_e(N,b) + \varepsilon/2\\ <&\varepsilon \Pr_e(N,b)/2 + \varepsilon/2 < \varepsilon/2 + \varepsilon/2 = \varepsilon.\end{aligned}$$

2. for all $b \le b_0$, Claim B.7 shows how to further find an integer N_2 such that for all $N > N_2$,

$$\begin{aligned}&\Pr_{overall}(N,b)\\ \le&\big[(1-\tau)(T-2N)/T + 2N/(T-i_0)\big]^b \Pr_e(N,b) + \varepsilon/2\\ <&\big[1-\tau(\kappa-2)/\kappa\big]^b \varepsilon/2 + \varepsilon/2 < \varepsilon/2 + \varepsilon/2 = \varepsilon.\end{aligned}$$

Thus, setting $N_0 = \max(N_1, N_2)$ completes the proof. □

Lemma B.7 *Let* $\Pr_e(N,b)$ *be the probability that a malicious* P_1 *who survives the gate checking stage succeeds with* b *bad gates selected for evaluation. Then*

1. *For any fixed* N, $\Pr_e(N,b)$ *strictly* increases *with* b.
2. *For any* b *and* $\varepsilon > 0$, $\exists N_0$ *such that* $\Pr_e(N,b) < \varepsilon$.

Proof. Since all buckets are of size 2, the following can be derived similarly to the proof of Claim B.4 (by setting $x = 1$),

$$\begin{aligned}\Pr_e(N,b) &= 1 - \frac{2^b\binom{N}{b}}{\binom{2N}{b}} = 1 - \frac{2N-2(b-1)}{2N-(b-1)} \cdot \cdots \cdot \frac{2N}{2N} \qquad (14)\\ &\le 1 - \left(\frac{2N-2b+2}{2N-b+1}\right)^b = 1 - \left(\frac{2-2(b-1)/N}{2-(b-1)/N}\right)^b\end{aligned}$$

First, note that (14) implies that $\Pr_e(N,b)$ increases strictly with b, because the greater b is, the more multiplicative fractions (all smaller than 1) are in the product.

Second, note that for any b, ε, $\left(\frac{2N-2b+2}{2N-b+1}\right)^b$ goes arbitrarily close to 1 when N is sufficiently large. Therefore, $\forall b, \varepsilon$, $\exists N_1$ such that for all $N > N_1$, $\Pr_e(N,b) < \varepsilon$. □

Lemma B.8 *If T, N, b, i are non-negative integers such that $T > 2N$, $T \geq b$, and $i \leq b$, then $\frac{\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} \leq \left(\frac{T-2N}{T}\right)^i \left(\frac{2N}{T-i}\right)^{b-i}$.*

Proof.

$$\begin{aligned}
\frac{\binom{T-b}{T-2N-i}}{\binom{T}{T-2N}} =& \frac{(T-b)!(T-2N)!(2N)!}{T!(T-2N-i)!(2N-b+i)!} \\
=& \frac{\big[(T-2N-i+1)\cdots(T-2N)\big]\big[(2N-b+i+1)\cdots 2N)\big]}{(T-b+1)(T-b+2)\cdots T} \\
=& \frac{(T-2N-i+1)\cdots(T-2N)}{(T-i+1)\cdots T} \cdot \frac{(2N-b+i+1)\cdots 2N}{(T-b+1)\cdots(T-i)} \\
\leq& \left(\frac{T-2N}{T}\right)^i \left(\frac{2N}{T-i}\right)^{b-i}
\end{aligned}$$

□