Pool: Scalable On-Demand Secure Computation Service Against Malicious Adversaries

Ruiyu Zhu
Indiana University
zhu52@indiana.edu

Yan Huang
Indiana University
yh33@indiana.edu

Darion Cassel*
Carnegie Mellon University
darionc@andrew.cmu.edu

ABSTRACT

This paper considers the problem of running a long-term on-demand service for executing actively-secure computations. We examined state-of-the-art tools and implementations for actively-secure computation and identified a set of key features indispensable to offer meaningful service like this. Since no satisfactory tools exist for the purpose, we developed Pool, a new tool for building and executing actively-secure computation protocols at extreme scales with nearly zero offline delay. With Pool, we are able to obliviously execute, for the first time, reactive computations like ORAM in the malicious threat model. Many technical benefits of Pool can be attributed to the concept of *pool*-based cut-and-choose. We show with experiments that this idea has significantly improved the scalability and usability of JIMU [38], a state-of-the-art LEGO protocol.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**;

KEYWORDS

scalable actively-secure computation

1 INTRODUCTION

Secure computation has long been speculated to be a key technology for safely utilizing sensitive data owned by two or more distrustful parties. Towards this goal, a number of theoretical and implementational breakthroughs have significantly advanced the practicality of secure computation. In the honest-but-curious model, convenient programming tools [20, 28, 34] have enabled not only benchmark applications such as AES and PSI, but also a range of challenging applications with complex logic [31, 37], or handling large-scale sensitive data [7, 23, 32]. Recent progresses have shown that, with surprisingly small added cost, these protocols can be executed even in presence of active adversaries [6, 19, 25, 27, 35], at hundreds of thousands logical-gates per second.

Following this exciting trend, we consider the problem of running actively-secure computation protocols as an on-demand service between two "cautious" collaborating parties. Suppose two standing servers are established that receive an *everlasting* sequence of *dynamically*-supplied requests,

$$(f_1, x_1, y_1), \quad (f_2, x_2, y_2), \quad \ldots,$$

then securely compute them on-the-fly, and finally return the results

$$z_1 \coloneqq f_1(x_1, y_1), \quad z_2 \coloneqq f_2(x_2, y_2), \quad \ldots$$

to the designated output receiver. Example scenarios of this service can be that two credit card issuers collaboratively mine their ever-growing databases of personal transactions using a secure computation protocol to better identify credit card frauds; or that two medical research institutions conduct secure cross-database queries over their sensitive medical records, which potentially involves private computations in the RAM-model.

First, we expect the security guarantee withholds across the *lifetime* of the everlasting secure computation service. Second, like many other computing services, it would be natural and vital for the service to be *scalable*. Here, good scalability implies being able to efficiently handle functions that would (1) involve a large number of inputs/outputs, (2) use arbitrarily many gates, and (3) need to be dynamically-defined on-the-fly, and (4) be reactive sense like privately indexing a RAM. For the service to prosper, it is also expected to offer convenient programming interfaces to allow non-crypto-expert application developers to create innovative applications to utilize the cryptographic marvels. We summarize a list of valuable features for realizing such a service in Table 1.

Unfortunately, after a closer examination of existing tools in this domain, we discover that none of them is satisfactory for running such a service. For instance, the WMK protocol recently developed by Wang et al. [35] offers very efficient gate execution and provides good programming support through emp-toolkit [34]. However, it couldn't efficiently handle certain reactive computations, such as RAM-based computation, against active adversaries (we will discuss reactive computations in more detail in Section 2.2). In the offline/online setting, protocols by Rindal-Rosulek [27] and Lindell-Riva [19] enjoy very short amortized time but require knowing the target function well in advance, in addition to requiring a substantial offline processing stage. It is neither clear how they could be practically applied to RAM-based computations. Recent works of JIMU [38] and NST [25] have revitalized much interest in practical LEGO protocols. Nevertheless, applied naïvely, these protocols will incur a prohibitive amount of time and memory in offline processing, thus do not scale well to large computations. Moreover, existing BatchedCut (a technique that batches the cut-and-choose procedures across many computation instances) protocols [19, 25, 27, 38] require carefully selecting some protocol

*Work done as an REU intern of Indiana University.

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134070

Table 1: Comparison of Representative Implementations for Actively-Secure Computation Protocols.

	Secure over Infinite Executions	Efficient gate processing	Efficient input/output processing	Short offline delay for big circuits	Reactive computations[1] (e.g., ORAM)	Memory-efficient Scaling	Blackbox programming APIs
KSS [17]		✓		✓		✓	
WMK [35]		✓		✓			✓
JIMU [38] NST [25]			✓		✓		
LR [19] RR [27]		✓					
WRK [36]		✓	✓		✓		
This Work	✓	✓	✓	✓	✓	✓	✓

[1] We discuss reactive computations in more detail in Section 2.2.

parameters (such as bucket size and check rate) based on circuit size and security requirement (e.g. 2^{-40} statistical security), but did not provide a general, systematic procedure that can efficiently scale up to arbitrary circuit-sizes and security requirements. Last but most importantly, all existing protocols only guarantee security for executing individual or a predefined (finite) number of instances, hence are bound to fail if the service keeps running indefinitely. Table 1 compares the state-of-the-art actively-secure computation protocols.

In this work, we aim at building an actively-secure computation framework that offers all those desirable features discussed above. We adopt the same rationale behind the works on amortizing secure computations, but go one step further by envisioning running actively secure computations as an on-demand long-term service between two mutually-distrusted organizations. Our starting point is a BatchedCut protocol such as [36, 38]. But instead of always beginning with garbling "sufficiently" many gates for a predefined function f and then depleting all garbled-gates in the end, we opt to always maintain a pool of garbled entries and always do cut-and-choose within the pool. This seemingly simple idea allows us to reap most of the benefit of BatchedCut without periodically suffering from long delays and huge storage demands due to offline processing, thus promising to offer a more secure, reliable, and consistent service using moderate hardware resources.

1.1 Contribution

New Techniques. We propose a *pool* technique to efficiently run batched cut-and-choose protocols at an unprecedented scale. As a result, we are able to achieve competitive online efficiency with nearly zero offline cost per execution. A security advantage of our approach is that, without much extra costs, it carries over the statistical security guarantee of secure computation from a single instance to *infinitely* many computation instances. We formally analyze the security of the pool mechanism and give an efficient algorithm to automatically identify the best parameters for running pool-based cut-and-choose protocols. The main price of our scheme is to build and maintain a pool, though our experiments show that building and maintaining the pool is inexpensive in practice (see Table 3) and the costs can be amortized over *infinitely* many executions. Our approach can be applied to two underlying protocols, JIMU [38] and WRK [36].

We also propose a more efficient way to realize actively-secure multiplexers (`MUX`). The basic idea is to use a separate pool for `MUXCORE`, a special gadget that is generated and checked independently of ANDs and can then be used to implement `MUX`. While wire processing accounts for roughly 60–70% of the cost of the underlying LEGO protocols, our technique reduces the number of wires of `MUX` by roughly 1/3. Overall, this optimization is able to boost the performance by 30% for `MUX` and 6–23% for several benchmark applications. (Section 7.2)

New Tool. We developed Pool, a software framework for building pool-based actively secure computation protocols. Pool offers a succinct set of APIs designed for creating future actively-secure computations. We evaluated our approach over several applications including Circuit ORAM, which were challenging to build and run in the malicious threat model. To the best of our knowledge, this is the first implementation of secure computation of ORAM in the standard malicious model. To demonstrate the scalability of our approach, we have run two single-threaded server programs for actively-secure computations executing about 278M logical-AND gates per hour, totaling at 47.3 billion logical-ANDs in seven days. The service can continue to execute even more gates if we did not interrupt and shutdown the servers. The source code of Pool is made available at https://github.com/jimu-pool to stimulate future exploration of related areas.

2 CRYPTOGRAPHIC BACKGROUND

This section briefly introduces the basic ideas of garbled circuits, the cut-and-choose paradigm, and LEGO protocols.

2.1 Garbled Circuits

Garbled circuits allow two parties holding inputs x and y, respectively, to evaluate an arbitrary function $f(x, y)$ without leaking any information about their inputs beyond what is implied by the function output. The basic idea is that one party (the circuit *garbler*) prepares an "encrypted" version of a circuit computing f; the second party (the circuit *evaluator*) then obliviously computes the output of the circuit without learning any intermediate values.

Starting with a Boolean circuit for f (agreed upon by both parties in advance), the circuit generator associates two random cryptographic keys w_i^0, w_i^1 (also known as *wire-labels*) with each wire i of the circuit (w_i^0 encodes a 0-bit and w_i^1 encodes a 1-bit). Then, for each binary gate g of the circuit with input wires i, j and output wire k, the generator computes ciphertexts

$$\mathrm{Enc}^k_{w_i^{b_i}, w_j^{b_j}}\left(w_k^{g(b_i,b_j)}\right)$$

for all inputs $b_i, b_j \in \{0, 1\}$. The resulting four ciphertexts, in random order, constitute a garbled gate. The collection of all garbled gates forms the garbled circuit that is sent to the evaluator. In addition, the generator reveals the mappings from output-wire keys to bits.

The evaluator must also obtain the appropriate keys (that is, the keys corresponding to each party's actual input) for the input wires. The generator can simply send $w_1^{x_1}, \ldots, w_n^{x_n}$, the keys that correspond to its own input where each $w_i^{x_i}$ corresponds to the generator's i^{th} input bit. The parties use *oblivious transfers* [13, 14, 22, 26] to enable the evaluator to obliviously obtain the input-wire keys corresponding to its own inputs. Given keys w_i, w_j associated with both input wires i, j of some garbled gate, the evaluator can compute a key for the output wire of that gate by decrypting the appropriate ciphertext. Thus, given one key for each input wire of the circuit, the evaluator can compute a key for each output wire of the circuit. With the mappings from output-wire keys to bits provided by the garbler, the evaluator can learn the actual output of f.

Free-XOR Technique. If the circuit garbler keeps a global secret label Δ and dictates that for every wire i in the circuit with 0-label w_i^0, its 1-label w_i^1 is always defined as $w_i^1 := w_i^0 \oplus \Delta$; and further, for every XOR gates with input wires i, j and an output wire k, the garbler always sets $w_k^0 := w_i^0 \oplus w_j^0$, then XOR can be securely computed by the evaluator alone through XOR-ing the two input wire-labels it got from evaluating previous gates. This idea first appeared in BMR [2] for the multi-party setting and was reinvigorated by Kolesnikov and Schneider [16] in the two-party setting.

2.2 Dealing with Active Adversaries

Active Adversaries. The garbled circuit protocol as described above only works for passive adversaries who always follow the protocol specification. However, this adversary model can be too weak in practice as an adversary doesn't have to follow the protocol and can actually deviate from the protocol in arbitrary ways. For example, a malicious garbler could plant garbage entries into a garbled gate and infer plaintext signals on intermediate wires by observing the evaluator's response. Following the seminal work of Canetti [3, 4], security of secure computation protocols in presence of active adversaries is defined and proved with respect to an *ideal* model execution where a trusted party exists to help compute the desired functionality. Our work considers protocols against these strong active adversaries.

The Cut-and-Choose Paradigm. The cut-and-choose paradigm is a popular and efficient mechanism for ensuring that the garbled circuit sent by the garbler is constructed correctly. The basic idea is that the circuit generator produces and sends several garbled circuits; the circuit evaluator checks a random subset of these, and evaluates the rest to determine the final result. Existing cut-and-choose protocols fall roughly into three categories: (1) *MajorityCut*, whose security holds as long as a majority of the evaluated circuits are correct; (2) *SingleCut*, which guarantees security as long as at least one of the evaluated circuits are correctly generated; and (3) *BatchedCut*, where the parties batch the cut-and-choose procedure either across multiple instances of an application or at the gate level. Zhu et al. [39] have formalized these mechanisms into zero-sum games and considered their *cost-aware* equilibrium solutions in certain circumstances.

Reactive Functionalities. Certain reactive functionalities, such as RAM-based secure computations, are especially cumbersome to handle by some cut-and-choose mechanisms [1]. For example, let $f^M(x, y)$ be a RAM-based computation that has access to a chunk of (encrypted) memory M, through a randomized ORAM scheme. In essence, the execution of $f^M(x, y)$ can be divided into a series of smaller circuits $f_1^{M_1}, \ldots, f_n^{M_n}$ such that:

$$\begin{aligned}
v_1 &:= f_1^{M_1}(x, y; \quad addresses_1),\\
v_2 &:= f_2^{M_2}(x, y; v_1, addresses_2),\\
v_3 &:= f_3^{M_3}(x, y; v_2, addresses_3),\\
&\cdots\\
v_n &:= f_n^{M_n}(x, y; v_{n-1}, addresses_n).
\end{aligned}$$

where $addresses_i$ is the set of (random) indices revealed by the ORAM mechanism to access the memory. Should the cut-and-choose mechanism be naïvely applied over individual $f_i^{M_i}$, the encrypted memory M_i will also need to be replicated. For cut-and-choose purposes, we could treat the memory as a part of the whole circuit f^M and duplicate this giant circuit with memory M. This strategy, however, is only of theoretical interest due to prominent scalability issues. Alternatively, we can decouple the memory from the circuit and treat each $f_i^{M_i}$ as a small circuit but replicating M_n 40 times (e.g. for 40-bit statistical security) would require M_1 to be replicated 40^n times because M_n is a state that depends on M_i for every $i < n$. Therefore, the LEGO approach has become a preferred candidate to deal with reactive functionalities such as RAM-based secure computations. We further note that these $f_i^{M_i}$ circuits typically involve a large number of gates to realize the underlying ORAM mechanism, thus can incur an intolerable offline latency if existing BatchedCut protocols [6, 19, 25, 27, 38] are naïvely applied.

2.3 LEGO Protocols

LEGO protocols [8, 9, 24, 25, 38] are typical BatchedCut protocols. State-of-the-art LEGO schemes are compatible with the free-XOR technique using either XOR-Homomorphic commitments [8, 9, 25] or XOR-Homomorphic IHashes [38]. With these protocols, it suffices to focus on how to treat ANDs, since all XORs can be securely computed locally without extra treatment to ensure honest behavior.

For a Boolean circuit C of N logical AND gates, the high-level steps of a LEGO protocol to compute C are,

(1) **Generate.** P_1 generates a total of T garbled AND gates.
(2) **Evaluate.** P_2 randomly picks $B \cdot N$ garbled-ANDs (where B is the bucket size) and groups them into N buckets. Each bucket will realize an AND 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 AND gates for correctness. If any of these gates is found faulty, P_2 aborts. Though, depending on the specific schemes, P_2 may not always be able to detect an error when given 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 plaintext values.

Wire-soldering. Every gate bucket realizes a logical gate, thus has input and output wires like the logical gate it maps to. Thus, in order to evaluate an independently generated garbled gate assigned to this bucket, an input-wire of the bucket (with wire-labels w^0_{bucket} and $w^1_{\text{bucket}} = w^0_{\text{bucket}} \oplus \Delta$ denoting 0 and 1) needs to be connected to the corresponding input-wire of a garbled gate inside the bucket (with wire-labels w^0_{gate} and $w^1_{\text{gate}} = w^0_{\text{gate}} \oplus \Delta$). This is done by requiring P_1 to send $d = w^0_{\text{bucket}} \oplus w^0_{\text{gate}}$ and P_2 to xor d with the bucket wire-label (either w^0_{bucket} or w^1_{bucket}) he obtained from evaluating the previous bucket. In order to prevent a malicious P_1 from sending a forged d, XOR-Homomorphic commitments (or i-hashes) can be used to let P_1 "commit" both w^0_{bucket} and w^0_{gate}, with which P_2 can verify the validity of d without learning any extra information about either w^0_{bucket} or w^0_{gate}, as the commitment (or i-hash) of $d = w^0_{\text{bucket}} \oplus w^0_{\text{gate}}$ can be derived homomorphically.

3 APPROACH OVERVIEW

Figure 1 illustrates the high-level idea of our approach. Recall that any function f can be computed using just a set of AND and XOR gates. While XOR can be securely computed without interaction [16], AND gates need to be garbled. Executing the whole circuit can be viewed as a sequential traversal of all the logical AND gates in a topological order. With LEGO protocols, every logical AND gate will be realized by a *bucket* (of garbled ANDs randomly drawn from the pool). Thus, we represent f as a stream of buckets shown on the right part of Figure 1.

Our approach assumes that both servers have access to the same pool of garbled AND gates. The pool needs to be built *only once throughout the lifetime of the two servers*. To build the pool from scratch, the fill_pool procedure below is repeated until the pool is completely filled:

> fill_pool:
> (1) The garbler generates a garbled AND gate g.
> (2) Based on a random bit with bias r_c (i.e., the bit turns out 1 with probability r_c ($0 < r_c < 1$); and 0 with probability $1 - r_c$), the parties decide whether to verify g for correctness (happening with probability r_c) or to place g into the pool (with probability $1 - r_c$).
> (3) A honest party aborts if g is found to be faulty.

Note that it is not necessary that every faulty gate can be detected faulty when being checked, i.e., the detection rate $r_d \leq 1$. The size of the pool, n, is a parameter configurable based on the trade-off between performance and budget. We note that, for the purpose of security proof, the randomness used at step 2 comes from a collaborative coin-tossing protocol, hence is unpredictable to the garbler.

Assume the bucket size is B. In order to execute a logical AND gate in the stream, the parties run exec_bucket:

> exec_bucket:
> (1) B garbled ANDs are randomly picked from the pool and put into the bucket.
> (2) The underlying LEGO protocol's bucket evaluation procedure is called to evaluate all the gates in this bucket to derive the bucket's output wire-label.
> (3) Call fill_pool repeatedly until the pool is again completely filled.

Note that the pool is always *full* before and after executing a logical AND gate. Again, for the purpose of security proof, the randomness used at step 1 comes from a collaborative coin-tossing protocol, hence is unpredictable to the garbler.

For an adversarial garbler, the only way to succeed in attacking our pool-based cut-and-choose mechanism is first successfully slipping some number of faulty gates into the pool without being detected, then hoping that *enough* faulty gates are simultaneously picked from the pool to be placed in the same bucket. The exact definition of "enough" will depend on the security premise of the underlying LEGO protocol. For example, using a bucket-level MajorityCut-able protocol [8] requires $B/2$ gates in a bucket to be faulty whereas a bucket-level SingleCut-able protocol [38] requires all B gates in a bucket to be faulty in a successful attack.

Although this pool idea can generally be applied to many LEGO-style protocols, our discussion in the rest of the paper assumes either JIMU [38] or WRK [36] is used as the base protocol since they offer better cut-and-choose properties that significantly simplify the security analysis. .

Security. We assume the underlying LEGO protocol is secure. Note that the only thing we change about the underlying LEGO protocol is the cut-and-choose mechanism. Thus, to show our approach secure, it suffices to prove that a cut-and-choose failure (i.e. enough bad gates entering the same bucket) happens with a bounded probability, ε. Our formal analysis in Section 4 provides security assurance for our approach.

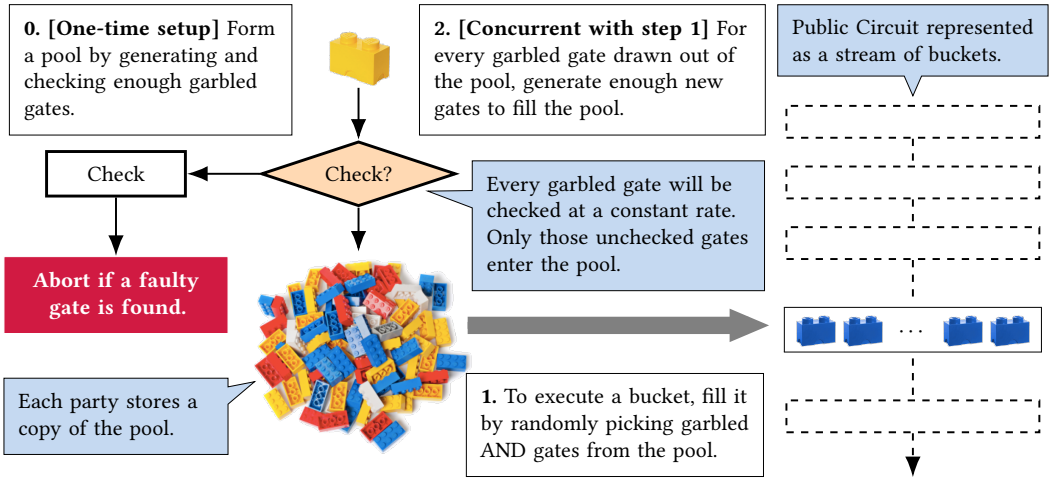


Figure 1: Approach Overview

Notations. Table 2 summarizes some variables used throughout this paper. Note that parameters in the upper-table (i.e., ε, n, r_d) are supposed to be set by the protocol users, whereas those listed in the lower-table (i.e., r_c, B) can be automatically calculated based on the parameters in the upper-table.

Table 2: A Summary of Variables.

ε	Probability of a successful attack throughout the lifetime of the pool.
n	Size of the pool.
r_d	Probability of a checked faulty gate being detected. (This is a constant determined by the underlying LEGO protocol.)
r_c	Probability of a freshly generated garbled gate being checked.
B	Bucket size.

4 POOL ANALYSIS FOR POOL-JIMU

We adopt the LEGO scheme of Zhu et al. [38] where an attack to the cut-and-choose mechanism succeeds only if a bucket contains B faulty gates. Our goal is to bound the probability of an *ever* successful attack to the cut-and-choose mechanism by a preset threshold ε, throughout the lifetime of the pool.

First, as the guard of the pool, we assume every garbled-gate is always selected to be checked with a constant probability r_c. Under this assumption, the *best* strategy for a malicious garbler is to try inserting all b bad gates into the pool from the very beginning (when the pool is initially setup). This is because: (1) the attacker will take exactly the same risk of being detected when b bad gates eventually slip into the pool; however, (2) getting the b bad gates into the pool earlier always leads to better chance of a successful attack because, in absence of successful attacks, drawing gates from the pool to fill the buckets will only consume (the limited number of) bad gates in the pool, hence reducing the probability of a successful attack.

Let b be an upper-bound on the number of bad gates that has *ever* passed through the guard and entering the pool before a successful attack. Let $P_B(n, b)$ be the probability of a successful attack throughout the lifetime of a pool of size-n and bucket-size-B provided that *at most* b bad gates are ever successfully inserted into the pool. Once we know how to compute $P_B(n, b)$ for every n and b, it is easy to upper-bound the success rate of the best attacking strategy by

$$\max_b (1 - r_c r_d)^b \cdot P_B(n, b) \tag{1}$$

for a fixed n, r_c and any positive integer b (it is easy to see that it suffices to consider integer b that is no larger than $\lfloor \log_{1-r_c r_d} \varepsilon \rfloor$, since this is the maximum number of bad gates that could ever be successfully inserted into the pool without being caught with probability larger than ε).

Next, we derive a recurrence that allows us to compute $P_B(n, b)$. Assume there are k bad gates left in the pool. On every draw of B gates, the probability that i bad gates are picked is $\binom{n-k}{B-i}\binom{k}{i} / \binom{n}{B}$. If $i < B$, then i bad gates are consumed so a future attack will succeed with probability $P_B(n, k-i)$. If $i = B$, which happens with probability $\binom{k}{B} / \binom{n}{B}$, an attack succeeds. Thus, $P(n, k)$ can be written as a weighted sum of $\{P_B(n, i)\}_{i=k-B}^{k}$ where the weights are the probabilities that exactly i bad gates are selected:

$$P_B(n,k) = \sum_{i=0}^{B-1} \frac{\binom{n-k}{B-i}\binom{k}{i}}{\binom{n}{B}} P_B(n, k-i) + \frac{\binom{k}{B}}{\binom{n}{B}}, \qquad \forall\, k \geq B;$$
$$P_B(n,k) = 0, \qquad \forall\, k < B.$$

Note that when $k < B$, there is less than B bad gates in the pool, so at least one good gate will appear in each bucket from that point on, hence $P_B(n, k) = 0$. Therefore, once n and B are fixed, $P_B(n, 1), \ldots, P_B(n, b)$ can be computed using dynamic programming altogether in $O(b)$ time.

How to *efficiently* find the best (r_c, B)

Input: ε, n.

Output: r_c, B, which minimize $B/(1 - r_c)$ and satisfy Form (1)$\leq \varepsilon$.

Set $t := \infty$, $r_0 := 0.01$, $b_0 := \lfloor \log_{1-r_0 r_d} \varepsilon \rfloor$, $B := 2$, and repeat the steps below until it exits from Step 5:

(1) Use the recurrence above to compute $P_B(n, k)$ for all $0 \leq k \leq b_0$.
(2) Solve the inequality
$$\max_{k \in \{1, \ldots, b_0\}} (1 - r_c r_d)^k \cdot P_B(n, k) \leq \varepsilon$$
for the smallest possible r_c. If no viable r_c can be found through solving the inequality, go to Step 5.
(3) If $r_c < r_0$, set $r_0 := r_0/2$, $b_0 := \lfloor \log_{1-r_0 r_d} \varepsilon \rfloor$, $B = 2$, and go to Step 1.
(4) If $t \geq B/(1 - r_c)$, then $t := B/(1 - r_c)$, $r_c^* := r_c$, $B^* := B$.
(5) Set $B := B + 1$. If $t < B$, then exit with output (r_c^*, B^*).
(6) If $B > n$, exit with output $\perp$.

Figure 2: Parameter Search for Pool-Jimu

Automated parameter selection. In existing BatchedCut protocols [19, 25], picking good cut-and-choose parameters for a particular circuit size and security parameter often requires considerable human intervention in making heuristic guesses. Such ad hoc procedures are obviously incompatible with the expectation of running dynamically-supplied functions on-the-fly! Next, we show how introducing a pool simplifies the process and enables full automation.

Here we want to pick (r_c, B) such that Form (1) $\leq \varepsilon$ holds while $B/(1 - r_c)$ is minimized, because $B/(1 - r_c)$ gates are expected to be garbled per bucket. Approached naïvely, this would require solving a complex non-linear optimization. However, we propose an efficient search algorithm to identify the best (r_c, B) for any fixed (ε, n) below (see Figure 2). The basic idea is to consider every possible integral B in an increasing order. When examining each potential B value, we first find the smallest r_c that allows the security constraint Form (1)$\leq \varepsilon$ to hold. Recall that the security constraint was derived assuming $r_c > r_0$, hence if a smaller r_c is obtained at Step 2, we will discard this r_c and decrease r_0 by 1/2 and recalculate r_c until $r_c > r_0$ is satisfied. Then in Step 4, we record the resulting $B/(1-r_c)$ value if it is smaller than the by-far smallest cost indicator t. We stop examining bigger Bs and exit at Step 5 if $(t < B)$ because there is no hope to find any smaller achievable target t values (see remark point (4) below). If the input (ε, n) is not securely achievable at all, our search algorithm will exit at Step 6.

We further make several remarks about our search algorithm:

(1) It suffices to start the search from $B = 2$ as $B = 1$ degenerates to semi-honest garbled circuit protocol thus will not offer an interesting result.
(2) Any (r, B) satisfying the inequality of Step 2 is a *security-wise* viable parameter, but we would like to find the smallest r_c (for that B) satisfying this constraint to minimize $B/(1-r_c)$. This can be accomplished efficiently through a binary search between r_0 and 1, because the left-hand-side of the inequality strictly decreases when r_c grows.
(3) In Step 3, if r_0 is found to be greater than r_c, then we cannot be sure that the r_c obtained is the smallest possible (for that B) satisfying the constraint in Step 2; otherwise, we are sure that r_c is optimal for the current B.
(4) If the search exits at Step 5 with $t < B$, then we know t must also be smaller than $B/(1 - r_c)$ for *any* non-negative r_c. Therefore, searching further can't yield any smaller target value t, hence the output (r_c^*, B^*) must be optimal.
(5) A search could also terminate from Step 6, which indicates n is too small to ever achieve ε statistical security at all.
(6) Initiating r_0 to 0.01 is arbitrary as r_0 can be initiated to any decimal between 0 and 1. But in practice, setting $r_0 := 0.01$ alleviates us from resetting r_0 and restart the search (Step 3) for every n we have ever tried (assuming $\varepsilon = 2^{-40}$).

Optimal strategy? In fact, the pool-based cut-and-choose game could be framed as a general cost-aware zero-sum game (the *utility* is each party's winning odds). Our analysis above is by no means the optimal strategy as it unnecessarily restricts the honest party to a particular set of strategies, i.e., using a constant r_c and constant B. We only claim that the solution described above is optimal under the premise of using constant r_c and B. It is entirely possible to guarantee an ϵ-bounded failure rate with even less costly strategies. We leave it as an interesting open question to seek the most cost-effective strategy to this pool-based cut-and-choose game.

Using additional pools. Because any Boolean circuit can be computed using AND and XOR gates and secure XOR can be realized without garbling, it suffices to just have a single pool of garbled AND gates to realize any function. However, we discovered that sometimes it may be beneficial to have pools of other types of garbled gadgets when those gadgets, treated as a whole, can be realized more efficiently than composed from individual AND gates. In these circumstances, the servers can maintain multiple pools, each with a different type of garbled gadgets. The security analysis remains the same for additional pools. We will soon see an example of exploiting a second pool for realizing `MUX` 30% more efficiently.

Costs. These include the time for initializing the pool, the storage for storing the pool, and the time for replenishing the pool. The time to initialize the pool is essentially the pool size n multiplied by $R_g/(1 - r_c)$ where R_g is the speed of garbled-gate generation and r_c is determined by n and ε in a way described earlier. See Table 3 for concrete numbers about pool initialization. We stress that the pool only need to be generated *once in the lifetime* of running the servers. Likewise, the non-amortizable time required to replenish the pool is merely $B \times R_g/(1 - r_c)$ per gate in the target Boolean circuit.

The storage costs of the pool can depend on n but also the role of a server, since the garbler remembers wires whereas the evaluator only need to store the (shorter) i-hashes of wires. Table 4 provides exact numbers. We also note that the evaluator can even hash every garbled gate it receives and organize all the hashes with a Merkle tree. Thus, at a logarithmic cost of operating the Merkle tree, only a single root hash needs to be stored on the evaluator side.

Round-trips. Our approach requires linear rounds but we will discuss how a simple buffering trick can avoid most of the round-trip overhead in Section 6.

5 MORE EFFICIENT SECURE MULTIPLEXERS

MUX is a frequently used component circuit in many computations such as private set intersection and ORAM. An ℓ-bit MUX takes two ℓ-bit inputs (say, x_0 and x_1), a 1-bit choice signal c, and outputs an ℓ-bit output x_c. Conventional approach implements an ℓ-bit MUX by repeatedly calling a 1-bit MUX ℓ times:

```
ℓ-MUX(x[ℓ], y[ℓ], c) {
    for i from 1 to ℓ
      ret[i] ← 1-MUX(x[i], y[i], c);
    return ret;
}
```

Since every 1-MUX can be realized using a single AND as

```
1-MUX(x, y, c) {
    return c∧(x⊕y)⊕y;
}
```

an ℓ-bit MUX only needs ℓ AND gates.

However, recall that with LEGO protocols, the dominating cost is actually due to the *expensive* wires (i.e., the input and output wires of garbled AND gates) instead of the garbled-tables. Existing implementation of ℓ-bit MUX requires ℓ AND gates thus involving 3ℓ wires. Below, we show an optimization that enables more efficient multiplexers by reducing roughly 1/3 of the wires.

Our approach. We note that the ℓ AND gates in an ℓ-bit MUX share a common input bit, i.e., the selection bit of the MUX. To exploit this observation, our key idea is to base our optimized ℓ-bit MUX on a single special $(\ell+1)$-input ℓ-output circuit gadget, which we call MUXCORE, and a few XOR gates. Our MUXCORE can be realized as below:

```
MUXCORE(a[ℓ], c) {
    for i from 1 to ℓ
      ret[i] ← a[i]∧c;
    return ret;
}
```

Then, let $\bigoplus$ be an ℓ-way parallel XOR ($\oplus$), so ℓ-MUX can be simply realized as

```
ℓ-MUX(x[ℓ], y[ℓ], c) {
    return MUXCORE(x⊕y, c) ⊕ y;
}
```

The new implementation will still use ℓ AND gates which are all wrapped in the MUXCORE gadget, however, since MUXCORE has only $2\ell+1$ wires, the total number of expensive wires in a ℓ-bit MUX can thus be reduced from $3\ell+1$ to $2\ell+1$ if we treat MUXCORE as a basic gadget for cut-and-choose.

Checking MUXCORE. To check a garbled MUXCORE gadget, the circuit evaluator simply uses $\ell+1$ randomly sampled input bits to obtain the corresponding $\ell+1$ input wire-labels, evaluates the ℓ AND gates and verifies that the outcomes are consistent with the commitments/i-hashes (just like the way garbled ANDs are checked in the underlying LEGO protocol). Note that because every garbled AND gate inside the MUXCORE gadget is checked with uniformly picked inputs, so any faulty AND gates in MUXCORE are still detected with the same probability r_d offered by the underlying LEGO protocols.

```
class Party {
public:
  /* Encode an input bit of garbler. */
  virtual wire** garblerIn(bool* b, int len)=0;
  virtual wire** garblerIn(int len)=0;

  /* Encode an input bit of evaluator. */
  virtual wire** evaluatorIn(bool* b, int len)=0;
  virtual wire** evaluatorIn(int len)=0;

  /* Reveal an output bit to garbler. */
  virtual bool garblerOut(wire* w)=0;

  /* Reveal an output bit to evaluator. */
  virtual bool evaluatorOut(wire* w)=0;

  /* Basic binary gates */
  virtual wire* and(wire* l, wire* r)=0;
  virtual wire* xor(wire* l, wire* r)=0;

  /* APIs for RAM accesses using ORAM */
  wire** initRAM(int nBlk, int sz);
  wire** accessRAM(wire** mem, wire** rORw,
            wire** index, wire** data);

  /* Run the computation specified by the
     function `f' using `ws'. */
  wire** exec((wire**)f(wire**), wire** ws);

  /* Run the computation specified in the
     circuit file `f' using  `ws'. */
  wire** exec(File* f, wire** ws);
}

class Garbler: public Party {
public:
  wire** garblerIn(bool* b, int len) { ... }
  wire** garblerIn(int len) { ... }
  wire** evaluatorIn(bool* b, int len) { ... }
  wire** evaluatorIn(int len) { ... }
  bool garblerOut(wire* wire) { ... }
  bool evaluatorOut(wire* wire) { ... }
  wire* and(wire* l, wire* r) { ... }
  wire* xor(wire* l, wire* r) { ... }
}

class Evaluator: public Party {
public:
  wire** garblerIn(bool* b, int len) { ... }
  wire** garblerIn(int len) { ... }
  wire** evaluatorIn(bool* b, int len) { ... }
  wire** evaluatorIn(int len) { ... }
  bool garblerOut(wire* wire) { ... }
  bool evaluatorOut(wire* wire) { ... }
  wire* and(wire* l, wire* r) { ... }
  wire* xor(wire* l, wire* r) { ... }
}
```

Figure 3: A succinct set of APIs offered by Pool

6 DESIGN AND IMPLEMENTATION

Design. We have designed and developed Pool, with the goal of making it easier for non-crypto-expert developers to create and run future secure computation services against *active* adversaries. Thanks to the combination of the pool technique and the LEGO-based cut-and-choose, we are able to encapsulate the sophisticated cryptography into a list of application programming interfaces (API) described in Figure 3.

The APIs include eight basic functions, four of which handle function inputs (garblerIn, evaluatorIn) and two of which handle outputs (garblerOut, evaluatorOut) while the rest two (**and**,

xor) handle AND and XOR gates, respectively. Note that there are two overloaded version of `garblerIn`, supposed to be invoked simultaneously by the garbler and the evaluator, respectively. So are the overloaded functions `evaluatorIn`. Since the behaviors of these eight functions depend on their calling party's role (either garbler or evaluator), we define them as virtual functions in the base class `Party` but provide concrete implementations in the sub-classes `Garbler` and `Evaluator`. We stress that since the data structures to represent wires are different on each side, the implementations of `wire` also differ between the `Garbler` and the `Evaluator`.

Pool provides two different ways to specify and execute computations, which are overloaded under the same function name `exec`. The first reads the circuit description from a circuit file (with the SHDL format used by Fairplay [21] and many other existing works [25, 29, 35]) and runs the circuit over the wire encodings supplied as the second argument to `exec`. Our second way allows to specify computations as normal C functions (passed to `exec` as a function pointer `f`). We assume the function `f` only calls AND and XOR, which will be bound to the implementations of **and** and **xor** functions Pool provides. To facilitate specifying functions in this manner, Pool offers a library of circuits such as multiplexers and basic arithmetic circuits like many other programming tools [10, 33, 34] do.

Finally, Pool's (`initRAM` and `accessRAM`) allow application developers to exploit the efficiency benefits of RAM-based secure computation through Circuit-ORAM [30].

Example. Figure 4 shows the use of Pool in developing custom applications. The example is for developing a secure `MUX` to select one of two 8-bit numbers. Note that a developer only needs to write six lines of code to implement `mux` and `mux8` based on ANDs and XORs. The rest two `main` functions are template procedures to run/test custom-built applications (`mux8` in our case). Of course, common circuits like `mux` and `mux8` are already part of Pool's library so developers can use them directly without reinventing the wheel.

Implementation Issues. Implemented naïvely, our protocols will require one round-trip per bucket for the evaluator to disclose his random choice of B garbled gates in the pool to place in a bucket, incurring a significant network round-trip overhead. To alleviate this issue, our simple strategy is to let the garbler maintain an additional *buffer* (of 128K garbled-gates) per pool so that we only need one round-trip per 128K/B buckets. Thus, empty spots in the pool will always be refilled with garbled-gates from the buffer; and whenever the buffer is depleted, the garbler will garble and commit 128K gates altogether, *after* which the evaluator refresh the randomness used to select garbled-gates from the pool till next round of buffer regeneration. This design choice also helps to exploit the fact that hardware-assisted AES runs faster in batches.

```
// Common application code
wire* mux(wire* x, wire* y, wire* c) {
  return xor(and(c, xor(x,y)), y)
}

wire** mux8(wire** ws) {
  wire* x = *ws, y = *ws+8, c= *ws+16;
  wire** ret = new wire*[8];

  for(int i=0; i<8; i++){
    ret[i] = mux(x+i, y+i, c);
  }
  return ret;
}

// Alice's main function
void main() {
  Garbler alice = new Garbler();
  bool inputA[9] = int2bits(0x01AA);
  wire **wsA = alice.garblerIn(inputA, 8);
  wire **wsB = alice.evalautorIn(8);
  wire **ws = new wire*[17];
  for(int i=0; i<8; i++) {
    *ws[i  ]=*wsA[i];
    *ws[i+8]=*wsB[i];
  }
  *ws[16] = **alice.garblerIn(inputA+8, 1);
  wire **os = alice.exec(mux8, ws);
  bool* ret = new bool[8];
  for(int i=0; i<8; i++)
    evaluatorOut(os[i]);
  delete os;
}

// Bob's main function
void main() {
  Evaluator bob = new Evaluator();
  bool inputB[8] = int2bits(0xBB);
  wire **wsA = bob.garblerIn(8);
  wire **wsB = bob.evalautorIn(inputB, 8);
  wire **ws = new wire*[17];
  for(int i=0; i<8; i++) {
    *ws[i  ]=*wsA[i];
    *ws[i+8]=*wsB[i];
  }
  *ws[16] = **bob.garblerIn(1);
  wire **os = bob.exec(mux8, ws);
  bool* ret = new bool[8];
  for(int i=0; i<8; i++)
    ret[i] = evaluatorOut(os[i]);
  printf("%d", bits2int(ret));
  delete os;
}
```

Figure 4: Example Application using Pool

7 EVALUATION

Evaluation Setup. All performance numbers are measured with single-threaded programs. We leased Amazon EC2 machines (instance type: `c4.2xlarge`, Ubuntu Linux 16.04) to conduct all the experiments. We evaluated our implementation in both LAN (2.5 Gbps bandwidth, < 1 ms latency) and WAN (200 Mbps bandwidth, 20 ms latency) settings. We have implemented Pool-Jimu which uses JIMU [38] as the underlying secure computation scheme, thus $r_d = 1/2$. We set $\varepsilon = 2^{-40}$. Reference performance data are measured by running implementations provided by their respective authors in a test environment identical to that of Pool-Jimu.

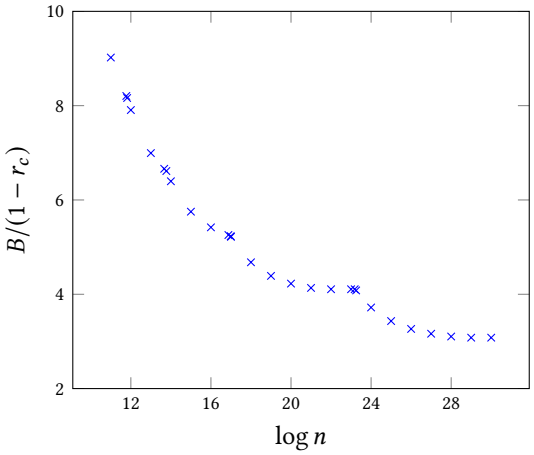
7.1 The Pool

System Efficiency. As was discussed in Section 4, the pool size has a decisive impact on the throughput of our system. Figure 5 depicts how the system performance changes as a result of varying the pool size. Recall that $B/(1-r_c)$ (the Y-axis in Figure 5a) is the expected number of garbled AND gates needed to execute a logical-gate. We also note because the X-axis is on a logarithmic scale, as n increases, $B/(1-r_c)$ actually drops faster than it appears. Therefore,

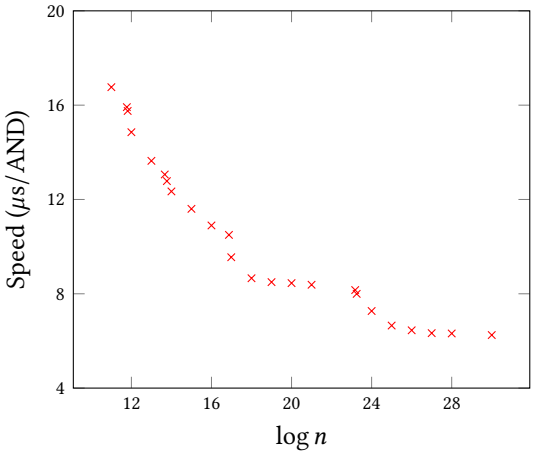
most of the cost savings can be reaped with relatively smaller n, e.g., $B/(1-r_c) \approx 4$ when $n = 8$M.

Figure 5b shows the relation between the pool size and the actual time cost per logical-ANDs of Pool-Jimu. Note that the observed data points in Figure 5b form a curve of similar shape as that of theoretical estimations in Figure 5a. A graphical ratio of 2 on the units of the y-axis between the two figures indicates a garbled-gate processing speed of roughly 500K garbled-gates/second, which coincides well with the micro-benchmark of the underlying JIMU protocol.

Compared to running LEGO protocols without a pool, the benefit of our approach is remarkable. For example, setting $n = 2^{20}$ allows us to execute 100K logical-ANDs/second for circuits of *any sizes*, whereas the same speed can only be achieved on running circuits of more than 100K ANDs using JIMU [38] without a pool. Note that the actual speed measurements also indicate that most efficiency advantage of batched cut-and-choose can be harvested with relatively small pools.



(a) Theoretical Analysis



(b) Practical Observation

Figure 5: Relating System Efficiency to Pool Size

Check Rate and Bucket Size. As was described in Section 4, we calculated the best (B, r_c) values for each fixed pool size to minimize $B/(1-r_c)$. Figure 6 shows, under that premise, how B and r_c changes as n increases. Note that r_c takes continuous values relatively close to 0, whereas B takes discrete integral values. As n grows, B will never increase but the check rate r_c will periodically jump up (synchronized with the changes in B) and then gradually decrease as n increases while B holds the same.

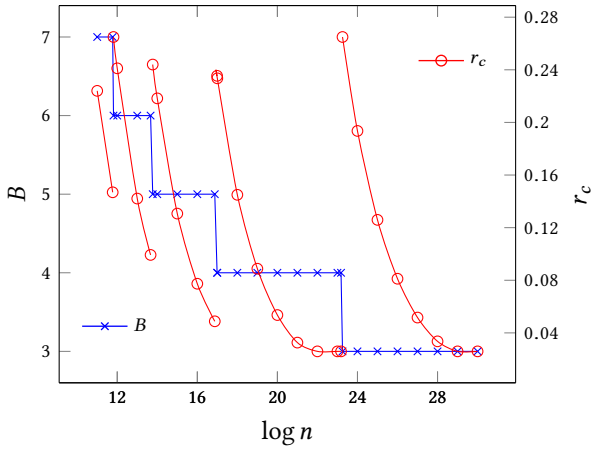


Figure 6: Pool Size versus B and r_c

Time Cost of Pool Initialization. We measured the pool initialization time for several representative pool sizes and give the timings in Table 3. Roughly, we observe that Pool can garble and "commit" 500K garbled-ANDs/second and check 850K garbled-ANDs/second. However, we note that each pool size will favor a particular r_c. Depending on the type of the gadget, pool initialization takes only seconds to dozens of minutes.

Table 3: Timings for Initializing Pool (seconds)

n	10K	100K	1M	10M
r_c	12.2%	5.5%	5.5%	26.5%
AND	0.34	2.94	29.5	411.4
MUXCORE-8	1.92	18.82	188.53	2374.3

Pool Storage. To store each AND gate, the garbler needs to remember three wires and a garbled-table whereas the evaluator only need to remember the commitments/i-hashes of the three wires and a garbled-table. For MUXCORE gadgets, the number of wires and garbled tables will depend on the width of the gadget. Table 4 shows the exact number of bytes needed per AND and per 8-bit MUXCORE.

However, when Pool is optimized for storage in the way discussed in Section 4, the storage requirements drop to 16 bytes (for the seed) on the garbler side and 32 bytes (for the root hash) on the evaluator side.

Table 4: Sizes of Garbled Gate/Gadget (bytes)

	Garbler's Side	Evaluator's Side
AND	288	240
MUXCORE-8	1856	1584

7.2 MUX Optimization

Figure 7 shows the savings in computing oblivious multiplexers due to the use of MUXCORE gadgets. We are able to reduce the overhead of MUX of various widths by about 30%, which is largely in line with our theoretical estimation based on the wire reduction rate and the fraction of overall overhead (60–70%) spent on processing wires.

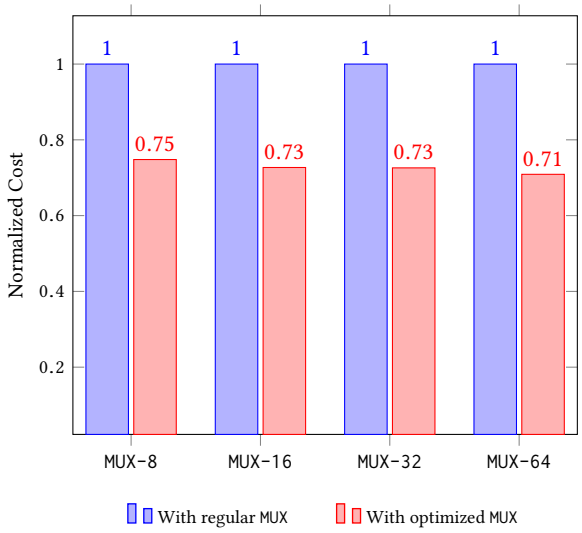
We have also evaluated the gain of this technique over several applications that use multiplexers as building blocks. We observe 6–23% improvements (Figure 7b), since the exact savings now will also depend on the proportion of MUX computation as well as the widths of the MUXs being used.

7.3 Applications

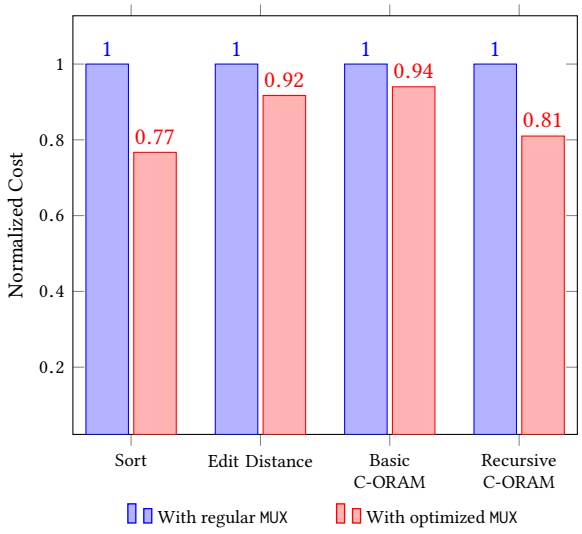
In this study, our main focus is reactive computations such as ORAMs that are difficult to realize with existing implementations of actively-secure protocols. The left half of Table 5 shows the per ORAM access time and bandwidth costs with either a basic Circuit-ORAM or a full-blown one with seven recursions. Because of pooling, we can execute these heavy-weighted computational tasks *without any delay for offline preprocessing*. We note that securely computing randomized ORAMs would be feasible with gate-level BatchedCut protocols like JIMU [38], NST [25], and WRK [36]. However, no prior work of this kind exists, partly due to technical concerns (such as the lack of well-defined programming interfaces) and the anticipated high latency due to offline processing.

For comparison purposes, we also evaluated Pool with several applications, including sort, hamming distance, and edit distance, scaled at moderate-size circuits. For these applications, we are able to run implementations provided by WMK [35] and JIMU [38], whose performance is representative of the state-of-the-art protocols in the malicious adversary model. Compared with the original JIMU protocols [38], using a smaller pool of 35K garbled gates already allows to reap almost all the benefit of batched cut-and-choose. With a bigger pool with 35M garbled gates, we actually observe 1.6–2.6x improvements in time and bandwidth. The speedup can also be explained by the significantly reduced memory usage thanks to the pool, which indirectly helps to reduce time due to improved caching and faster memory accesses. Comparing to WMK [35], Pool-Jimu is more efficient on Hamming Distances because Hamming Distance is secret-input-intensive while Pool-Jimu inherits the efficiency of secret-input processing from JIMU [38]. On the other hand, WMK [35] is still more efficient in computation-intensive applications such as Sort and Edit Distance, though its cut-and-choose mechanism makes it infeasible to support reactive computations such as ORAM.

We have also attempted to run these applications with several other BatchedCut protocols including NST [25], RR [27], and LR [19]. Unfortunately, these proof-of-concept implementations



(a) Microbenchmarks



(b) Applications

Figure 7: Efficiency Gain from Optimized MUX. (Applications were run in the same way as they were measured in Table 5.)

do not provide explicit APIs for users to build applications that were not already included in their implementation. It is also unclear how to efficiently calculate important protocol parameters such as bucket size and check rates for general circuits. In fact, the parameter selection procedures suggested in their security analysis requires computing a great number of combinatorial formulas that involve very big integers, which only worked for determining parameters for a few specific small-scale scenarios but didn't scale up well in general.

Nonetheless, the micro-benchmarks reported in the literature can still shed some light on the comparison. Researchers [38] have shown that the performance of NST [25] is very similar to JIMU [38] which we have included in Table 5. Wang et al. [35] show that

Table 5: Performance of Selected Applications. (Units of the numbers are either seconds or GB.)

		Basic C-ORAM[1]		Recursive C-ORAM[2]		Sort[3]		Hamm. Dist.[4]		Edit Dist.[5]	
		Time	BW	Time	BW	Time	BW	Time	BW	Time	BW
WMK	LAN	Hard to Support (for its cut-and-choose mechanism)				46.2	14.2	44.5	9.2	302.3	93.8
	WAN					575.3		381.5		3781.5	
JIMU	LAN	Never Done Before (for API and memory scalability issues)				228.4	28.4	40.7	6.04	1677.2	203
	WAN					1172.3		244.4		8754	
This work (Pool-Jimu)	n = 35K (LAN)	46.6	6.38	15.4	1.9	149.3	27.0	36.9	6.42	1247	215
	n = 35K (WAN)	279.6		83.2		1168		268.9		9450	
	n = 35M (LAN)	31.4	3.96	11.2	1.08	87.1	15.9	24.9	3.8	741.2	127
	n = 35M (WAN)	174.7		48.0		662.6		160		5333	

When n = 35K, we set $B = 5, r_c = 12.5\%$; when n = 35M, we set $B = 3, r_c = 12.3\%$. Optimized `MUX`es are used whenever possible.

[1] Accessing an array of 10000 32-bit blocks using Circuit-ORAM without recursion (1.82M ANDs);
[2] Accessing an array of 10000 32-bit blocks using Circuit-ORAM with a recursion factor of 8 and a cutoff threshold of 256 (resulting in 2 levels of recursion totaling at 665K ANDS);
[3] Sort 4096 32-bit numbers (10.2M ANDs);
[4] Hamming distance between two 1M-bit strings (2.1M ANDs);
[5] Edit distance between two 1024-nucleotide DNA (73.3M ANDs).

LR [19] and RR [27] are about 1.5–3x faster than WMK [35], hence about 2–5x more efficient than Pool-Jimu. Note that LR and RR require significant function-dependent offline processing, which would adversely affect their general applicability. In comparison, Pool-Jimu is able to run all these six applications (and any other dynamically defined functions) using the same single pool of AND gates.

We also note that Wang et al. has recently proposed WRK, a highly efficient, constant-round protocol [36] based on authenticated multiplicative triples and authenticated garbling. Section 8 discusses how the idea of Pool can be adapted to make WRK more scalable.

Extreme Scales. To evaluate the scalability of Pool-Jimu, we have run two single-threaded programs on two LAN-connected servers (Intel Xeon 2.5 GHz) for executing actively-secure computations with a pool of 16M gates. The service has been up *non-stop* for seven days (until we intentionally shut it down), executing 47.3 billion gates at about 278M logical-ANDs/hour.

8 APPLYING POOL TO WRK

The pool idea can also be combined with WRK [36], an *authenticated-garbling*-based protocol that is by far the fastest actively-secure two-party computation scheme. We call our WRK-based protocol Pool-WRK.

A Brief Overview of WRK. WRK is a constant-round two-party computation protocol based on authenticated multiplicative triples, i.e., $(a_1 \oplus a_2) \wedge (b_1 \oplus b_2) = c_1 \oplus c_2$ where $a_1, a_2, b_1, b_2, c_1, c_2 \in \{0, 1\}$. The protocol requires a preparation phase to generate a linear (in the circuit size) number of such triples and distribute them properly between the two parties. That is, P_1 holds three bits a_1, b_1, c_1 and their authentication tags (127-bit each) $\langle a_1 \rangle, \langle b_1 \rangle, \langle c_1 \rangle$ that allow P_1 to later prove to P_2 the use of authentic values of a_1, b_1, c_1 when needed; while similarly P_2 holds a_2, b_2, c_2 and $\langle a_2 \rangle, \langle b_2 \rangle, \langle c_2 \rangle$. Using each authenticated multiplicative triple, the parties will collaboratively garble a binary AND gate such that (1) the permutation of the garbled entries are determined by the authenticated random bits from both parties; and (2) a honestly garbled entry in the table can always be verified upon decryption. As a result, although a malicious garbler could render some garbled entries invalid, it cannot gain any information by observing failed gate evaluations because in the adversary's perspective the failures always happen at random places.

The most significant cost of WRK protocol is due to the preparation phase (called $\mathcal{F}_{\text{pre}}$) that generates the authenticated multiplicative triples. The improved preparation protocol Wang et al. proposed to realize $\mathcal{F}_{\text{pre}}$ is similar to batched style of cut-and-choose, hence able to achieve asymptotic efficiency similar to LEGO protocols. While the focus of the original WRK paper was on designing authenticated garbling and proving its security, techniques provided in our work helps to scale up their scheme to run arbitrary

How to *efficiently* find the best B

Input: ε, n.
Output: the smallest B that satisfies Form (2)$\leq \varepsilon$.

Set $t := \infty$, $b_0 := \lfloor -\log_2 \varepsilon \rfloor$, $B := 2$, and repeat the steps below until it exits from Step 2:

(1) Use the recurrence above to compute $P_B(n,k)$ for all $0 \leq k \leq b_0$.
(2) If $\max_{k\in\{1,\ldots,b_0\}} 2^{-k} \cdot P_B(n,k) \leq \varepsilon$, then output B and halt; otherwise, set $B := B+1$.
(3) If $B > n$, exit with output $\perp$.

Figure 8: Parameter Search for Pool-WRK

size computations with no preparation delay and limited storage (independent of the circuit size).

Pool Analysis Adapted to WRK. WRK's expensive preparation phase also used the ideas of bucketing and batched cut-and-choose, except that it not only detects faults in the generation of *every* multiplicative triple with probability 1/2, but also *allows all checked triples to be used to form buckets*. Thus, to specialize our pool analysis for WRK, we set $r_c = 1$ (since all triples are checked) and $r_d = 1/2$. Therefore, assuming the bucket size B is a constant, the success rate (Form (1) of Section 4) of the best attacking strategy becomes

$$\max_b 2^{-b} \cdot P_B(n,b) \tag{2}$$

where b can be any positive integer (but essentially bounded by $\lfloor \log_{1-r_c r_d} \varepsilon \rfloor$) and the analysis of $P_B(n,b)$ is identical to that appeared in Section 4, because like JIMU [38], WRK only requires a single honestly generated multiplicative triple in each bucket to guarantee security. As r_c is now fixed to 1, the parameter selection procedure can be significantly simplified to the one described in Figure 8:

Finally, assuming the bucket size needs to be constant, B calculated as above is known to be optimal.

Benefits. The main benefit the pool brings to the WRK scheme is improved scalability. Compared to plain WRK, when executing large circuits the long stalls and the big storage requirement of the preparation phase are no longer needed. We have listed the pool parameters and compared them to the plain WRK (Table 6). A pool allows us to completely drop the circuit-size constraint for achieving a particular efficiency level. Also, note that the minimal pool-sizes needed in our approach are only about 60% of the minimal circuit sizes required by plain WRK. An intuitive explanation of this phenomenon is that, as the total number of bad entries used by an attacker is bound by $-\log_2 \varepsilon$, the probability of having a bad bucket has to be smaller than an inverse polynomial of the size of the pool when a pool is employed. Without a pool, however, that probability will continually increase as the pre-computed leaky-ANDs are consumed for evaluating circuits. So it requires fewer leaky-ANDs in the pool to achieve the same level of efficiency.

Table 6: Compare WRK and Pooled WRK in terms of $\mathcal{F}_{\text{pre}}$.

	Bucket Size		3	4	5
40-bit statistical security	WRK	Minimal circuit size[1] (by # of logical-ANDs)	280K	3.1K	320
		Minimal circuit size[2] (by # of leaky-ANDs)	840K	12400	1600
	WRK with Pool	Minimal pool size[3] (by # of leaky-ANDs)	479K	7673	1073
		Circuit-size constraint	***No size constraints***		
64-bit statistical security	WRK	Minimal circuit size[1] (by # of logical-ANDs)	1.2B	780K	21K
		Minimal circuit size[2] (by # of leaky-ANDs)	3.6B	3120K	105K
	WRK with Pool	Minimal pool size[3] (by # of leaky-ANDs)	1.96B	1963K	68.3K
		Circuit-size constraint	***No size constraints***		
80-bit statistical security	WRK	Minimal circuit size[1] (by # of logical-ANDs)	300B	31M	330K
		Minimal circuit size[2] (by # of leaky-ANDs)	900B	124M	1650K
	WRK with Pool	Minimal pool size[3] (by # of leaky-ANDs)	501.8B	79.15M	1093K
		Circuit-size constraint	***No size constraints***		

[1] These numbers are quoted from WRK [36].
[2] Each of these numbers is obtained by multiplying the minimal number of logical AND gates in the circuit and its corresponding bucket size.
[3] Pool size is measured by the number of leaky-ANDs instead of logical ANDs because the notion of logical-ANDs is less relevant until a bucket of leaky-ANDs are picked from the pool.

9 OTHER RELATED WORK

Kreuter et al. [17] proposed a technique to run arbitrarily large-scale secure computations against malicious adversaries. The basic idea is to use an additional set of oblivious transfers to allow the evaluator to secretly learn either the seed for verifying a circuit or the input wire-labels for evaluating the circuit. Although Kreuter et al.'s original protocol was based on MajorityCut, the idea is later adopted by Wang et al. in their SingleCut protocol [35], which is by far the most efficient SingleCut protocol in the single-execution setting. The heavily optimized WMK [35] is able to process 227K logical-ANDs per second and roughly 50K input/output wires per second and seems able to run circuits of any size without significant preprocessing. In comparison, Pool executes the gates 1.5–2x slower but is 2.4x, 24x, and 600x faster in handling the evaluator's

input, garbler's input, and outputs, and scales up very well. Finally, it seems clumsy to use [17, 35] to handle RAM-based secure computations because their underlying cut-and-choose mechanisms are not very compatible with reactive computations.

Motivated by the need for secure computation as a commodity service, researchers have conceived asymptotically more efficient BatchedCut protocols aiming at efficient *iterative* execution of a function with different inputs [12, 18]. Lindell et al. proposed an efficient symmetric-key operation based input consistency enforcement technique and gave the first implementation of such a protocol [19]. More recently, Rindal and Rosulek [27] have pushed Kolesnikov et al.'s work on Dual-Execution protocols [11, 15] into the offline/online setting with an efficient method for input consistency and a lightweight PSI, both tailored to the dual-execution paradigm. Both works exploited parallel hardware support for minimizing the online and offline times. In contrast, our work aims to batch the cut-and-choose procedure at the basic-gates level and focuses on single-thread implementations to minimize the number of CPU cycles required by our protocol. Unlike our approach, it would be unrealistic to support RAM-based computation using these protocols.

Damgård et al. proposed SPDZ [5, 6], a protocol capable of supporting more than two participants in computing arithmetic circuits. It consists of a somewhat homomorphic cryptosystem-based constant-round offline stage and a linear-round (in circuit depth) online stage. These protocols would be useful to compute shallow circuits in low network latency environments but less competitive comparing to state-of-the-art constant-round LEGO protocols.

10 CONCLUSION

Running gate-level BatchedCut secure computation protocols with a pool leads to a number of benefits including consistent, faster execution of any-size circuits with nearly zero offline processing. We instantiated this idea with two state-of-the-art secure computation schemes and incorporated it into a software framework that offers several valuable properties for delivering actively-secure computation as an on-demand service. We hope Pool will help spur interest in developing and deploying practical secure computation services.

ACKNOWLEDGMENTS

We thank Xiao Wang for suggestions on efficient implementations. This work is supported by NSF award #1464113 and NIH 1U01EB023685-01.

REFERENCES

[1] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. 2015. How to efficiently evaluate RAM programs with malicious security. In *EUROCRYPT*.
[2] Donald Beaver, Silvio Micali, and Phillip Rogaway. 1990. The round complexity of secure protocols. In *STOC*.
[3] Ran Canetti. 2000. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology* 13, 1 (2000), 143–202.
[4] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*.
[5] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel Smart. 2013. Practical covertly secure MPC for dishonest majority–or: breaking the SPDZ limits. In *ESORICS*.
[6] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*.
[7] Jack Doerner, David Evans, and Abhi Shelat. 2016. Secure Stable Matching at Scale. In *ACM CCS*.
[8] Tore Frederiksen, Thomas Jakobsen, Jesper Nielsen, Peter Nordholt, and Claudio Orlandi. 2013. Minilego: Efficient secure two-party computation from general assumptions. In *EUROCRYPT*.
[9] Tore Frederiksen, Thomas Jakobsen, Jesper Nielsen, and Roberto Trifiletti. 2015. *TinyLEGO: An Interactive Garbling Scheme for Maliciously Secure Two-party Computation.* http://eprint.iacr.org/2015/309
[10] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. 2011. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium*.
[11] Yan Huang, Jonathan Katz, and David Evans. 2012. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE Symposium on Security and Privacy*.
[12] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex Malozemoff. 2014. Amortizing garbled circuits. In *CRYPTO*.
[13] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending oblivious transfers efficiently. In *CRYPTO*.
[14] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2015. Actively secure OT extension with optimal overhead. In *CRYPTO*.
[15] Vladimir Kolesnikov, Payman Mohassel, Ben Riva, and Mike Rosulek. 2015. Richer efficiency/security trade-offs in 2PC. In *TCC*.
[16] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages, and Programming*.
[17] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. 2012. Billion-Gate Secure Computation with Malicious Adversaries. In *USENIX Security Symposium*.
[18] Yehuda Lindell and Ben Riva. 2014. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In *CRYPTO*.
[19] Yehuda Lindell and Ben Riva. 2015. Blazing fast 2pc in the offline/online setting with security for malicious adversaries. In *ACM CCS*.
[20] Chang Liu, Xiao Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy*.
[21] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay-Secure Two-Party Computation System. In *USENIX Security Symposium*.
[22] Moni Naor and Benny Pinkas. 2001. Efficient oblivious transfer protocols. In *SODA*.
[23] Kartik Nayak, Xiao Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel secure computation made easy. In *IEEE Symposium on Security and Privacy*.
[24] Jesper Nielsen and Claudio Orlandi. 2009. LEGO for two-party secure computation. In *TCC*.
[25] Jesper Nielsen, Thomas Schneider, and Roberto Trifiletti. 2017. Constant Round Maliciously Secure 2PC with Function-independent Preprocessing using LEGO. In *NDSS*.
[26] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. 2008. A framework for efficient and composable oblivious transfer. In *CRYPTO*.
[27] Peter Rindal and Mike Rosulek. 2016. Faster malicious 2-party secure computation with online/offline dual execution. In *USENIX Security Symposium*.
[28] Ebrahim Songhori, Siam Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. 2015. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *IEEE Symposium on Security and Privacy*.
[29] Stefan Tillich and Nigel Smart. 2014. *Circuits of Basic Functions Suitable For MPC and FHE.* http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/
[30] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *ACM CCS*.
[31] Xiao Wang, Yan Huang, Hubert Chan, Abhi Shelat, and Elaine Shi. 2014. SCORAM: oblivious RAM for secure computation. In *ACM CCS*.
[32] Xiao Wang, Yan Huang, Yongan Zhao, Haixu Tang, XiaoFeng Wang, and Diyue Bu. 2015. Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In *ACM CCS*.
[33] Xiao Wang, Chang Liu, Yan Huang, Kartik Nayak, Elaine Shi, and Michael Hicks. 2015. *ObliVM.* https://github.com/oblivm
[34] Xiao Wang, Alex Malozemoff, and Jonathan Katz. 2016. *EMP-toolkit: Efficient MultiParty computation toolkit.* https://github.com/emp-toolkit
[35] Xiao Wang, Alex Malozemoff, and Jonathan Katz. 2017. Faster Secure Two-Party Computation in the Single-Execution Setting. In *EUROCRYPT*.
[36] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated garbling and efficient maliciously secure two-party computation. In *ACM CCS*.
[37] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting square-root oram: Efficient random access in multi-party computation. In *IEEE Symposium on Security and Privacy*.
[38] Ruiyu Zhu and Yan Huang. 2017. JIMU: Faster LEGO-based Secure Computation using Additive Homomorphic Hashes. In *ASIACRYPT*.
[39] Ruiyu Zhu, Yan Huang, Abhi Shelat, and Jonathan Katz. 2016. The Cut-and-Choose Game and its Application to Cryptographic Protocols. In *USENIX Security Symposium*.