nanoPI: Extreme-Scale Actively-Secure Multi-Party Computation

Resolving the Space-Round Dilemma using Lightweight Program Instrumentation

Ruiyu Zhu
Indiana University
zhu52@indiana.edu

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

Amr Sabry
Indiana University
sabry@indiana.edu

Yan Huang
Indiana University
yh33@indiana.edu

ABSTRACT

Existing actively-secure MPC protocols require either linear rounds or linear space. Due to this fundamental space-round dilemma, no existing MPC protocols is able to run large-scale computations without significantly sacrificing performance. To mitigate this issue, we developed nanoPI, which is practically efficient in terms of both time and space. Our protocol is based on WRK [44, 45] but introduces interesting and necessary modifications to address several important programmatic and cryptographic challenges. A technique that may be of independent interest (in transforming other computation-oriented cryptographic protocols) is a *staged execution model*, which we formally define and realize using a combination of lightweight static and dynamic program instrumentation. We demonstrate the unprecedented scalability and performance of nanoPI by building and running a suit of benchmark applications, including an actively-secure four-party logistical regression (involving 4.7 billion ANDs and 8.9 billion XORs) which finished in less than 28 hours on four small-memory machines. Our integrated framework nanoPI is open-sourced at https://github.com/nanoPIMPC/nanoPI.

CCS CONCEPTS

• **Security and privacy** → **Privacy-preserving protocols**; • **Theory of computation** → **Cryptographic protocols**; • **Software and its engineering** → *Dynamic analysis*; *Frameworks*; Semantics;

KEYWORDS

Large-scale actively-secure constant-round MPC

ACM Reference Format:
Ruiyu Zhu, Darion Cassel, Amr Sabry, and Yan Huang. 2018. nanoPI: Extreme-Scale Actively-Secure Multi-Party Computation : Resolving the Space-Round Dilemma using Lightweight Program Instrumentation. In *Proceedings of 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243850

1 INTRODUCTION

Multi-party computation (MPC) is an important cryptographic technique that enables decentralized collaborative computations over sensitive datasets privately held by multiple distrustful parties [9, 34, 41]. After decades of intensive research, state-of-the-art two-party secure computation protocols are able to execute more than 100K AND-gates/second [20, 43, 44, 47, 48] even in the presence of full-malicious adversaries. Recently, Wang et al. have developed authenticated garbling technique which enables efficient constant-round n-party computations secure against up to $n-1$ malicious adversaries [45]. Such throughput would meet the expectation of numerous real-world computations that involve secret data at scale, even when the parties are spread around the globe!

On the other hand, the *space-complexity* of actively-secure computation protocols has been largely overlooked in existing efforts. Instead, researchers have focused intensively on improving the time, bandwidth, and round efficiency of these protocols. However, space is in general a resource as valuable and scarce as time/bandwidth. In fact, space can play an even more critical role in the particular context of secure computation, motivated by two common scenarios, and potentially their combination:

(1) **Resource-constrained devices.** Much sensitive data has been collected and stored on personal devices such as smart phones, watches and IoT devices, whose space budgets can be stringent compared to conventional computers. Nevertheless, people often still hope to run secure computation directly on such devices because it can substantially simplify the trust model when the secret data doesn't need to leave these devices [6, 13, 31].
(2) **Computations at scale.** Generic computations acceptable to most MPC protocols are represented by boolean circuits, whose scale is typically hundreds to thousands times larger than the same computation executed in assembly code. Moreover, some interesting applications of multi-party computation seem inherently computation-intensive. E.g., it has long been envisioned that secret-shares of sensitive user data such as medical records or business transactions can be delegated to multiple independent, untrusted principals, who run MPC protocols in predefined ways to mine useful information, e.g., a prediction model, out of the secret data [26, 30, 37]. As a result, it is not uncommon to run into circuits with billions of gates.

Regarding the space challenge, we examined the literature and existing MPC prototypes but found, unfortunately, that for all known protocols and their variants, either the performance is severely impeded by the large number of rounds required to run the protocol, or they could not even complete due to their enormous demand in space. The work of Whitewash [5] researched ways to reduce the memory footprint of secure computation protocols. However, their study was constrained in the two-party, server-assisted setting (where a mutually-trusted non-colluding server exists) and only against semi-honest adversaries.

Let $|C_f|$ be the size of the *circuit* representation of a function f. A root cause of the scalability issue for the state-of-the-art actively-secure MPC protocols is that they are trading $O(|C_f|)$ space for the performance benefit of being constant-round. Apparently, *constant-round* and *constant-space* have been two fundamentally incompatible assets of any actively-secure computation schemes.

In an attempt to resolve the challenge, Zhu et al. [48] proposed the idea of pool-based cut-and-choose and developed a prototype (based on JIMU [47]) that is able to efficiently run applications with billions (or trillions) of gates using a relatively small constant space. Other advantages brought by their framework include better APIs (for active-security), long-term security, and efficient support of computations over *dynamic* data.[1] The pool-based framework fits well to the needs of establishing long-term commodity services for secure computations. However, it remains open how their idea can be combined with WRK protocols [44, 45], which are more updated and able to generalize to more than two parties.

Therefore, the main quest of this work stems from the question:

> Can we design an actively-secure *multi-party* computation scheme that can *efficiently* execute circuits at *arbitrary* scale using *limited space* independent of $|C_f|$?

We answered this question positively by developing a prototype and have experimentally shown that actively-secure MPC can be efficiently run at extreme-scales using constant space.

Threat Model. In this paper, we only consider protocols that are secure against malicious (aka. active) adversaries. Such adversaries are allowed to behave in arbitrary ways to compromise security. In particular, we make no assumption on the number of parties an adversary can corrupt in the protocol. This is by far the strongest security model one can ever hope to accomplish. Comparing to the frequently-used honest-but-curious threat model, the malicious model is clearly preferred in business scenarios where the stakes are high.

1.1 Contribution

New Problems. First, we unveil the space-round dilemma, a severe issue that plagues the scalability of all existing actively-secure multi-party computation protocols. We further zoomed in to WRK, a state-of-the-art MPC scheme, and discovered several hidden issues that prevent it from being efficiently scalable. These include

(1) WRK protocols *require* fully-unrolling its target computation into circuits. Otherwise, they are no longer constant- (but *linear*-) round. This is caused by several factors including its sub-protocols Π_{abit} (for producing authenticated bits), Π_{aAND} (for producing authenticated ANDs), but most importantly, an undocumented missing step to bridge the gap between the use of *function-independent* Π_{aAND} and Π_{abit} sub-protocols and its *function-dependent pi2pc* protocol, which needs extra rounds of communication. In addition, their authenticated garbling protocol Π_{2pc} has ignored the enormous space demand for storing all wires in the circuit.

(2) To efficiently produce arbitrary number of abits within constant memory, WRK's Π_{abit} would need to be invoked multiple times. However, the global secret Δ used in Π_{abit} to MAC abits will change across different invocations of Π_{abit}. This issue, if left untreated, can lead to actual security attacks.

Our Solutions. To build efficiently scalable MPC protocols, our starting point is WRK [44, 45], the most efficient MPC protocol known so far that is secure against any number of active adversaries.[2] At a high level, we propose three enhancements to WRK to allow it to execute arbitrarily large circuits within constant space.

(1) We change WRK's circuit processing protocol Π_{2pc} (resp. Π_{mpc}) to $\Pi_{\mathsf{2pc}}^{\mathsf{Scalable}}$ (resp. $\Pi_{\mathsf{mpc}}^{\mathsf{Scalable}}$) which can efficiently process large circuits in small space. We propose a lightweight program transformation that can be applied to programs specified in an imperative programming language. As a result, all binary gates in the circuit will be automatically executed in batches (which we dubbed as *stages*), meanwhile all intermediate wires will be automatically deallocated based on their static scoping information. Since it is a common practice to trade space for roundtrip in MPC protocols, our use of lightweight static and dynamic instrumentation technique may be of independent interest in improving other protocols.

(2) We change WRK's authenticated bit (abit) sub-protocol Π_{abit} into $\Pi_{\mathsf{abit}}^{\mathsf{Scalable}}$ where values of Δ are guaranteed to be identical across different invocations of the abit protocol. This change reduces the space requirement of the abit sub-protocol because the abits can be obtained in many small batches.

(3) We change WRK's authenticated AND (aAND) sub-protocol Π_{aAND} into $\Pi_{\mathsf{aAND}}^{\mathsf{Scalable}}$ by maintaining a fixed-size pool of leaky-aANDs for efficient cut-and-choose purpose. Since leaky-aANDs are always picked from the pool, it allows fast generation of arbitrarily many aANDs using constant space.

Our result is a $O(p)$-space, $O(n|C_f|/p)$-round actively secure MPC protocol where p is a user-set constant. Comparing to existing linear-round MPC protocols, ours allows to reduce the penalty of round-latency by a factor of p, regardless of the depth of application circuits. We have formally proved the security of our protocols. Comparing to existing constant-round WRK protocols, ours scales much better in space-complexity and provides easier-to-use APIs and long-term security if running as commodity secure computation services.

We implemented and experimentally evaluated the effectiveness of our ideas. With the proposed techniques, we are able to run, for the first time, a 4-party actively-secure logistic regression with 4.7 billion AND gates in 27.9 hours on mediocre machines (`c5.large`, 4GB memory, 4.75 cents/hour). As a highlight of scalability, our protocol executed a 4-party actively-secure computation of a circuit with more than 40.8 billion ANDs (in addition to 122 billion XORs) on 4 Google Compute Engine `n1-standard-1` instances (1 vCPU, 3.75 GB memory, 4.75 cents/hour) in 16 days. Notably, no more than 398MB memory (and 0 disk space) is used at any point during the computation. Like Pool-JIMU, our protocol also meets other expectations of being used for running long-term commodity services.

[1] In a computation over dynamic data, some of the input may not be available by the time the computation starts. An example computation of this kind is secure evaluation of RAM programs, where some inputs to the circuit need to be read on-the-fly from the Oblivious-RAM.

[2] We consider it trivially secure if all players are adversarial since there is no honest player remaining to be protected.

We packed our compiler and cryptographic implementation into a toolchain and open-sourced it on GitHub.[3]

2 PRELIMINARIES

We describe some building blocks of our MPC protocol, including garbled circuits, WRK, and pool-based cut-and-choose.

2.1 Garbled Circuits

To compute an arbitrary function f using garbled circuit, the basic idea is to let one party (called the *garbler*) prepare an "encrypted" version of the circuit computing f; the second party (called the *evaluator*) then obliviously evaluates the encrypted circuit without learning any intermediate values. Starting with a Boolean circuit for f (agreed upon by both parties in advance), the garbler associates two random cryptographic keys $\mathsf{L}_i^0, \mathsf{L}_i^1$ (also known as *wire-labels*) for the i-th wire in the circuit (L_i^0 encodes a 0-bit and L_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 garbler computes ciphertexts $\mathsf{Enc}_{\mathsf{L}_i^{b_i}, \mathsf{L}_j^{b_j}}\left(\mathsf{L}_k^{g(b_i,b_j)}\right)$ for all possible values of $b_i, b_j \in \{0,1\}$. The resulting four ciphertexts, in random order, constitute a garbled gate for g. In addition, the garbler reveals the mappings from output-wire keys to bits. To start circuit evaluation, the evaluator obtains the appropriate keys for the initial input-wires either through direct messages or *oblivious transfer* [17, 18, 33] from the garbler. Given keys $\mathsf{L}_i, \mathsf{L}_j$ 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. With the mappings from output-wire keys to bits provided by the garbler, the evaluator can learn the actual output of f.

The Point-and-Permute Technique. The *point-and-permute* technique proposed by Pinkas et al. [38] enables the evaluator to always compute a single decryption per gate. The idea is to use L_i^0 to represent a random bit $\lambda_i \in \{0,1\}$ on the i-th wire, so that bit $b_i \oplus \lambda_i$ can be revealed to the evaluator to index the garbled entry for decryption. More specifically, let $\lambda_i, \lambda_j, \lambda_k$ be the *permutation* bits of the two input-wires and the output-wire of a gate. Then L_i^0 and L_j^0 should be used to encrypt output key $\mathsf{L}_k^{g(\lambda_i,\lambda_j)\oplus\lambda_k}$. Thus, a garbled table for g = AND can be expressed as the forth column of Table 1. Also, because the evaluator doesn't know $\lambda_i, \lambda_j, \lambda_k$, it is safe to send the garbled table without further permutation. Note that in the random oracle model, $\mathsf{Enc}_{x,y}(z)$ can be realized as $\mathsf{H}(x,y) \oplus z$ where H is modeled as a random oracle.

The Free-XOR Technique. The Free-XOR technique [3, 21] allows XOR gates to be securely computed without any interaction even in presence of malicious adversaries. The basic idea is to let the circuit garbler keep a global secret Δ and dictate that for every wire i whose 0-label is L_i^0, its 1-label L_i^1 is always defined as $\mathsf{L}_i^1 := \mathsf{L}_i^0 \oplus \Delta$. Further, for an XOR gate with input wires i, j and output wire k, the garbler will always set $\mathsf{L}_k^0 := \mathsf{L}_i^0 \oplus \mathsf{L}_j^0$. Thus, XOR can be securely computed by the evaluator alone through XOR-ing the two input wire-labels it obtained from evaluating previous gates.

2.2 WRK Protocols

The garbling protocol given in Section 2.1 can only thwart semi-honest adversaries. In the standard *malicious* threat model, however, a malicious circuit generator can put erroneous rows into the garbled table. Based on the values of the permutation bits $\lambda_i, \lambda_j, \lambda_k$ along with the fact of whether the evaluation succeeds, an malicious garbler can learn extra information about the plaintext wire signals involved in the erroneous gates. To thwart such attacks, Wang et al. [44] proposed a seminal technique called *authenticated garbling*. The basic idea is to hide the permutation bits from any subset of the parties so that in event of a malicious generator corrupting some garbled rows, it has no clue of which pair of plaintext values a garbled row is associated with. Meanwhile, authenticated garbling enables the circuit evaluator to locally verify whether a decrypted row was indeed correctly constructed. Therefore, a protocol execution will fail or succeed, but in either case its behavior is independent of any honest party's secret inputs.

WRK is by far the most practical constant-round actively-secure n-party computation scheme that tolerates any number of corrupted parties. It is compatible with the powerful Free-XOR technique. Nevertheless, it requires $O(n|C_f|)$ space and works only in the random oracle model. Their key enabling tool is *authenticated AND triples* (aAND) that are pre-computed using a separate secure computation protocol. An AND triple in the two-party setting is a tuple of six bits a_1, b_1, c_1 held by party P_1 and a_2, b_2, c_2 held by P_2 such that $(a_1 \oplus a_2) \cdot (b_1 \oplus b_2) = c_1 \oplus c_2$. Assume P_1 has a secret value $\Delta_1 \in \{0,1\}^n$. We denote by $[b]^1$ an authenticated bit b of party P_1, which refers to a distributed tuple $(b, \mathsf{M}[b], \mathsf{K}[b])$ such that $\mathsf{M}[b] = \mathsf{K}[b] \oplus b\Delta_1$ where P_1 has $(b, \mathsf{M}[b])$, and P_2 knows $\mathsf{K}[b]$. We call $\mathsf{M}[b] \in \{0,1\}^n$ the Message Authentication Code (MAC) of b, and $\mathsf{K}[b] \in \{0,1\}^n$ the *verification key* of b's MAC. An *authenticated AND triple* is just a tuple of six authenticated bits $[a_1]^1, [b_1]^1, [c_1]^1, [a_2]^2, [b_2]^2, [c_2]^2$ such that $(a_1 \oplus a_2)(b_1 \oplus b_2) = c_1 \oplus c_2$. WRK runs in two high-level phases: the offline phase precomputes and *stores* all abits and aANDs needed later in the protocol, followed by a function-dependent online phase that generates and evaluates an *authenticated* garbled circuit using the abits and aAND prepared earlier.

Next, we give an intuitive tutorial of WRK in the two-party setting but refer to [45] for extending it to the multi-party setting. Let i, j, k be the three wires associated to an AND gate. In two-party setting, to hide the permutation bits $\lambda_i, \lambda_j, \lambda_k$, WRK divides them into XOR-based bit-shares, $[\lambda_i^1]^1, [\lambda_j^1]^1, [\lambda_k^1]^1$ and $[\lambda_i^2]^2, [\lambda_j^2]^2, [\lambda_k^2]^2$, held by P_1 and P_2, respectively, such that $\lambda_i^1 \oplus \lambda_i^2 = \lambda_i, \lambda_j^1 \oplus \lambda_j^2 = \lambda_j, \lambda_k^1 \oplus \lambda_k^2 = \lambda_k$. Now the first question is, to produce the first garbled row of Table 1, how could the circuit generator (call it P_1) compute $(\lambda_i\lambda_j \oplus \lambda_k)\Delta$ without actually knowing $\lambda_i, \lambda_j, \lambda_k$? WRK addresses this challenge by dividing $(\lambda_i\lambda_j \oplus \lambda_k)\Delta$ into two XOR-shares S^{P_1} and S^{P_2} such that $S^{P_1} \oplus S^{P_2} = (\lambda_i\lambda_j \oplus \lambda_k)\Delta$, thus only requiring P_1, P_2 to locally derive S^{P_1}, S^{P_2}, respectively. But then how can the parties *locally* compute S^{P_1}, S^{P_2} from values that they already know? This is exactly where aANDs come handy: if P_1 and P_2 already have the respective shares of an aAND $([a_1]^1 \oplus [a_2]^2)([b_1]^1 \oplus [b_2]^2) = [c_1]^1 \oplus [c_2]^2$ with $a_1 \oplus a_2 = \lambda_i$ and $b_1 \oplus b_2 = \lambda_j$, then P_1, P_2 can learn c_1 and c_2, respectively, with $c_1 \oplus c_2 = \lambda_i\lambda_j$. Consequently, P_1 can compute $(c_1 \oplus \lambda_k^1)\Delta$ from c_1, λ_k^1 and Δ, all

[3]https://github.com/nanoPIMPC/nanoPI.

Table 1: Garbling in Honest-but-curious Adversary Model $\left(\text{Note } H_{b_i,b_j} \overset{\text{def}}{=} \mathsf{H}\left(\mathsf{L}_i^{b_i}, \mathsf{L}_j^{b_j}\right), \forall b_i, b_j \in \{0,1\}.\right)$

$b_i \oplus \lambda_i$	$b_j \oplus \lambda_j$	$b_k \oplus \lambda_k$	Point & Permute	With Random Oracle H	Free-XOR
0	0	$z_{00} = \lambda_i\lambda_j \oplus \lambda_k$	$\mathsf{Enc}_{\mathsf{L}_i^0,\mathsf{L}_j^0}\left(\mathsf{L}_k^{z_{00}}, z_{00}\right)$	$H_{0,0} \oplus (\mathsf{L}_k^{z_{00}}, z_{00})$	$H_{0,0} \oplus (\mathsf{L}_k^0 \oplus z_{00}\Delta, z_{00})$
0	1	$z_{01} = \lambda_i\overline{\lambda}_j \oplus \lambda_k$	$\mathsf{Enc}_{\mathsf{L}_i^0,\mathsf{L}_j^1}\left(\mathsf{L}_k^{z_{01}}, z_{01}\right)$	$H_{0,1} \oplus (\mathsf{L}_k^{z_{01}}, z_{01})$	$H_{0,1} \oplus (\mathsf{L}_k^0 \oplus z_{01}\Delta, z_{01})$
1	0	$z_{10} = \overline{\lambda}_i\lambda_j \oplus \lambda_k$	$\mathsf{Enc}_{\mathsf{L}_i^1,\mathsf{L}_j^0}\left(\mathsf{L}_k^{z_{10}}, z_{10}\right)$	$H_{1,0} \oplus (\mathsf{L}_k^{z_{10}}, z_{10})$	$H_{1,0} \oplus (\mathsf{L}_k^0 \oplus z_{10}\Delta, z_{10})$
1	1	$z_{11} = \overline{\lambda}_i\overline{\lambda}_j \oplus \lambda_k$	$\mathsf{Enc}_{\mathsf{L}_i^1,\mathsf{L}_j^1}\left(\mathsf{L}_k^{z_{11}}, z_{11}\right)$	$H_{1,1} \oplus (\mathsf{L}_k^{z_{11}}, z_{11})$	$H_{1,1} \oplus (\mathsf{L}_k^0 \oplus z_{11}\Delta, z_{11})$

Table 2: WRK's Authenticated Garbling in Malicious Adversary Model $\left(\text{Note } H_{b_i,b_j} \overset{\text{def}}{=} \mathsf{H}\left(\mathsf{L}_i^{b_i}, \mathsf{L}_j^{b_j}\right), \forall b_i, b_j \in \{0,1\}.\right)$

Share of P_1's Garbled Entry		
$H_{0,0} \oplus \Big(\mathsf{L}_k^0 \oplus (c_1 \oplus \lambda_k^1)\Delta \oplus \mathsf{K}[c_2] \oplus \mathsf{K}[\lambda_k^2]$,	$c_1 \oplus \lambda_k^1$,	$\mathsf{M}[c_1] \oplus \mathsf{M}[\lambda_k^1]\Big)$
$H_{0,1} \oplus \Big(\mathsf{L}_k^0 \oplus (c_1 \oplus \lambda_k^1 \oplus \lambda_i^1)\Delta \oplus \mathsf{K}[c_2] \oplus \mathsf{K}[\lambda_k^2] \oplus \mathsf{K}[\lambda_i^2]$,	$c_1 \oplus \lambda_k^1 \oplus \lambda_i^1$,	$\mathsf{M}[c_1] \oplus \mathsf{M}[\lambda_k^1] \oplus \mathsf{M}[\lambda_i^1]\Big)$
$H_{1,0} \oplus \Big(\mathsf{L}_k^0 \oplus (c_1 \oplus \lambda_k^1 \oplus \lambda_j^1)\Delta \oplus \mathsf{K}[c_2] \oplus \mathsf{K}[\lambda_k^2] \oplus \mathsf{K}[\lambda_j^2]$,	$c_1 \oplus \lambda_k^1 \oplus \lambda_j^1$,	$\mathsf{M}[c_1] \oplus \mathsf{M}[\lambda_k^1] \oplus \mathsf{M}[\lambda_j^1]\Big)$
$H_{1,1} \oplus \Big(\mathsf{L}_k^0 \oplus (c_1 \oplus \lambda_k^1 \oplus \lambda_i^1 \oplus \lambda_j^1)\Delta \oplus \mathsf{K}[c_2] \oplus \mathsf{K}[\lambda_k^2] \oplus \mathsf{K}[\lambda_i^2] \oplus \mathsf{K}[\lambda_j^2] \oplus \Delta$,	$c_1 \oplus \lambda_k^1 \oplus \lambda_i^1 \oplus \lambda_j^1$,	$\mathsf{M}[c_1] \oplus \mathsf{M}[\lambda_k^1] \oplus \mathsf{M}[\lambda_i^1] \oplus \mathsf{M}[\lambda_j^1]\Big)$

Share of P_2's Garbled Entry		
$\Big(\mathsf{M}[c_2] \oplus \mathsf{M}[\lambda_k^2]$,	$c_2 \oplus \lambda_k^2$,	$\mathsf{K}[c_1] \oplus \mathsf{K}[\lambda_k^1]\Big)$
$\Big(\mathsf{M}[c_2] \oplus \mathsf{M}[\lambda_k^2] \oplus \mathsf{M}[\lambda_i^2]$,	$c_2 \oplus \lambda_k^2 \oplus \lambda_i^2$,	$\mathsf{K}[c_1] \oplus \mathsf{K}[\lambda_k^1] \oplus \mathsf{K}[\lambda_i^1]\Big)$
$\Big(\mathsf{M}[c_2] \oplus \mathsf{M}[\lambda_k^2] \oplus \mathsf{M}[\lambda_j^2]$,	$c_2 \oplus \lambda_k^2 \oplus \lambda_j^2$,	$\mathsf{K}[c_1] \oplus \mathsf{K}[\lambda_k^1] \oplus \mathsf{K}[\lambda_j^1]\Big)$
$\Big(\mathsf{M}[c_2] \oplus \mathsf{M}[\lambda_k^2] \oplus \mathsf{M}[\lambda_i^2] \oplus \mathsf{M}[\lambda_j^2]$,	$c_2 \oplus \lambda_k^2 \oplus \lambda_i^2 \oplus \lambda_j^2 \oplus 1$,	$\mathsf{K}[c_1] \oplus \mathsf{K}[\lambda_k^1] \oplus \mathsf{K}[\lambda_i^1] \oplus \mathsf{K}[\lambda_j^1]\Big)$

of which P_1 already knows. Because $(c_1 \oplus \lambda_k^1)\Delta \oplus (c_2 \oplus \lambda_k^2)\Delta = (\lambda_i\lambda_j \oplus \lambda_k)\Delta$, one would wish P_2 to be able to locally compute $(c_2 \oplus \lambda_k^2)\Delta$. Unfortunately, P_2 cannot because it does not know Δ.

To resolve this, WRK exploited the fact that P_2 already knows $\mathsf{M}[c_2]$ and $\mathsf{M}[\lambda_k^2]$, both generated from the same Δ_1 (P_1's global secret for authenticating P_2's bits). Because $\mathsf{K}[b] \oplus \mathsf{M}[b] = b\Delta_1$ for any authenticated bit b, it suffices to require P_1 to set $\Delta = \Delta_1$, and define

$$S^{P_1} \overset{\text{def}}{=} (c_1 \oplus \lambda_k^1)\Delta \oplus \mathsf{K}[c_2] \oplus \mathsf{K}[\lambda_k^2]$$

$$S^{P_2} \overset{\text{def}}{=} \mathsf{M}[c_2] \oplus \mathsf{M}[\lambda_k^2]$$

so that $S^{P_1} \oplus S^{P_2} = (\lambda_i\lambda_j \oplus \lambda_k)\Delta$ while the parties can each locally compute S^{P_1} and S^{P_2}, respectively. Finally, to prevent a malicious P_1 from replacing $c_1 \oplus \lambda_k^1$ with an arbitrary bit, WRK requires P_1 to provide $\mathsf{M}[c_1] \oplus \mathsf{M}[\lambda_k^1]$, the MAC of $c_1 \oplus \lambda_k^1$, so that P_2 can verify the correctness of a garbled row. Further, observing that the MACs are XOR-homomorphic and $\lambda_i\overline{\lambda}_j = \lambda_i \oplus \lambda_i\lambda_j$, $\overline{\lambda}_i\lambda_j = \lambda_j \oplus \lambda_i\lambda_j$, and $\overline{\lambda}_i\overline{\lambda}_j = 1 \oplus \lambda_i \oplus \lambda_j \oplus \lambda_i\lambda_j$, so the other three garbled rows can also be computed using the same aAND used for computing the first row (see Table 2).

Note that because the MACs and keys are used in constructing garbled tables, the length of the MACs and keys becomes a computational security parameter.

Generating Authenticated AND Triples. The parties need to run a separate secure protocol in a secret-input-independent preparation phase to generate aAND triples. In fact, this protocol, dubbed Π_{aAND}, dominates the overall cost of the WRK protocols. WRK's Π_{aAND} works in two high-level steps:

(1) Generating leaky-aANDs using Π_{LaAND}. A leaky-aAND triple has the same property as aAND except that a cheating party is able to correctly guess a honest party's first abit output with probability 1/2, at the risk of being caught with probability 1/2.
(2) Combine every B randomly-chosen leaky-aANDs into a fully-secure aAND. The integer B is known as the *bucket size.*

2.3 Pool-based Cut-and-choose

Many state-of-the-art implementations of actively-secure computation protocols [20, 28, 36, 39, 47], including WRK, are based on the idea of *batched cut-and-choose.* These protocols, however, suffered from scalability issues as they require linear storage (in the length of the computation) because much per-gate information needs to be stored *before* the cut-and-choose challenges can be revealed. To overcome this issue, Zhu et al. [48] proposed to maintain a fixed-size pool for keeping necessary information to do cut-and-choose. Namely, the garbled gates used for checking/evaluation will always be selected from a fixed-size pool and the pool will be refilled immediately after any garbled gate is consumed. They have an example instantiation (called Pool-JIMU) of the idea on top of JIMU protocol and showed its extraordinary scalability advantage. In addition, Pool-JIMU also offers unpaired *long-term* statistical security guarantee, i.e., cut-and-choose failures are bounded throughout the *life-time* of the pool regardless of how many secure computation instances have been executed. They have shed some light on applying the idea also on improving the scalability of WRK's expensive aANDs generation protocol. However, they overlooked several important technical issues in combining Pool with WRK. As we will show in Section 4, these oversights can lead to serious performance issues, and even real attacks.

3 THE SPACE-ROUND DILEMMA

Generally speaking, there are two flavors of MPC protocols. Early MPC protocols such as BGW [4] and CCD [7] require constant round *per layer of ANDs*. Protocols in this category may be useful for computing wide but shallow circuits, assuming there is sufficient space to hold the secret values of all the input wires of a layer of ANDs. Note that these protocols could run in constant space, simply at the cost of incurring linear rounds in the size of the circuits. For general computations, due to the prohibitive time cost incurred by the round-trips (especially when network latency is high), constant-round MPC protocols are thought to be much more desirable.

The second category of actively-secure MPC protocols, pioneered by BMR [3] and resurrected by many recent works [11, 12, 27, 44, 45], are able to execute any circuit in constant rounds. However, a largely overlooked drawback of this category of protocols is their space complexity. In fact, we have examined all known MPC protocols in this category but find that they all rely a common online-offline trick to circumvent the round complexity. Namely, they all use a constant-round but linear-space offline preparation phase, followed by a function-dependent online phase. Therefore, their space complexities grow linearly in the running time of the computation.

Table 3 summarizes a list of state-of-the-art actively-secure MPC protocols whose practical efficiency have been verified with actual software implementations. We consider the round, computation, and communication complexity of these protocols in view of their space complexity. Note that all the constant-round, asymptotically more efficient protocols [10, 12, 15, 19, 20, 28, 36, 39, 44, 45, 47] cannot even be launched without $O(|C_f|)$ space to hold the whole garbled circuits, which is typically the case in practice. In contrast, our protocol can always run within a *user-specified* small space $O(p)$. Although ours uses linear rounds, experiments show that, compared with the $O(1)$-round WRK, the actual delay due to roundtrips in our protocol is hardly noticeable even with a relatively small space (Figure 13). Note that the main focus of this work is the more general n-party setting, and the two-party protocols are included to show that this dilemma is somehow fundamental.

Concrete Impact of Space. In order to appreciate the impact of space complexity of WRK protocols, we have run a number of experiments using WRK. Our two-party setting experimental results are depicted in Figure 1. We measured the actual *peak* memory usages of WRK running different circuits whose AND counts range from 35K to 8M, in the two-party setting. Clearly, their memory usage grows linearly with the number of ANDs in the circuit, which aligns well with our theoretical analysis. To get some idea about the peak memory consumption of executing circuits with more than 8M ANDs, we used a linear model $\hat{k} \cdot |C_f| + \hat{b}$ to extrapolate the curve based on 29 points of actual observations, where $\hat{k}, \hat{b}$ are assigned with minimal values indicated by any pair of observed points. Hence the dashed estimation line is an *underestimation* of the actual peak memory usage.

Table 3: Compare state-of-the-art implementations of actively-secure MPC protocols that allow dishonest-majority.

		Space	Round	Comp./Comm. complexity[§]
2-party setting	[22], [1, 25]	$O(1)$	$O(1)^*$	$O(s\|C_f\|)$
	[35]	$O\left(\|C_f\|\right)$	$O(d)$	$O\left(\|C_f\|\left(1+s/\log\|C_f\|\right)\right)$
	[43]	$O(1)$	$O(1)^*$	$O\left(\|sC_f\|\right)$
	[15, 28, 39]	$O(\tau\|C_f\|)$	$O(1)^\dagger$	$O\left(\|C_f\|\left(1+s/\log\tau\right)\right)$
	[10, 20, 36, 47]	$O\left(\|C_f\|\right)$	$O(1)^\dagger$	$O\left(\|C_f\|\left(1+s/\log\|C_f\|\right)\right)$
	[44]	$O\left(\|C_f\|\right)$	$O(1)^\dagger$	$O\left(\|C_f\|\left(1+s/\log\|C_f\|\right)\right)$
	[48]	$O(p)$	$O\left(\|C_f\|/p\right)$	$O\left(\|C_f\|\left(1+s/\log p\right)\right)$
n-party setting	[8]	$O\left(\|C_f\|\right)$	$O(d)$	$O\left(n\|C_f\|\left(1+s/\log\|C_f\|\right)\right)$
	[19]	$O\left(\|C_f\|\right)$	$O(d)$	$O\left(n\|C_f\|\left(1+s/\log\|C_f\|\right)\right)$
	[12]	$O\left(n\|C_f\|\right)$	$O(1)^\dagger$	$O\left(n\|C_f\|\left(n+s/\log\|C_f\|\right)\right)$
	[45]★	$O\left(n\|C_f\|\right)$	$O(1)^\dagger$	$O\left(n\|C_f\|\left(n+s/\log\|C_f\|\right)\right)$
	This work	$O(p)$	$O\left(n\|C_f\|/p\right)$	$O\left(n\|C_f\|\left(n+s/\log p\right)\right)$

$|C_f|$ denotes the circuit size of f. τ is the number of executions in a batch. d, the circuit depth, is application-dependent. p is an (almost) application-independent (except for circuit size) parameter determined by the size of local memory. s is the statistic security parameter.

§ Worst-case complexity of per-party work.

* Asymptotically more expensive and hard to run computations on dynamic data.

† Will fail for many circuits due to its extraordinary space requirement.

★ It has been experimentally shown that, in practice, WRK is substantially more efficient than other MPC protocols due to its small constant factors.

Note that even on a nowadays powerful cloud server with 128GB memory (roughly the `c5.18xlarge` instance at $3/hour on Amazon EC2), WRK would not be able to securely compute the edit distance of two 350-nucleotide genome strings. On less expensive computers like PCs (and resp. mobile devices), WRK is even incapable of executing 20 iterations of SHA256 (resp. 20 iterations of AES). Finally, we stress that this is only in the lighter-weight two-party setting. In a general n-party setting, WRK's peak memory usage will have to be multiplied by a factor of $n-1$ because each party needs to store a abit's key and MAC for every one of the rest $n-1$ parties. Therefore, it is really stretching for this state-of-the-art MPC protocol to compute something practically useful when n increases.

The Impact of Rounds and Inefficacy of Naïve Adaptations. A natural attempt to reduce the space requirement of original WRK is to limit the batch sizes of its abits and leaky-aANDs generation subroutines to some constants so that the resulting WRK-variant can be *streamingly* executed. However, not only will this small change affect protocol correctness and introduce security vulnerabilities (as we will explain in Section 4), the prohibitive overhead incurred

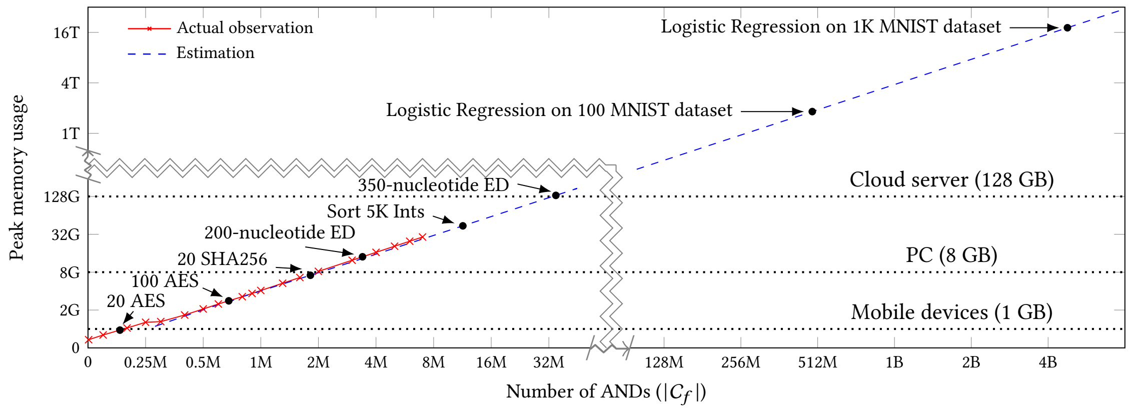

Figure 1: Peak memory usage of two-party WRK protocol. Security parameters: $s = 40$, $\kappa = 128$. The estimation assumes a linear model $\hat{k} \cdot |C_f| + \hat{b}$ for the peak memory usage. We chose to underestimate by setting $\hat{k} := \min_{\substack{\text{for all observed}\\(g_1,m_1),(g_2,m_2)}} \left\{ \frac{m_1 - m_2}{g_1 - g_2} \right\}$ and $\hat{b} := \min_{\substack{\text{for all observed}\\(g_1,m_1),(g_2,m_2)}} \left\{ \left| m_1 - \frac{m_1 - m_2}{g_1 - g_2} g_1 \right| \right\}$.

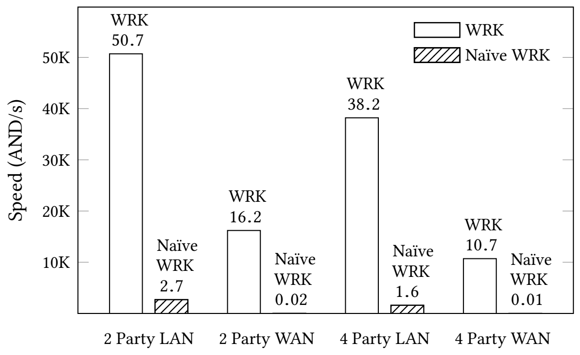

Figure 2: Performane degradation of naïvely adapted WRK Assume $s = 40$, $\kappa = 128$. Speeds measured on running an random circuit mixing 25% ANDs and 75% of XORs. The bucket size is 3. The network latency of LAN and WAN are 0.2ms and 40ms, respectively.

by the roundtrips also renders the performance of the streamed protocol execution pointless.

In order to quantify the performance penalty, we have tried out this naïve modification to WRK and experimentally measured its performance in various setups. As Figure 2 shows, the naïvely converted WRK variant runs more than 20x slower than original WRK even in the low-latency LAN setting. In a WAN network, the factor of slowdown increases to 800–1000, which makes it practically unusable!

Finally, although we picked WRK as the baseline in our case-study both because of its performance advantage over its peer protocols and also the availability of implementation, the fundamental conflict between the round and space complexities of malicious MPC protocols is present in all known MPC protocols. The community has yet to see any actively-secure MPC implementation that is able to run extremely large circuits at a reasonably fast speed. Thus, new secure mechanisms are needed to better reconcile the conflicts between space and round in MPC protocol design.

4 DIAGNOSIS AND APPROACH OVERVIEW

In this section, we first analyze the root causes of WRK's space-round dilemma. Then, we sketch our ideas to address the challenge, even under very stringent space/time budgets.

For presentation clarity, we restrict our discussion to the two-party setting, but the ideas can naturally be carried over to the multi-party setting.

4.1 Root Causes

Root cause 1: The **abit** *protocol.* The way in which abits are generated and used in WRK requires $O(|C_f|)$ space: (1) Note that **abit** is realized using OT extension, which requires constant rounds per batch. Therefore, computing $O(|C_f|)$ abits within constant rounds implies using $O(|C_f|)$ space so that $O(|C_f|)$ OTs can run in parallel. (2) The call to **abit** in step (2) of protocol Π_{2pc} (Figure 14) assumes that the $O(|C_f|)$ tuples returned by **abit** are all stored in the program by both parties.

Root cause 2: The **aAND** *protocol.* For similar reasons as above, the way **aAND** is generated and used in WRK also seriously limits its scalability: (1) WRK's aANDs generation protocol Π_{aAND} is constant-round but uses $O(C_f)$-space. (2) Even if WRK's Π_{aAND} was efficiently executable in constant-space, the way **aAND** gets used in WRK's Π_{2pc} also prevents Π_{2pc} from running in constant-space. This second issue happens to be occluded by a presentation flaw in their papers [44, 45].

Take their two-party authenticated garbling Π_{2pc} [44] as an example. The ideal **aAND** of $\mathcal{F}_{Pre}$ used by Π_{2pc} (which we excerpted from [44] as Figure 3) is actually different from their $\mathcal{F}_{aAND}$ functionality (which we copied below for easy comparison):

$\mathcal{F}_{aAND}$

Honest case: Generate uniform $[r_1]^1$, $[r_2]^1$, $[r_3]^1$, and $[s_1]^2$, $[s_2]^2$, $[s_3]^2$, such that $(r_1 \oplus s_1) \wedge (r_2 \oplus s_2) = r_3 \oplus s_3$.
Corrupted parties: A corrupted party gets to specify the randomness used on its behalf by the functionality.

The gap is that: $\mathcal{F}_{aAND}$ does not allow the participants to control the values of the abits, whereas the **aAND** of $\mathcal{F}_{Pre}$ does allow P_1

$\mathcal{F}_{\mathsf{Pre}}(\kappa)$

init: Upon receiving **init** from P_2 and Δ_1 from P_1, choose uniform $\Delta_2 \in \{0,1\}^\kappa$ and store Δ_1, Δ_2. Send Δ_2 to P_2.

abit: Upon receiving **abit** from both P_1 and P_2, sample uniform $r, s \in \{0, 1\}$, uniform $\mathsf{K}[r], \mathsf{M}[r], \mathsf{K}[s], \mathsf{M}[s] \in \{0,1\}^\kappa$ such that $\mathsf{K}[r] = \mathsf{M}[r] \oplus r\Delta_B$ and $\mathsf{M}[s] = \mathsf{K}[s] \oplus s\Delta_1$. Send $(s, \mathsf{M}[s], \mathsf{K}[r])$ to P_2.

aAND: Upon receiving (**aAND**, $(r_1, \mathsf{M}[r_1], \mathsf{K}[s_1])$, $(r_2, \mathsf{M}[r_2], \mathsf{K}[s_2])$, $(r_3, \mathsf{M}[r_3], \mathsf{K}[s_3])$) from P_1 and (**aAND**, $(s_1, \mathsf{M}[s_1], \mathsf{K}[r_1])$, $(s_2, \mathsf{M}[s_2], \mathsf{K}[s_2])$) from P_2, verify that $\mathsf{M}[r_i] = \mathsf{K}[r_i] \oplus r_i\Delta_2$ and that $\mathsf{M}[s_i] = \mathsf{K}[s_i] \oplus s_i\Delta_1$ for $i \in \{1, 2\}$ and send cheat to P_1 and P_2 if not. Otherwise, set $s_3 := r_3 \oplus (r_1 \oplus s_1)(r_2 \oplus s_2)$, set $\mathsf{K}[r_3] := \mathsf{M}[r_3] \oplus r_3\Delta_2$ and $\mathsf{M}[s_3] := \mathsf{K}[s_3] \oplus s_3\Delta_1$. Send $(s_3, \mathsf{M}[s_3], \mathsf{K}[r_3])$ to P_2.

Figure 3: The two-party $\mathcal{F}_{\mathsf{Pre}}$ functionality. (κ is a computational security parameter.) (excerpted from [44])

(resp. P_2) to specify the abit-values r_1, r_2 (resp. s_1, s_2). WRK's Π_{aAND} only realizes $\mathcal{F}_{\mathsf{aAND}}$ but Π_{2pc} actually depends on the extra control offered by $\mathcal{F}_{\mathsf{Pre}}$'s **aAND** so that its abit generation can be treated as function-independent offline work! The same presentation flaw also appeared in the multi-party version of WRK [45].

This issue can be fixed by introducing an extra round for the parties to align the random abits returned by $\mathcal{F}_{\mathsf{abit}}$ and $\mathcal{F}_{\mathsf{aAND}}$. We looked into their source code and verified that this is what actually happens in their software implementation. We stress, however, that this extra treatment won't affect the round complexity of WRK only if sufficient space is available to cache the abits associated with all the wires and AND triples so that all the messages can be sent in the same round. In practice, since space is eventually limited by some constant, Π_{2pc} has to use $O(|C_f|)$ rounds.

Root cause 3: The function-dependent protocol Π_{2pc}*.* If a wire splits into multiple wires (which are used as input to different AND gates), then the same authenticated permutation bit has to be used to compute all those AND gates involving the split wires. This implies that Π_{2pc} needs to be aware of all wire-connection information of the circuit, which depends on the function and even the specific ways to construct the function. WRK obtains this wire-connection information by fully unroll the function, which is not feasible when $|C_f|$ is large. If only partial information of wire-connections is known, as is the case when Pool-JIMU executes programs, the naïve Π_{2pc} must keep every wire around just in case some are later found split into other wires. This is clearly against the idea of space-efficient streamed execution.

Concrete Space Analysis. Assume 128-bit computational security, every party needs to store at least a triple $(\mathsf{L}_\gamma, \mathsf{M}[\lambda^1_\gamma], \mathsf{K}[\lambda^2_\gamma])$ per wire (which is $16 \times 3 = 48$ bytes) and three authenticated bits (which is $32 \times 3 = 96$ bytes) per leaky-aAND. Because each AND defines a new wire and needs an nonleaky-aAND to compute, so at least $48 + 96B$ bytes memory are needed for each AND where B is the bucket size. In addition, to efficiently implement Π_{abit} and Π_{aAND}, it is important to batch-run sufficiently many AESNI instructions, which demands additional large contiguous memory to pack the data to run through the AES cipher.

In the general n-party setting, each garbler has to store roughly $16+32(n-1)+96B(n-1)$ bytes per AND: 16 bytes for the wire-label L_γ, $32(n-1)$ bytes for wire permutation abit share, and $96B(n-1)$ for $3B$ MACs and $3B$ keys (but the length of each M or K now has expanded $n-1$ times). The evaluator has to store $(48+96B)(n-1)$ bytes per AND because $n-1$ wire-labels per wire is needed at the time of gate evaluation.

4.2 New Challenges and Key Solution Ideas

One would naturally think carrying the pool idea [48] over here to WRK will resolve the space-round issue, just like it did to JIMU [47]. Unfortunately, this is not the case. In fact, efficiently running WRK protocols with limited space requires not only cryptographic enhancements, but also some new programming language support that no existing MPC frameworks has offered. Next, we overview the new challenges and our key ideas to address them.

4.2.1 *The Authenticated Garbling Phase.*

WRK requires aligning aANDs to their corresponding wire-permutation-bits. In combination with the wire management problem, this poses new scalability challenges. We address these challenges through novel combination of static and dynamic program instrumentation techniques, which we find most interesting.

The high level idea is that, given a computation specified as an imperative program, we first apply a source-to-source transformation to insert appropriate function calls for wire management and *staged* gate execution. As a result, when the statically instrumented program runs, it behaves like running the original WRK except that it also collects runtime information (such as wire-connection, gate and wire counts, etc. that are not available at compile-time) to automatically batch gate execution in stages.

Aligning aANDs with Wire-permutation-bits. Let α, β be the two input-wires of an AND gate and $\lambda_\alpha, \lambda_\beta$ be the permutation bits on the input-wires. Since the aANDs (used to garble AND gates) and the abits (used as wire-permutation-bits) are precomputed independently, in step (4a) of Figure 14, we must ensure the values of the secretly-shared permutation bits $\lambda_\alpha, \lambda_\beta$ are consistent with those returned by $\mathcal{F}_{\mathsf{aAND}}$. (We invite the readers to read Section 2.2 for reasons why the bits need to be consistent.) As was explained in Section 4.1, this incurs an extra round. To alleviate the impact of round latency due to the alignment, it is important to batch many AND gates together to share a single network roundtrip. On the other hand, the space available to store the per-gate information needed for garbling will limit the number of ANDs to be processed in a batch. Finally, this alignment process has to be function-dependent to properly handle wire-splits. (In contrast, Pool-JIMU has a constant-round wire-soldering step but can be trivially done in a function-independent way, thus don't require the advanced programming techniques as WRK protocols do.)

Managing the Wires. Due to space constraint, it is infeasible to fully unroll a program to obtain all the intermediate wires and their connection information. However, WRK's online circuit phase does need information of *all* wires to complete efficiently in constant rounds. Our goal here is to run WRK in constant space without overly penalize speed. But is it possible to finish a long computation while keeping only constantly many wires?

A key idea to answer this question is to leverage the program representation of the computation. In practice, useful computations almost always have a constant-size representation no matter how long they need to run. In fact, this idea was used by Zhu et al. to reduce its space complexity: Pool-JIMU only maintains a small set of wires corresponding to the set of program variables visible at current point of runtime execution. There, wires are created (and destructed, resp.) as the execution enters (and exits) their corresponding variables' defining scopes.

Unfortunately, this idea does not directly work with WRK. Take the following simple function (1-bit multiplexer) as an example.

```
mux1(x, y, c) {
    t := x ⊕ y;
    t := c ∧ t;
    return t ⊕ y;
}
```

Should the old strategy be used, several issues arise, evidencing new programming challenges:

(1) A scope can be small, e.g., function `mux1`'s scope contains only two binary gates. If the wires corresponding to variables `t`, `c`, etc. are to be destructed on exiting their scope, then the AND gates inside the scope have to be executed (before its input-wires are destructed), thus incurring constant rounds *per scope-exit*. Therefore, for small scopes, freeing up wires upon exiting their defining scopes limits the batch size and incurs more rounds.
(2) Some statements, like "`t := c ∧ t`" in the example above, cannot be directly supported without additional treatment. This is because the execution of some ANDs has to be delayed (so that they are batch-executed with other ANDs). Therefore, the output of an AND gate cannot overwrite input-wires of any ANDs (including its own) whose execution is currently delayed.
(3) Copy assignments like "`x := y`" cannot be executed as usual as in JIMU-Pool: the wire associated with variable `y` may be destructed at some point of exiting its scope whereas the variable `x` may have a longer lifespan, e.g., when `x` is not a local variable but `y` is a local variable.
(4) General compositions of binary operations such as "`x ∧ y ∧ z`" and "`x ⊕ y ∧ z`" cannot be directly executed as with Pool-JIMU, simply because gate execution are delayed and batched, and that in-place wire assignment is not possible.

Our Solution. We propose a novel program execution model that addresses these new challenges. Comparing to the normal stack-based program execution, our new execution model preserves the final outcome and space-efficiency, but circumvents the inability of normal stack-based model in efficiently supporting WRK. The high-level key ideas can be informally described as a list of rules:

(1) Every gate will run after certain delay. That is, a gate is processed as pushing the gate into a queue, marking it *ready* to execute once the next batch (called *stage*) is triggered.
(2) A stage is automatically and *dynamically* triggered when there is not enough memory to batch more gates.
(3) Every wire will be marked as *destructable* (but not actually destructed) when exiting its static scope. The actual destruction occurs automatically after executing every stage.
(4) An assignment always implicitly creates a new wire, and binds the target variable with the new wire.
(5) Before each assignment, the previous wire associated with the target variable needs also to be marked as destructable, while the actual destruction happens only after the current stage is executed.
(6) All expressions must be translated to *three-address assignments*.

Why would these transformation rules resolve our challenge? And even if they do, wouldn't it be cumbersome and error-prone for programmers to manually enforce them for every specific application program? We answer both questions by developing a static program rewriter (as a standalone executable) and a dynamic program instrumenter (as a collection of functions to be linked with user's application code) for a subset of C, then formally prove that executions of the transformed programs must produce identical outcome as the original programs but only consume small space. Out of the six rules above, rule (1) and rule (6) are implemented by the static rewriter; rule (2) is implemented by the dynamic instrumenter; and rule (3), rule (4), rule (5) are jointly realized by both. Leveraging loops and recursion, our language is capable of specifying many useful boolean circuits such as AES, edit distance and logistical regression in a highly compact way. As a result, our toolchain is able to completely automate the ideas to efficiently run WRK for circuits of arbitrary size.

4.2.2 *The Preparation Phase.*

As we explained earlier, both Π_{abit} and Π_{aAND} in the preparation phase are plagued by the space-round dilemma. We first show that naïvely converting WRK's Π_{abit} to a space-efficient variant can introduce security vulnerabilities. Then we explain ideas to generate abits in an efficient and scalable way without compromising security.

Attacking Naïvely Scaled Π_{abit}. The basic idea to scale up Π_{abit} is to break the constant-round, single-batch of ℓ abits generation process into ℓ/k batches, each producing k abits (k is set based on available resource). However, realizing this by naïvely calling WRK's Π_{abit} repetitively invites security attacks. Recall that in WRK abits are generated using *random correlated oblivious transfers* (RCOT), a sub-protocol that takes no input from the parties and returns P_1 (the RCOT sender) ℓ random correlated message pairs $\left\{\left(m_0^i, m_1^i\right)\right\}_{1\le i\le \ell}$ so that $\exists\Delta \in \{0,1\}^n, \forall i, m_0^i \oplus m_1^i = \Delta$; and returns P_2 (the RCOT receiver) $\left\{\left(b, m_b^i\right)\right\}_{1\le i\le \ell}$. Note that there is no guarantee that the same Δ should be output across different RCOT calls. Hence, WRK's proof of security no longer applies. Even worse, this can actually leave the main secure computation protocol vulnerable to, what we call, *inconsistent-*Δ *attacks*!

To see how an inconsistent-Δ attack works, note that with all but negligible probability, P_1 as an RCOT sender will get two values of the correlation-difference, say Δ and Δ', from two calls to RCOT protocol. So with high probability, in some iterations of step (4a), the abit r_α of a particular aAND is authenticated with Δ', whereas the rest of the abits involved in the same **aAND** call are authenticated with Δ. By definition, it is easy to verify that **aAND** will fail if $r_\alpha = 1$ but succeed if $r_\alpha = 0$. Since everyone knows which abits are associated with which Δ values, just by observing whether the execution fails or not, an attacker learns an abit of its peer's. Since a leaked abit can associate with the permutation bit corresponding to an input-wire carrying another party's secret input, that party's secret input bit will be leaked by observing if a protocol execution fails!

Secure, Scalable $\Pi_{\text{abit}}^{\text{Scalable}}$. To avoid the inconsistent-Δ issue, we propose two minor modifications to WRK's Π_{abit}: (1) we change the underlying RCOT protocol so that it takes a predefined Δ as input from the sender which is used to form random correlated messages; and (2) to prevent an adversary from deliberately using different Δ in different RCOT batches, we add a consistency check step at the end of *every* batch of RCOTs to ensure that identical Δ values are used across different batches. Our abit protocol, called $\Pi_{\text{abit}}^{\text{Scalable}}$, can efficiently produce an arbitrary number of abits using constant-space. $\Pi_{\text{abit}}^{\text{Scalable}}$ is specified in Figure 8.

Fast, Scalable $\Pi_{\text{aAND}}^{\text{Scalable}}$. We improve the memory-scalability of cut-and-choosing leaky-AND triples through maintaining a fixed-size pool of leaky-AND triples. Instead of storing $O(|C_f| \cdot B)$ leaky-aANDs before randomly grouping every B leaky-aANDs into a bucket, we can carry out the random grouping *always* within a pool of p leaky-aANDs while refilling a used leaky-aAND as soon as it is marked to be converted into a fully-secure AND triple. Thus, WRK's cut-and-choose-based Π_{aAND} is modified to efficiently run in constant space using pool-based cut-and-choose. Comparing to Pool-JIMU, the differences here are: (1) every leaky-aAND is checked for validity with a constant fault-detection rate 1/2, and (2) every leaky-aAND can be checked *and* combined with other leaky-aANDs to form a fully-secure aAND. As a result listed in [48, Table 6], it suffices to maintain a pool of 479K leaky-AND triples to achieve bucket size 3. (Note that the smaller the buckets are, the faster WRK's Π_{aAND} runs.) Our aAND protocol $\Pi_{\text{aAND}}^{\text{Scalable}}$ can efficiently generate any number of aANDs using constant-space. $\Pi_{\text{aAND}}^{\text{Scalable}}$ is specified in Figure 9.

4.3 Putting It All Together

We integrated the cryptographic enhancements and PL techniques mentioned above and provide a complete toolchain, nanoPI, for non-crypto-experts to develop and execute long-term or extreme-scale, actively-secure MPC protocols. The high-level workflow of nanoPI is depicted in Figure 4. Our system consists of the following components:

- **A Static Rewriter.** It instruments circuit functions written by application developers with resource management APIs provided by our backend. In essence, the static writer extracts program's static scope information and passes it to our backend interpreter to allow improved resource management.
- **Cryptographic functions.** They realize our improved efficiently scalable variant of WRK components such as functions `abitGen`, `aANDGen`, and `runCircuit`, etc.
- **Stage functions.** These functions are responsible to automatically arrange the gates into different stages, securely execute the gates respecting their topological order, retain necessary resource for intermediate values to connect the stages while recycling others as soon as possible.
- **Basic Circuits Library.** This is a set of basic circuits frequently used in building real world applications. We follow the common practice of existing secure computation frameworks [14, 29, 40, 42] which included optimized version of basic circuits to facilitate non-crypto-expert application developers.

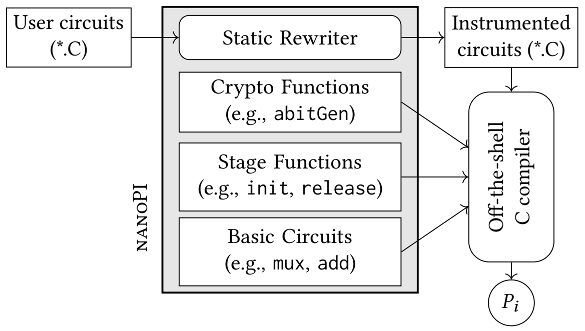
User circuits (*.C)
Static Rewriter
Instrumented circuits (*.C)
Crypto Functions (e.g., abitGen)
Stage Functions (e.g., init, release)
Basic Circuits (e.g., mux, add)
nanoPI
Off-the-shell C compiler
P_i

Figure 4: The overview of nanoPI's workflow.

5 PROTOCOL DETAILS

We divide the formal description of our approach into smaller pieces to facilitate the proof of security. Section 5.1 describes the enabling PL techniques for efficiently scaling up WRK's online phase. Section 5.3 presents techniques for space-efficient generation of abits and aANDs. While our description focuses on the two-party setting, the ideas naturally generalize to multi-party settings (see Section 6).

5.1 Scalable Authenticated Garbling

We use an idealized subset of C as a compact circuit description language. We develop (and prove the correctness of) a new program execution semantics for this subset of C that models the space requirements of WRK's authenticated garbling algorithm (Π_{2pc}) without overly penalizing its performance due to network round-trips. As a proof-of-concept, we formalize the subset-of-C circuit description language using the grammar below:

Programs	p	::=	$d_1; d_2; \ldots; F_1; F_2; \ldots; b$
Variable Declarations	d	::=	`bit` x | `bit`$[n]$ x
Function Declarations	F	::=	$f(d_1, d_2, \ldots)\{b\}$
Blocks	b	::=	$d_1; d_2; \ldots; s_1; s_2; \ldots$
Circuit Variables	X	::=	x | $x[i]$
Iteration Variables	i	::=	0 | 1 | 2 . . .
Operators	$\odot$	::=	`nand` | `nor` | . . .
Statements	s	::=	X `=` `true` | X `=` `false`
			| X `=` $X_1 \odot X_2$
			| `repeat` i $[0..n]$ $s_1; s_2; \ldots$
			| $f(X_1, X_2, \ldots)$

A *program* in this language is a sequence of (global) variable declarations $d_1, d_2, \ldots$ followed by a sequence of function declarations $F_1, F_2, \ldots$ followed by a block. A *block* is itself a sequence of (local) variable declarations followed by a sequence of statements. Functions are only called for effect in this language. There are two types of values: bits and arrays of bits of a fixed size. In order to be able to concisely describe circuits with common structure, the language includes a limited form of iteration whose bounds must be known at compile time and arrays to refer to homogeneous collections of wires. We will use the following program as a small running example in this section.

```
bit r;         // We assume r, rs[0], rs[1], xs[0],
bit[2] rs;     // and xs[1] were all initialized and
bit[2] xs;     // store secret values.

f (bit x) { bit t; t = t ∧ x; r = r ∧ t;}

repeat i [0..2] { f (xs[i], xs[i]); rs[i] = r;}
```

Canonical Semantics. A well-established method for specifying the formal semantics of languages like our subset of C is via a set-theoretic denotational model [46]. The details of such a denotational model can be dramatically simplified if we pre-process programs to eliminate convenient-for-use but semantically distracting features. In our case, we use well-understood correctness-preserving program transformations (see for example [2]) to (i) unroll all loops, (ii) inline all functions, and (iii) replace all arrays by collections of scalars. This preprocessing step terminates in our setting because the source program is assumed to describe a finite circuit. The syntax of the resulting *scoped* circuit description language can thus be simplified to the following:

Programs	p	::=	b
Blocks	b	::=	$d_1; d_2; \ldots; s_1; s_2; \ldots$
Variable Declarations	d	::=	$\texttt{bit}\ x$
Statements	s	::=	$x\ \texttt{=}\ \texttt{true} \mid x\ \texttt{=}\ \texttt{false}$
			$\mid x\ \texttt{=}\ x_1 \odot x_2 \mid \{b\}$

Programs in this core language are *almost* straight-line programs corresponding to a sequential composition of gates, except that we retain nested lexical scope relations. This is important for reasoning about space allocation and de-allocation as explained next. Our running example expands to the following:

```
bit r;
bit rs0; bit rs1;
bit xs0; bit xs1;
{ // first call to f
  bit x;
  x = xs0;
  bit t;
  t = t ∧ x;
  r = r ∧ t;
}
rs0 = r;
{ // second call to f
  bit x;
  x = xs1;
  bit t;
  t = t ∧ x;
  r = r ∧ t;
}
rs1 = r;
```

We stress that because we retain scope information, only one set of declarations for x, y, and t exists at any particular time irrespective of how many times the original loop was iterated. Similarly, only one instance of the local variables of a function will be live irrespective of how many times the original function was inlined.

The denotational semantics for this core language uses conventional tools (e.g, [46]) described next. We first define the auxiliary notions of an *environment* ρ and a *store* σ. An environment maps variables to (heap) locations and a store maps these locations to values. An environment ρ can be extended with a new entry $x \longleftarrow \ell$ to produce a new environment $\rho[x \longleftarrow \ell]$. We note that, reflecting the usual rules of static scoping, the original ρ may already have an entry for x and that entry is *shadowed* by the new binding. In other words, the rule for looking up the current binding for x in an environment is defined as follows:

$$(\rho[x \longleftarrow \ell])(y) = \begin{cases} \ell & \text{if } x = y \\ \rho(y) & \text{otherwise} \end{cases}$$

which resolves a variable lookup to the statically innermost declaration. The store σ is a mapping from locations to values, but it *never* contains duplicate locations in its domain. Thus for stores, the notation $\sigma[\ell \longleftarrow v]$ creates a location ℓ if ℓ is fresh in σ and otherwise performs an in-place update of the contents of ℓ.

The semantics of a declaration in the scoped language is modeled as a transformation that takes a current environment and store, and returns an extended environment in which the new variable is bound to a fresh location that is initialized to `false` in the store:

$$\mathcal{D}[\![\texttt{bit}\ x]\!]\langle\rho,\sigma\rangle \quad = \quad \langle\rho[x \longleftarrow \ell], \sigma[\ell \longleftarrow \texttt{false}]\rangle \qquad \text{where } \ell \text{ is fresh in } \sigma$$

To calculate the environment and store for a sequence of declarations, we simply apply the function $\mathcal{D}[\![\cdot]\!]$ to each declaration in turn, passing the resulting environment and store from one declaration to the next. Formally:

$$\overline{\mathcal{D}}[\![d; ds]\!]\langle\rho,\sigma\rangle \quad = \quad \overline{\mathcal{D}}[\![ds]\!](\mathcal{D}[\![d]\!]\langle\rho,\sigma\rangle)$$

Each statement receives an environment and a store and computes a (possibly modified) store that is propagated to the following statement. The result of the program is the final store. Formally, the semantics of blocks and statements is defined using two mutually recursive functions $\mathcal{B}[\![\cdot]\!]$ and $\mathcal{S}[\![\cdot]\!]$ defined below. Each of these functions takes a current environment and store and returns only a store.

$$\begin{aligned}
\mathcal{B}[\![ds; ss]\!]\langle\rho,\sigma\rangle &= \overline{\mathcal{S}}[\![ss]\!](\overline{\mathcal{D}}[\![ds]\!]\langle\rho,\sigma\rangle) \\
\overline{\mathcal{S}}[\![s; ss]\!]\langle\rho,\sigma\rangle &= \overline{\mathcal{S}}[\![ss]\!]\langle\rho, \mathcal{S}[\![s]\!]\langle\rho,\sigma\rangle\rangle \\
\mathcal{S}[\![x\ \texttt{=}\ \texttt{true}]\!]\langle\rho,\sigma\rangle &= \sigma[\rho(x) \longleftarrow \texttt{true}] \\
\mathcal{S}[\![x\ \texttt{=}\ \texttt{false}]\!]\langle\rho,\sigma\rangle &= \sigma[\rho(x) \longleftarrow \texttt{false}] \\
\mathcal{S}[\![x\ \texttt{=}\ y \odot z]\!]\langle\rho,\sigma\rangle &= \sigma[\rho(x) \longleftarrow \sigma(\rho(y)) \odot \sigma(\rho(z))] \\
\mathcal{S}[\![\{b\}]\!]\langle\rho,\sigma\rangle &= \mathcal{B}[\![b]\!]\langle\rho,\sigma\rangle
\end{aligned}$$

The maximum size of the environment is the maximum number of locations that are simultaneously live in one static scope. In our running example, when we are executing inside either call for `f`, the environment contains entries for `r, rs0, rs1, xs0, xs1, ys0, ys1, x, y, t`. Upon exiting the scope, the environment reverts to only having entries for `r, rs0, rs1, xs0, xs1, ys0, ys1`. More explicitly, environments provide a formalization of the notion of *live locations*: only locations reachable through the environment are live and all other locations can be recycled.

Staged Semantics. The semantics above models a standard execution in which statements are executed sequentially, entering and exiting scopes as determined by the syntactic structure of the program. This semantics can be adapted to reflect the staged execution model that we propose for executing WRK's authenticated garbling protocol Π_{2pc}, since it adopts a more liberal execution schedule which allows *delayed* and *batched* execution of primitive operations, *even if these operations occur in different scopes.* To model such a semantics, we proceed in three stages.

First, we define a source-to-source translation that replaces every assignment by a new declaration, thus allocating a new heap location instead of in-place updating the existing one. Intuitively this allows the history of values assigned to a variable to co-exist simultaneously, modeling rule (4) of our intuitive solution outlined in Section 4.2. In detail, this translation replaces every assignment "`x = v`" by a declaration "`bit x`" that is immediately followed by an assignment "`x = v`". For convenience, we abbreviate this combination as "`bit x = v`". The situation for non-local variables is more subtle: updates to fresh locations originating from a non-local variable need to be stored back in the non-local variable upon exiting a local scope. This is done by appropriately inserting "`update`" instructions whenever exiting a scope. The example code from page 9 will thus be translated into:

```
1   bit r;
2   bit rs0; bit rs1;
3   bit xs0; bit xs1;
4   { // first call to f
5     bit x = xs0;
6     bit t;
7     bit t = t ∧ x;
8     non_local bit r = r ∧ t;
9   }
10  update r;
11  bit rs0 = r;
12  { // second call to f
13    bit x = xs1;
14    bit t;
15    bit t = t ∧ x;
16    non_local bit r = r ∧ t;
17  }
18  update r;
19  bit rs1 = r;
```

Note that this is no longer a legal program in our subset-of-C language but can be viewed as a program in an intermediate language where, e.g., having multiple declarations for the same variable within a scope causes no problem. The repeated declarations of `t` in the same scope just resolve to different entries in the environment which is harmless as each subsequent use of `t` sees the previous declaration and `t` is only visible within the inner scope. For `r`, the new declaration in the inner scope would update the local copy with the value of `r ∧ t` but this update would be lost as soon as we exit the inner scope. For that reason, the translation adds a special label to declarations arising from assignments to non-local variable declarations like the declaration for `r`; as we explain next, the special label is used to maintain a connection, called η, between the local and non-local instances of `r`. Specifically, upon exiting the inner scope, an `update` operation uses η to copy the contents of the local instance of `r` back into the non-local instance. See below for more explanation about η.

Second, we split the semantic definition of staged execution into two functions that communicate through a shared queue containing delayed *thunks* [16] or *closures* [23]. One function (see Figure 5) traverses the program as usual but instead of executing expressions like "$\texttt{bit}\ x = y\ \odot\ z$" immediately, it instead allocates a fresh location ℓ, extends the environment with a binding of the declared variable to ℓ, extends the store with a marker • indicating that ℓ is currently uninitialized, constructs a closure $\ell \leftarrow \rho(y)\ \odot\ \rho(z)$ containing the locations referenced by y and z, and pushes this closure on the shared queue. The other function (see Figure 6) traverses the queue of closures and executes asynchronously. In order to maintain a global order of execution that respects the original data dependencies, an additional data structure η is maintained to connect local and non-local instances of the same variable. Note that updates to η are *in-place*, namely, "$\eta[x\backslash\ell]$" denotes updating the location where the non-local variable x's latest value is stored to ℓ. The `update` operations introduced in the previous stage use this η association to push the proper updates on the shared queue so that future references to non-local locations see the latest value assigned within the inner scope. As shown in Figure 5, η is updated when processing declarations (by $C_d[\![\cdot]\!]$) and used for interpreting "`update`" instructions (by $C_s[\![\cdot]\!]$).

For example, a staged execution of the above running example could result in a series of configurations given in Table 4. Focus on the part of the execution surrounding and including the first inner scope. Initially, the declarations simply extend the environment with fresh locations which are uninitialized. At line 7, we enter a new scope and extend the environment with a new location `0x1014` for `x`. But instead of immediately updating the contents of `0x1014`, we push a closure that can do the update asynchronously when it is invoked. Execution continues, creating fresh locations, and delaying updates to the store. Note that on line 10, the reference to `t` resolves to the latest allocated location `0x101c` for `t`. Once we exit the scope at line 11, the environment reverts back to ρ_5 which was the environment before entering the local scope. In the next line, we communicate the latest value for `r` which was stored in `0x1020` to the non-local location `0x1000`. Table 4 illustrates a situation where the closures queue reaches its critical size after executing line 16 of the above translated program. At that point, all the pending updates are performed and the resulting store will be identical to the one reached by a canonical execution.

For an intuitive argument of correctness of this part of semantics, let's take the conventional semantics of assignments:

$$\mathcal{S}[\![x\ =\ \texttt{true}]\!]\langle\rho,\sigma\rangle = \sigma[\rho(x) \leftarrow \texttt{true}]$$

and compare it with the new staged semantics for assignments:

$$C_s[\![x\ =\ \texttt{true}]\!]\langle\rho,\sigma,\eta,\mathcal{L}\rangle = \langle\sigma,\eta,\mathcal{L} + \{\rho(x) \leftarrow \texttt{true}\}\rangle$$

We see that the only difference is that the update $\rho(x) \leftarrow \texttt{true}$ is associated with $\mathcal{L}$ instead of σ. The situation is similar for declarations: the original semantics is:

$$\mathcal{D}[\![\texttt{bit}\ x = \texttt{true}]\!]\langle\rho,\sigma\rangle = \langle\rho[x \leftarrow \ell],\sigma[\ell \leftarrow \texttt{true}]\rangle$$

and the new staged semantics is:

$$\begin{aligned} &C_d[\![\texttt{bit}\ x = \texttt{true}]\!]\langle\rho,\sigma,\eta,\mathcal{L}\rangle = \\ &\qquad\langle\rho[x \leftarrow \ell],\sigma[\ell \leftarrow \bullet],\eta,\mathcal{L} + \{\ell \leftarrow \texttt{true}\}\rangle \end{aligned}$$

where ℓ is fresh in σ in both cases. Again the difference is that the update $\ell \leftarrow \texttt{true}$ is associated with $\mathcal{L}$ instead of σ.

These examples illustrate that the staged semantics preserves the atomic actions of the canonical semantics but we have not guaranteed yet that these actions will happen in the same order. To enforce this, we insist that the actions in $\mathcal{L}$ are performed in the original order, i.e., by treating $\mathcal{L}$ as an FIFO queue. Semantically, we formalize this by always inserting new closures to the right in Figure 5 and extracting them from the left in Figure 6. In the latter

$$
\begin{array}{rcll}
C_s[\![x\ =\ \texttt{true}]\!]\langle \rho,\sigma,\eta,\mathcal{L}\rangle & = & \langle \sigma,\eta,\mathcal{L}+\{\rho(x)\leftarrow \texttt{true}\}\rangle & \\
C_s[\![x\ =\ \texttt{false}]\!]\langle \rho,\sigma,\eta,\mathcal{L}\rangle & = & \langle \sigma,\eta,\mathcal{L}+\{\rho(x)\leftarrow \texttt{false}\}\rangle & \\
C_s[\![x\ =\ y\ \odot\ z]\!]\langle \rho,\sigma,\eta,\mathcal{L}\rangle & = & \langle \sigma,\eta,\mathcal{L}+\{\rho(x)\leftarrow \rho(y)\odot\rho(z)\}\rangle & \\
C_s[\![\texttt{update}\ x]\!]\langle \rho,\sigma,\eta,\mathcal{L}\rangle & = & \langle \sigma,\eta,\mathcal{L}+\{\rho(x)\leftarrow \eta(x)\}\rangle & \\
C_d[\![\texttt{bit}\ x\ =\ \texttt{true}]\!]\langle \rho,\sigma,\eta,\mathcal{L}\rangle & = & \langle \rho[x\leftarrow\ell],\sigma[\ell\leftarrow\bullet],\eta,\mathcal{L}+\{\ell\leftarrow \texttt{true}\}\rangle & \text{where } \ell \text{ is fresh in } \sigma \\
C_d[\![\texttt{bit}\ x\ =\ \texttt{false}]\!]\langle \rho,\sigma,\eta,\mathcal{L}\rangle & = & \langle \rho[x\leftarrow\ell],\sigma[\ell\leftarrow\bullet],\eta,\mathcal{L}+\{\ell\leftarrow \texttt{false}\}\rangle & \text{where } \ell \text{ is fresh in } \sigma \\
C_d[\![\texttt{bit}\ x\ =\ y\ \odot\ z]\!]\langle \rho,\sigma,\eta,\mathcal{L}\rangle & = & \langle \rho[x\leftarrow\ell],\sigma[\ell\leftarrow\bullet],\eta,\mathcal{L}+\{\ell\leftarrow \rho(y)\odot\rho(z)\}\rangle & \text{where } \ell \text{ is fresh in } \sigma \\
C_d[\![\texttt{non\_local bit}\ x\ =\ \texttt{true}]\!]\langle \rho,\sigma,\eta,\mathcal{L}\rangle & = & \langle \rho[x\leftarrow\ell],\sigma[\ell\leftarrow\bullet],\eta[x\backslash\ell],\mathcal{L}+\{\ell\leftarrow \texttt{true}\}\rangle & \text{where } \ell \text{ is fresh in } \sigma \\
C_d[\![\texttt{non\_local bit}\ x\ =\ \texttt{false}]\!]\langle \rho,\sigma,\eta,\mathcal{L}\rangle & = & \langle \rho[x\leftarrow\ell],\sigma[\ell\leftarrow\bullet],\eta[x\backslash\ell],\mathcal{L}+\{\ell\leftarrow \texttt{false}\}\rangle & \text{where } \ell \text{ is fresh in } \sigma \\
C_d[\![\texttt{non\_local bit}\ x\ =\ y\ \odot\ z]\!]\langle \rho,\sigma,\eta,\mathcal{L}\rangle & = & \langle \rho[x\leftarrow\ell],\sigma[\ell\leftarrow\bullet],\eta[x\backslash\ell],\mathcal{L}+\{\ell\leftarrow \rho(y)\odot\rho(z)\}\rangle & \text{where } \ell \text{ is fresh in } \sigma
\end{array}
$$

Figure 5: Formal semantics with collecting closures.

Table 4: Staged execution of our running example code with a stage execution kicks in after line 16.

line #	ρ	σ	η	$\mathcal{L}$
0	$\rho_0 = \{\}$	$\sigma_0 = \{\}$	$\eta_0 = \{\}$	$\mathcal{L}_0 = \{\}$
1	$\rho_1 = \rho_0[\texttt{r} \leftarrow \texttt{0x1000}]$	$\sigma_1 = \sigma_0$	$\eta_1 = \eta_0$	$\mathcal{L}_1 = \mathcal{L}_0$
2	$\rho_2 = \rho_1[\texttt{rs0} \leftarrow \texttt{0x1004}]$	$\sigma_2 = \sigma_1$	$\eta_2 = \eta_1$	$\mathcal{L}_2 = \mathcal{L}_1$
3	$\rho_3 = \rho_2[\texttt{rs1} \leftarrow \texttt{0x1008}]$	$\sigma_3 = \sigma_2$	$\eta_3 = \eta_2$	$\mathcal{L}_3 = \mathcal{L}_2$
4	$\rho_4 = \rho_3[\texttt{xs0} \leftarrow \texttt{0x100c}]$	$\sigma_4 = \sigma_3$	$\eta_4 = \eta_3$	$\mathcal{L}_4 = \mathcal{L}_3$
5	$\rho_5 = \rho_4[\texttt{xs1} \leftarrow \texttt{0x1010}]$	$\sigma_5 = \sigma_4$	$\eta_5 = \eta_4$	$\mathcal{L}_5 = \mathcal{L}_4$
6	$\rho_6 = \rho_5$	$\sigma_6 = \sigma_5$	$\eta_6 = \eta_5$	$\mathcal{L}_6 = \mathcal{L}_5$
7	$\rho_7 = \rho_6[\texttt{x} \leftarrow \texttt{0x1014}]$	$\sigma_7 = \sigma_6$	$\eta_7 = \eta_6$	$\mathcal{L}_7 = \mathcal{L}_6 + \{\texttt{0x1014} \leftarrow (*\ \texttt{0x100c})\}$
8	$\rho_8 = \rho_7[\texttt{t} \leftarrow \texttt{0x1018}]$	$\sigma_8 = \sigma_7$	$\eta_8 = \eta_7$	$\mathcal{L}_8 = \mathcal{L}_7$
9	$\rho_9 = \rho_8[\texttt{t} \leftarrow \texttt{0x101c}]$	$\sigma_9 = \sigma_8$	$\eta_9 = \eta_8$	$\mathcal{L}_9 = \mathcal{L}_8 + \{\texttt{0x101c} \leftarrow (*\ \texttt{0x1018}) \land (*\ \texttt{0x1014})\}$
10	$\rho_{10} = \rho_9[\texttt{r} \leftarrow \texttt{0x1020}]$	$\sigma_{10} = \sigma_9$	$\eta_{10} = \eta_9[\texttt{r}\backslash\texttt{0x1020}]$	$\mathcal{L}_{10} = \mathcal{L}_9 + \{\texttt{0x1020} \leftarrow (*\ \texttt{0x1000}) \land (*\ \texttt{0x101c})\}$
11	$\rho_{11} = \rho_5$	$\sigma_{11} = \sigma_{10}$	$\eta_{11} = \eta_{10}$	$\mathcal{L}_{11} = \mathcal{L}_{10}$
12	$\rho_{12} = \rho_{11}$	$\sigma_{12} = \sigma_{11}$	$\eta_{12} = \eta_{11}$	$\mathcal{L}_{12} = \mathcal{L}_{11} + \{\texttt{0x1000} \leftarrow (*\ \texttt{0x1020})\}$
13	$\rho_{13} = \rho_{12}[\texttt{rs0} \leftarrow \texttt{0x1024}]$	$\sigma_{13} = \sigma_{12}$	$\eta_{13} = \eta_{12}$	$\mathcal{L}_{13} = \mathcal{L}_{12} + \{\texttt{0x1024} \leftarrow (*\ \texttt{0x1000})\}$
14	$\rho_{14} = \rho_{13}$	$\sigma_{14} = \sigma_{13}$	$\eta_{14} = \eta_{13}$	$\mathcal{L}_{14} = \mathcal{L}_{13}$
15	$\rho_{15} = \rho_{14}[\texttt{x} \leftarrow \texttt{0x1028}]$	$\sigma_{15} = \sigma_{14}$	$\eta_{15} = \eta_{14}$	$\mathcal{L}_{15} = \mathcal{L}_{14} + \{\texttt{0x1028} \leftarrow (*\ \texttt{0x1010})\}$
16	$\rho_{16} = \rho_{15}[\texttt{x} \leftarrow \texttt{0x102c}]$	$\sigma_{16} = \sigma_{15}$	$\eta_{16} = \eta_{15}$	$\mathcal{L}_{16} = \mathcal{L}_{15}$
	If a staged execution kicks in here, then it updates $\sigma_{16} = \{(\texttt{0x1000}, \texttt{false}), (\texttt{0x1014}, \texttt{true}), (\texttt{0x101c}, \texttt{false}), (\texttt{0x1020}, \texttt{false}), (\texttt{0x1024}, \texttt{false}), (\texttt{0x1028}, \texttt{true})\}$ and $\mathcal{L}_{16} = \{\}$.			
17	$\rho_{17} = \rho_{16}[\texttt{t} \leftarrow \texttt{0x1030}]$	$\sigma_{17} = \sigma_{16}$	$\eta_{17} = \eta_{16}$	$\mathcal{L}_{17} = \mathcal{L}_{16} + \{\texttt{0x1030} \leftarrow (*\ \texttt{0x102c}) \land (*\ \texttt{0x1028})\}$
18	$\rho_{18} = \rho_{17}[\texttt{r} \leftarrow \texttt{0x1034}]$	$\sigma_{18} = \sigma_{17}$	$\eta_{18} = \eta_{17}[\texttt{r}\backslash\texttt{0x1034}]$	$\mathcal{L}_{18} = \mathcal{L}_{17} + \{\texttt{0x1034} \leftarrow (*\ \texttt{0x1000}) \land (*\ \texttt{0x1030})\}$
19	$\rho_{19} = \rho_{18}$	$\sigma_{19} = \sigma_{18}$	$\eta_{19} = \eta_{18}$	$\mathcal{L}_{19} = \mathcal{L}_{18}$
20	$\rho_{20} = \rho_{19}$	$\sigma_{20} = \sigma_{19}$	$\eta_{20} = \eta_{19}$	$\mathcal{L}_{20} = \mathcal{L}_{19} + \{\texttt{0x1000} \leftarrow (*\ \texttt{0x1034})\}$
21	$\rho_{21} = \rho_{20}[\texttt{rs1} \leftarrow \texttt{0x1038}]$	$\sigma_{21} = \sigma_{20}$	$\eta_{21} = \eta_{20}$	$\mathcal{L}_{21} = \mathcal{L}_{20} + \{\texttt{0x1038} \leftarrow (*\ \texttt{0x1000})\}$

$$
\begin{array}{rcl}
\langle \rho,\sigma,\eta,\{\ell\leftarrow\texttt{true}\}+\mathcal{L}\rangle & = & \langle \rho,\sigma[\ell\leftarrow\texttt{true}],\eta,\mathcal{L}\rangle \\
\langle \rho,\sigma,\eta,\{\ell\leftarrow\texttt{false}\}+\mathcal{L}\rangle & = & \langle \rho,\sigma[\ell\leftarrow\texttt{false}],\eta,\mathcal{L}\rangle \\
\langle \rho,\sigma,\eta,\{\ell\leftarrow\ell_1\odot\ell_2\}+\mathcal{L}\rangle & = & \langle \rho,\sigma[\ell\leftarrow\sigma(\ell_1)\odot\sigma(\ell_2)],\eta,\mathcal{L}\rangle
\end{array}
$$

Figure 6: Formal semantics for executing delayed closures.

figure, the three equations simply move the atomic actions from $\mathcal{L}$ to the store σ.

We formalize the argument that the staged semantics produces the same answers as the canonical semantics in the following proposition.

Proposition 5.1 (Correctness of Staged Semantics). *The sequence of atomic actions $\ell_i \leftarrow v_i$ on the store performed in the canonical semantics coincides with the ones performed in the staged semantics.*

Third, in a configuration $\langle \rho,\sigma,\eta,\mathcal{L}\rangle$, the environment ρ and the delayed closures $\mathcal{L}$ contain all the live locations. (The locations in the association η are already in the environment ρ.) Any method that shrinks σ by collecting all other locations, i.e., all locations that do not occur in ρ or in $\mathcal{L}$, would be sound. Our specification for this abstract notion of garbage collection is inspired by the line of work on abstract models of memory management initiated by

Morrisett et al. [32]. We need a rule:

$$\langle \rho, \sigma[\ell \leftarrow v], \eta, \mathcal{L} \rangle = \langle \rho, \sigma, \eta, \mathcal{L} \rangle$$

if ℓ occurs in neither ρ nor $\mathcal{L}$. The rule specifies that the configuration consisting of the environment, store, association η, and closures on the left is indistinguishable from the configuration on the right. First the location ℓ occurs exactly once in the left store $\sigma[\ell \leftarrow v]$ as locations are always unique in the store. That location is deallocated on the right leaving the store σ without the location ℓ. The correctness of this rule is straightforward to prove by inspecting the semantic relations in Figs. 5 and 6. Indeed, in every semantic rule that mentions a store lookup $\sigma(\ell)$ or a store update $\sigma[\ell \leftarrow v]$, the location ℓ is already present in either ρ or $\mathcal{L}$. Hence all other locations are inaccessible and can be garbage collected.

Therefore the actual space used at any point during the execution is proportional to the number of live locations at that point, i.e., the number of locations accessible through either the current environment or the closures data structure. At one extreme, the closures data structure can be restricted to have exactly one closure which is immediately executed. The space usage in this case would coincide with that of the standard scoped execution and that corresponds to the minimum-space/maximum-rounds needed to execute the circuit. At the other extreme, the closures data structure can extend through the entire program forcing all locations to be simultaneously live, achieving minimum-rounds at the cost of maximum-space. In practice, the size of the closures data structure can be a constant independent of the size of the program and the space usage is only a constant factor more than the minimal one. Thus, we conclude with a proposition about the space efficiency of staged execution semantics.

Proposition 5.2 (Space Usage). *If the staged semantics collects p operations for each stage, its space usage at each point in the execution is proportional to p plus the number of live locations in the environment.*

5.2 Fixing Π_{2pc}

We modifies WRK's Π_{2pc} so that it uses $\mathcal{F}_{aAND}$ instead of $\mathcal{F}_{Pre}$'s **aAND**. This is done by replacing the step (4a) of Figure 14 with the following:

(4) (a) P_1 and P_2 invoke $\mathcal{F}_{aAND}$. In return, P_1 receives random $[x_1]^1, [x_2]^1, [x_3]^1$ and P_2 receives random $[y_1]^2, [y_2]^2, [y_3]^2$ such that $(x_1 \oplus y_1)(x_2 \oplus y_2) = x_3 \oplus y_3$. P_1 and P_2 securely align $x_1 \oplus y_1$ with $r_\alpha \oplus s_\alpha$, and securely align $x_2 \oplus y_2$ with $r_\beta \oplus s_\beta$. Finally, P_1 sets $r_\sigma := x_3$ and P_2 sets $s_\sigma := y_3$, thus $r_\sigma \oplus s_\sigma = \lambda_\alpha \wedge \lambda_\beta$.

The modified step *securely aligns* two secret bits a, b, each of which is already divided into two abits, e.g., $([a_1]^1, [a_2]^2)$, $([b_1]^1, [b_2]^2)$. The alignment operation can be simply done by letting the two parties to exchange $[a_1 \oplus b_1]^1$ and $[a_2 \oplus b_2]^2$. This allows them to check in plaintext if $a_1 \oplus b_1 = a_2 \oplus b_2$, which is equivalent to $a_1 \oplus a_2 = b_1 \oplus b_2$. If the equality doesn't hold, P_1 should flip its bit a_1 while P_2 resetting $\mathsf{K}[a_1] := \mathsf{K}[a_1] \oplus \Delta_2$.

Proposition 5.3. *If H is modeled as a random oracle, the original WRK protocols with modified step (4a) described above securely computes f against malicious adversaries with statistical security 2^{-s} in a hybrid model with ideal $\mathcal{F}_{abit}$ and $\mathcal{F}_{aAND}$.*

Intuitively, because both x_1 and r_α are uniform, revealing $x_1 \oplus r_\alpha$ won't give P_2 any advantage in guessing either x_1 or r_α. The proposition above is proved in a hybrid model with ideal $\mathcal{F}_{abit}$ and $\mathcal{F}_{aAND}$. The proof is in the full version of the paper.

5.3 Scalable Generation of abits and aANDs

We describe our scalable abit generation protocol $\Pi_{abit}^{Scalable}$ in the $\mathcal{F}_{Leaky\text{-}RCOT}$-hybrid model. $\mathcal{F}_{Leaky\text{-}RCOT}$ functionality is given in Figure 7, which can be efficiently implemented based on actively-secure OT extension by Keller et al. [18].

$\mathcal{F}_{Leaky\text{-}RCOT}(n, \ell)$

Public parameters: security parameter n, batch-size ℓ.

Honest case: Upon receiving $\Delta \in \{0,1\}^n$ from P_1, for every $i \in [\ell]$ choose uniform $c_i \in \{0,1\}$ and uniform $m_i \in \{0,1\}^n$. Send $\{m_i\}_{i\in[\ell]}$ to P_1 and send $\{(c_i, m_i \oplus c_i \cdot \Delta)\}_{i\in[\ell]}$ to P_2.

Dishonest case: Upon receiving x guesses $\{(i_j, b_j)\}_{j\in[x]}$ (where $i_j \in [n]$, $b_j \in \{0,1\}$) from P_2, check if $\Delta_{i_j} = b_j$ (i.e., whether the i_j-th bit of Δ is equal to b_j) for every $j \in [x]$. If *all* checks pass, send nothing to P_1 but "attack succeed" to P_2; otherwise, in case *any* check fails, send "P$_2$ cheats" to P_1 and "bad guess" to P_2.

Figure 7: The leaky-RCOT functionality.

Our $\Pi_{abit}^{Scalable}$ in Figure 8 generates abits in batches of p. A key insight is that we can allow different batches to share the same Δ by letting the sender to pick uniform rank-n matrix A and $\Delta' \in \{n+s\}$ for each batch subject to $A \times \Delta' = \Delta$. Δ' will be used to run a batch of $p+s$ Leaky-RCOT, and A is used to specify an entropy-preserving linear transform to compress the correlated leaky messages returned by $\mathcal{F}_{Leaky\text{-}RCOT}$. It is important to run the consistency check in (4c) to prevent active attackers from deliberately using different Δ in different batches. The basic idea of this check resembles the ones used in several prior works [18, 36, 47].

Proposition 5.4. $\Pi_{abit}^{Scalable}$ *of Figure 8 can securely realize* $\mathcal{F}_{abit}$.

The aAND generation protocol $\Pi_{aAND}^{Scalable}$ is given in Figure 9. The cut-and-choose bucket size of $\Pi_{aAND}^{Scalable}$ will be derived from the statistical security parameter s and the space budget of the pool, using the process explained below.

Determine Cut-and-choose Bucket Size. Given statistical parameter s and pool size p, the parties want to find the most efficient bucket size B to bound the failure rate by 2^{-s}.

In $\Pi_{aAND}^{Scalable}$, every faulty leaky-aAND will be detected with probability 1/2. Additionally, every leaky-aAND can be checked but also later combined with other randomly picked $B-1$ leaky-aAND to form a secure aAND. Let $P_B(p,b)$ be the probability of a successful attack throughout the lifetime of a pool of size p and bucket-size B provided that *at most b* faulty leaky-aAND have ever been successfully inserted into the pool. The security of $\Pi_{aAND}^{Scalable}$ is guaranteed if a single leaky-AND in each bucket is honestly-generated. So we have the following recurrence for $P_B(p,k)$:

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

Protocol $\Pi_{\text{abit}}^{\text{Scalable}}(n)$

Input: P_1's input is Δ and P_2 has no input.
Output: P_1 gets (up to) *infinitely* many uniform K_i where $\mathsf{K}_i \in \{0,1\}^n$; P_2 gets (up to) *infinitely* many uniform random (b_i, M_i) where $\mathsf{M}_i \in \{0,1\}^n$ such that $\mathsf{M}_i = \Delta \oplus b_i \cdot \mathsf{K}_i$ for all i.
Side-effect: Both parties use constant space.

(1) P_1 generates uniform rank-n matrix $A \in \{0,1\}^{n\times(n+s)}$ and uniform $\Delta' \in \{0,1\}^{n+s}$ subject to $A \times \Delta' = \Delta$. P_1 commits A.

(2) P_1 uses Δ' to run $\mathcal{F}_{\text{Leaky-RCOT}}(n+s, s)$ with P_2 so that P_1 gets $\{\mathsf{K}_i'^*\}_{i\in[s]}$ and P_2 obtains $\{(b_i^*, \mathsf{M}_i'^*)\}_{i\in[s]}$.

(3) P_1 opens A to P_2. P_1 computes $\mathsf{K}_i^* := A \times \mathsf{K}_i'^*$ for all $i \in [s]$. P_2 computes $\mathsf{M}_i^* := A \times \mathsf{M}_i'^*$ for all $i \in [s]$.

(4) To obtain every subsequent batch of p authenticated bits,
  (a) P_1 regenerates uniform rank-n matrix $A \in \{0,1\}^{n\times(n+s)}$ and uniform $\Delta' \in \{0,1\}^{n+s}$ subject to $A \times \Delta' = \Delta$. P_1 commits A.
  (b) P_1 uses the refreshed Δ' to run $\mathcal{F}_{\text{Leaky-RCOT}}(n+s, p+s)$ (with P_2) so P_1 learns $\{\mathsf{K}_i'\}_{i\in[p+s]}$ and P_2 learns $\{(b_i, \mathsf{M}_i')\}_{i\in[p+s]}$. Then P_1 opens the refreshed A to P_2. For every $i \in [p+s]$, P_1 computes $\mathsf{K}_i := A \times \mathsf{K}_i'$ and P_2 computes $\mathsf{M}_i := A \times \mathsf{M}_i'$.
  (c) Repeat the following check s times (for $i \in [s]$),
    (i) P_2 sends $b_i^* \oplus b_{p+i}$ to P_1.
    (ii) P_1 computes $\hat{\mathsf{K}} = \mathsf{K}_i^* \oplus \mathsf{K}_{p+i} \oplus (b_i^* \oplus b_{p+i})\Delta$ whereas P_2 computes $\hat{\mathsf{M}} = \mathsf{M}_i^* \oplus \mathsf{M}_{p+i}$. P_1 and P_2 invoke $\mathcal{F}_{\text{Equal}}$ using $\hat{\mathsf{K}}$ and $\hat{\mathsf{M}}$, respectively. They abort if $\mathcal{F}_{\text{Equal}}$ returns unequal.
  (d) To refresh $\{(\mathsf{K}^*, b^*, \mathsf{M}^*)\}_{i\in[s]}$, for every $i \in [s]$,
    (i) P_1 sets $\mathsf{K}_i^* := \mathsf{K}_{p+i}$.
    (ii) P_2 sets $b_i^* := b_{p+i}$, $\mathsf{M}_i^* := \mathsf{M}_{p+i}$.
  (e) P_1 outputs $\{\mathsf{K}_i\}_{i\in[p]}$. P_2 outputs $\{(b_i, \mathsf{M}_i)\}_{i\in[p]}$.

Figure 8: Scalable abit protocol using $\mathcal{F}_{\text{Leaky-RCOT}}$, $\mathcal{F}_{\text{Equal}}$ and $\mathcal{F}_{\text{com}}$.

Protocol $\Pi_{\text{aAND}}^{\text{Scalable}}(s, p)$

Public parameters: Security parameter s, pool size p.
Input: P_1 and P_2 each has $\perp$.

One-time pool-initialization: P_1 and P_2 invoke WRK's original leaky aAND protocol to generate p leaky aAND triples, which form the pool. Both parties use the public parameter search algorithm to compute cut-and-choose bucket size B from (s, p).

Online steps:

(1) P_1 and P_2 run WRK's original leaky aAND protocol to generate p leaky aAND triples that form a *buffer*.

(2) P_1 and P_2 collaboratively select B random leaky ANDs from the pool and use WRK's original combining algorithm to form a fully-secure aAND triple.

(3) They move B leaky aAND triples from the buffer to fill the pool. If there is not enough leaky aANDs in the buffer, go to step (1).

Figure 9: Scalable aAND from a pool of leaky-aANDs.

Therefore, once p and B are fixed, $P_B(p,1), \ldots, P_B(p,b)$ can be computed altogether using dynamic programming in $O(b)$ time.

Assuming the bucket size B is a constant, the success rate of the best attacking strategy is then

$$\max_b 2^{-b} \cdot P_B(n, b) \tag{1}$$

where b can be any positive integer (but is somehow bounded by s). To find the best B, simply check all possible integer values of B starting from 2 and stop as soon as a satisfying B is found.

Set $t := \infty$, $b_0 := s$, $B := 2$, and repeat the steps below until it exits from (2):
(1) Use the recurrence above to compute $P_B(p,k)$ for all $0 \le k \le b_0$.
(2) If $\max_{k\in\{1,\ldots,b_0\}} 2^{-k} \cdot P_B(p,k) \le 2^{-s}$, then output B and halt; otherwise, set $B := B+1$.
(3) If $B > p$, exit with output $\perp$.

Proposition 5.5. *Protocol $\Pi_{\text{aAND}}^{\text{Scalable}}(s,p)$ of Figure 9 securely realizes $\mathcal{F}_{\text{aAND}}$.*

Formal proofs of the two propositions above can be found in the full version of this paper.

Security of nanoPI. Proposition 5.4, 5.5, and 5.3 together guarantee the security of the cryptographic modifications of nanoPI, while Proposition 5.1 guarantees that the instrumentation we used preserves the semantics of our modified cryptographic protocol. Therefore, nanoPI is secure.

6 THE MULTI-PARTY COMPUTATION SETTING

A clear advantage of WRK is that it is easy to generalize it into a constant-round secure multi-party computation protocol. In this setting, an n-party authenticated bit of P_i, denoted as $[b]^i$, refers to the tuple $\left(b, \{\mathsf{M}_j[b]\}_{j\neq i}, \{\mathsf{K}_j[b]\}_{j\neq i}\right)$ where P_i holds $\left(b, \{\mathsf{M}_j[b]\}_{j\neq i}\right)$ and P_j ($\forall j \neq i$) holds $\mathsf{K}_j[b]$ and Δ_j so that $\mathsf{M}_j[b] = \mathsf{K}_j[b] \oplus b\Delta_j$. Similarly, an n-party sub-protocol is used to securely compute n-party authenticated AND triples

$$\left([a_1]^1 \oplus \cdots \oplus [a_n]^n\right) \cdot \left([b_1]^1 \oplus \cdots \oplus [b_n]^n\right) = [c_1]^1 \oplus \cdots \oplus [c_n]^n$$

where the triple $\left([a_i]^i, [b_i]^i, [c_i]^i\right)$ denotes the i-th party's authenticated bit-shares of the AND triple. In the online stage, one of the parties will execute as the circuit evaluator, collecting the shares of the garbled table from the rest $n-1$ parties and evaluating the combined garbled circuit.

We first fix the presentation flaw of WRK's Π_{mpc} so it works in a hybrid model with ideal $\mathcal{F}_{\text{abit}}^n$ and $\mathcal{F}_{\text{aAND}}^n$. Then we show how to efficiently scale up $\mathcal{F}_{\text{abit}}^n$ and $\mathcal{F}_{\text{aAND}}^n$. For the convenience of the readers, we copied the definition of WRK's $\mathcal{F}_{\text{abit}}^n$ and $\mathcal{F}_{\text{aAND}}^n$ below.

$\mathcal{F}_{\text{abit}}^n$

Honest case: The box receives (input, i, ℓ) from all parties and picks random bit-string $x \in \{0,1\}^\ell$. For each $j \in [\ell]$, $k \neq i$,the box picks random $\mathsf{K}_k[x_j]$, and computes $\{\mathsf{M}_k[x_j] := \mathsf{K}_k[x_j] \oplus x_j\Delta_k\}_{k\neq i}$, and sends them to parties. That is, for each $j \in [\ell]$, it sends $\{\mathsf{M}_k[x_j]\}_{k\neq i}$ to P_i and sends $K_k[x_j]$ to P_k for each $k \neq i$.
Corrupted parties: A corrupted party can choose their output from the protocol.

$\mathcal{F}^n_{\mathsf{aAND}}$

Honest case: Generate uniform $[r^i_1]^i$, $[r^i_2]^i$, $[r^i_3]^i$ such that $\left(\bigoplus_i r^i_1\right) \wedge \left(\bigoplus_i r^i_2\right) = \bigoplus_i r^i_3$.

Corrupted parties: A corrupted party can choose their output from the protocol.

6.1 Fixing WRK's Π_{mpc}

The multi-party authenticated garbling protocol Π_{mpc} [45, Figure 2, Figure 3] also needs two alignment operations per AND. Similarly, we replace step (4a) of WRK's Π_{mpc} with the following:

(4) (a) All $P_i (1 \le i \le n)$ jointly invoke $\mathcal{F}_{\mathsf{aAND}}$ so that P_i receives $[x^i_1]^i, [x^i_2]^i, [x^i_3]^i$ where $\left(\bigoplus_i x^i_1\right) \wedge \left(\bigoplus_i x^i_2\right) = \bigoplus_i x^i_3$. P_i securely aligns $\bigoplus_i x^i_1$ with $\bigoplus_i r^i_\alpha$, and securely aligns $\bigoplus_i x^i_2$ with $\bigoplus_i r^i_\beta$. Finally, P_i sets $r^i_\sigma := x^i_3$ thus $\bigoplus_i r^i_\sigma = \lambda_\alpha \wedge \lambda_\beta$.

Note that the secure alignment becomes a multi-party operation over two secret bits a, b, each of which is divided into n shares, e.g. $a^i (1 \le i \le n)$ and $b^i (1 \le i \le n)$. The alignment is done by letting P_i broadcast $[a^i \oplus b^i]^i$. This allows them to check whether $\bigoplus_i (a^i \oplus b^i) = 0$ which is equivalent to $\bigoplus_i a^i = \bigoplus b^i$. If the equality doesn't hold, then P_1 flips a^1 while every $P_i (2 \le i \le n)$ setting $\mathsf{K}_1[a^1] := \mathsf{K}_1[a^1] \oplus \Delta_i$. Since x^1_1 and r^1_α are uniform, revealing $x^1_1 \oplus r^1_\alpha$ won't leak any information on x^1_1 or r^1_α.

6.2 Efficiently Scalable $\Pi^{\mathsf{Scalable},n}_{\mathsf{abit}}(s, p)$

WRK [45]'s $\mathcal{F}^n_{\mathsf{abit}}$ is built from the two-party $\mathcal{F}_{\mathsf{abit}}$. So we can scale up their Π^n_{abit} protocol by instantiating every call to $\mathcal{F}_{\mathsf{abit}}$ in their Π^n_{abit} using our $\Pi^{\mathsf{Scalable}}_{\mathsf{abit}}$.

Note that $\Pi^{\mathsf{Scalable}}_{\mathsf{abit}}(s, \ell)$ in Figure 10 guarantees that each P_i uses consistent choice bits across corresponding instances $\mathcal{F}_{\mathsf{abit}}$. $\mathcal{F}_{\mathsf{abit}}$ guarantee that a consistent Δ is used between different executions.

6.3 Efficiently Scalable $\Pi^{\mathsf{Scalable},n}_{\mathsf{aAND}}(s, p)$

The efficient scalability of our multi-party $\Pi^{\mathsf{Scalable},n}_{\mathsf{aAND}}$ protocol is derived from the same idea of employing pool-based cut-and-choose. The multi-party $\Pi^{\mathsf{Scalable},n}_{\mathsf{aAND}}$ is the same as the two-party $\Pi^{\mathsf{Scalable}}_{\mathsf{aAND}}$, except that it uses the multi-party version of the procedures to generate leaky-aANDs, produce random cut-and-choose choices, and combine several leaky-aANDs into a full-secure aAND, which were detailed in [45]. The security analysis of the pool-based cut-and-choose remain identical to that for the two-party setting.

7 EVALUATION

Experimental Setup. Unless specified otherwise, we used Amazon EC2 compute-optimized instances `c5.large` to run most of our experiments. The `c5.large` virtual machines are equipped with 2 vCPU and 4 GB memory, running Ubuntu 18.04 and costing only ¢8.5/hour. We measured the LAN setting to be roughly 2 Gbps with 0.2 ms latency; and the WAN setting roughly 200 Mbps with 40ms round-trip latency.

All experiments were configured to achieve 128-bit computational security and 40-bit statistical security. When no specific application is mentioned, experiments are run over a random circuit of 25% ANDs and 75% XORs, with speeds measured over a period of at least 10 minutes. Experimental comparisons with related work were made by running reference implementations in the same hardware and network environment.

Protocol $\Pi^{\mathsf{Scalable},n}_{\mathsf{abit}}(s, \ell)$

Public parameters: Security parameter s, batch size ℓ.
Input: $P_i (1 \le i \le n)$ has $\perp$.

(1) Set $\ell' := \ell + s$. P_i uniformly picks $x \in \{0, 1\}^{\ell'}$ and a random seed S_i.

(2) Each P_i runs an instance of $\mathcal{F}_{\mathsf{abit}}$ with every P_k where $k \ne i$ using batch size ℓ'. As a result, P_i as the OT receiver gets $\left\{\{x_{k,\iota}, \mathsf{M}_k[x_{k,\iota}]\}_{\iota \in [\ell']}\right\}_{k \in [n], k \ne i}$ and every P_k as an OT sender gets $\{\mathsf{K}_k[x_{k,\iota}]\}_{\iota \in [\ell']}$.

(3) For every $\iota \in [\ell']$, P_i picks a uniform x_ι and aligns all the values of $x_{k,\iota}$ with x_ι. The alignment is done by first revealing whether $x_{k,\iota} = x_\iota$ to every P_k, then leaving $x_{k,\iota}$ unchanged if the equality holds; while flipping $x_{k,\iota}$ (with P_k refreshing $\mathsf{K}_k[x_{k,\iota}] := \mathsf{K}_k[x_{k,\iota}] \oplus \Delta_k$) otherwise.

(4) (Consistency Checks) For $j \in [s]$, run the following steps:
  (a) All the parties collaborate to toss a random ℓ'-bit string r.
  (b) For every distinct $i, k \in [n]$, P_i and P_k perform:
    (i) P_i computes and broadcasts $X_j = \bigoplus_{\iota=1}^{\ell'} r_\iota x_\iota$, then computes $\left\{\mathsf{M}_k[X_j] = \bigoplus_{\iota=1}^{\ell'} r_\iota \mathsf{M}_k[x_\iota]\right\}_{k \ne i}$.
    (ii) P_k computes $\left\{\mathsf{K}_k[X_j] := \bigoplus_{\iota=1}^{\ell'} r_\iota \mathsf{K}_k[x_\iota]\right\}_{k \ne i}$.
    (iii) P_i sends $\mathsf{M}_k[X_j]$ to P_k who verifies its validity.

(5) All parties return the first ℓ objects.

Figure 10: The protocol $\Pi^{\mathsf{Scalable},n}_{\mathsf{abit}}$ realizing $\mathcal{F}^n_{\mathsf{abit}}$

7.1 Scalability

nanoPI can efficiently run circuits at unprecedented scales. Table 5 shows several benchmark computations we tested, none of which were ever possible to run using prior techniques. Most notably, we have tested nanoPI on building a multi-party actively-secure logistical regression and run it over realistic datasets like MNIST [24]. Followed the observation of SecureML [30], we chose statistical gradient descend (SGD) approach of training, used RELU to approximate the logistic function, and handled decimal arithmetics using 24-bit fixed-point number system. Still, the resulting circuit is gigantic, consisting of 4.7 billion ANDs and 8.9 billion XORs, assuming a two-pass scan of a 1K records dataset is involved in SGD training.

We have also tested nanoPI on running long-term actively-secure multi-party computation services, executing a random circuit. The experiments were run on `n1-standard-1` machines provided by Google Compute Engine connected with LAN, executing 40.8 billions ANDs in 16 days in the four-party setting.

7.2 Performance

As is evidenced from the benchmark applications (Table 5), our two-party protocols executed at fairly consistent speeds: roughly

46K AND/s on LAN and 15.8K AND/s on WAN. In four-party scenarios, the speeds reduce to roughly 31.6K AND/s and 10K AND/s, respectively. The bandwidth overhead is about 759 bytes per AND gate varying a little over different applications. Also note that the bandwidth cost of $\Pi_{\mathrm{abit}}^{\mathrm{Scalable}}$ and $\Pi_{\mathrm{aAND}}^{\mathrm{Scalable}}$ combined is about 5 times that of authenticated circuit garbling, which matches well with our theoretical analysis.

Figure 11 shows how increasing memory helps speedup our protocol. In the LAN setting, we observe a seemingly linear correlation between memory budget and protocol execution speed when memory budget is less than 50MB. Once over 200MB memory is available, the speed stays roughly the same, mainly because when the batch size is large enough, time spend on computation will dominate other factors including round-trip latency. For the WAN setting, the linear correlation continues until ~500MB memory is available, since now the latencies are close to two orders of magnitude larger than LAN. Our approach scales with the number of parties much like WRK (see Figure 12).

7.3 Comparison with Related Work

Table 6 shows how the performance of this work compares with two closest pieces of work, WRK and Pool-JIMU, in the two-party setting. We considered three kinds of memory budgets in combination with three types of network environments.

Compared with Pool-JIMU, our protocol and WRK will outperform Pool-JIMU's speed by 1.4–3.5x in low bandwidth network. This is mainly because Pool-JIMU is 6–8x more costly in bandwidth. But if the network bandwidth is high Pool-JIMU can be 50–130% faster than both our work and WRK due to its more efficient local computation. However, Pool-JIMU doesn't work in the general MPC setting (Figure 13).

Compared with WRK, performance of our protocols is only slightly cutback if memory budget is 200MB or greater. When memory budget is low, although our approach runs slower than WRK, ours can still run arbitrarily large circuits whereas WRK can only run small circuits that fits in the available memory. Interestingly, in the setup with 200MB memory and a (200Mbps, 40ms) WAN, WRK runs 86% faster than ours while either reducing or boosting the network performance allows our approach to catch up with WRK. This is because switching from (200Mbps, 40ms) to (20Mbps, 40ms) network, the speed bottleneck of WRK and our protocol will shift from roundtrip latency to network transmission; while switching from (200Mbps, 40ms) to (2Gbps, <1ms) network, the bottleneck will shift from roundtrip latency to computation. This shows that the drawback of our approach is only evident in limited scenarios when latency cost dominates both computation and transmission.

8 CONCLUSION

Round and space complexity are two conflicting but equally important goals in designing actively-secure MPC protocols. We gave an effective programming technique to scale WRK protocols up to arbitrary-size circuits. The programmatic and cryptographic transformations discussed in this paper is integrated into nanoPI, a toolchain opensourced on github to semi-automate the development and deployment of extreme-scale actively-secure MPC applications.

ACKNOWLEDGMENTS

We thank Xiao Wang and Weixi Ma for helpful comments and discussions. We thank Rob Henderson for pointers on obtaining and managing a cluster of experiment machines. We appreciate the generous free credits from Google for using the Google Cloud Platform. This work is supported by NSF award #1464113 and NIH 1U01EB023685-01.

REFERENCES

[1] Arash Afshar, Payman Mohassel, Benny Pinkas, and Ben Riva. 2014. Non-interactive secure computation based on cut-and-choose. In *EUROCRYPT*.
[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[3] Donald Beaver, Silvio Micali, and Phillip Rogaway. 1990. The round complexity of secure protocols. In *STOC*.
[4] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*.
[5] Henry Carter, Charles Lever, and Patrick Traynor. 2014. Whitewash: Outsourcing garbled circuit generation for mobile devices. In *ACSAC*.
[6] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. 2016. Secure outsourced garbled circuit evaluation for mobile devices. *Journal of Computer Security* 24, 2 (2016), 137–180.
[7] David Chaum, Claude Crépeau, and Ivan Damgard. 1988. Multiparty unconditionally secure protocols. In *STOC*.
[8] 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*.
[9] Jack Doerner, David Evans, and Abhi Shelat. 2016. Secure Stable Matching at Scale. In *ACM CCS*.
[10] Tore Frederiksen, Thomas Jakobsen, Jesper Nielsen, Peter Nordholt, and Claudio Orlandi. 2013. Minilego: Efficient secure two-party computation from general assumptions. In *EUROCRYPT*.
[11] Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Actively Secure Garbled Circuits with Constant Communication Overhead in the Plain Model. In *TCC*.
[12] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. 2017. Low cost constant round MPC combining BMR and oblivious transfer. In *ASIACRYPT*.
[13] Yan Huang, Peter Chapman, and David Evans. 2011. Privacy-Preserving Applications on Smartphones.. In *HotSec*.
[14] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. 2011. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium*.
[15] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex Malozemoff. 2014. Amortizing garbled circuits. In *CRYPTO*.
[16] Peter Z Ingerman. 1961. A way of compiling procedure statements with some comments on procedure declarations. *Commun. ACM* 4, 1 (1961), 55–58.
[17] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending oblivious transfers efficiently. In *CRYPTO*.
[18] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2015. Actively secure OT extension with optimal overhead. In *CRYPTO*.
[19] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *CCS*.
[20] Vladimir Kolesnikov, Jesper Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. 2017. DUPLO: Unifying Cut-and-Choose for Garbled Circuits. In *ACM CCS*.
[21] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved garbled circuit: Free XOR gates and applications. In *ICALP*.
[22] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. 2012. Billion-Gate Secure Computation with Malicious Adversaries. In *USENIX Security Symposium*.
[23] Peter J Landin. 1964. The mechanical evaluation of expressions. *Comput. J.* 6, 4 (1964), 308–320.
[24] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. *AT&T Labs. Available: http://yann.lecun.com/exdb/mnist* (2010).
[25] Yehuda Lindell. 2016. Fast cut-and-choose-based protocols for malicious and covert adversaries. *Journal of Cryptology* 29, 2 (2016), 456–490.
[26] Yehuda Lindell and Benny Pinkas. 2002. Privacy preserving data mining. *Journal of Cryptology* 15, 3 (2002).
[27] Yehuda Lindell, Benny Pinkas, Nigel P Smart, and Avishay Yanai. 2015. Efficient constant round multi-party computation combining BMR and SPDZ. In *CRYPTO*.
[28] Yehuda Lindell and Ben Riva. 2015. Blazing fast 2pc in the offline/online setting with security for malicious adversaries. In *ACM CCS*.
[29] 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*.

Table 5: Experiments on selected applications

		AES (1024 cipher blocks)		Sorting (5K 32-bit integers)		DNA Edit-distance (2000-nucleotides seq.)		Logistic Regression (1K-row, 784-column MNIST dataset)	
Gate Count (#AND, #XOR)		(6.9M, 25.9M)		(11.4M, 39.6M)		(340M, 880M)		(4.7B, 8.9B)	
Number of Parties		2	4	2	4	2	4	2	4
Time	LAN (minutes)	2.57	3.83	4.17	6.07	123	179	1676	2218
	WAN (minutes)	7.62	11.83	12.03	18.85	359	559	82 hours	129 hours
Bandwidth†	$\Pi^{\text{Scalable}}_{\text{abit}}, \Pi^{\text{Scalable}}_{\text{aAND}}$ (GB)	4.44	26.66	7.19	43.15	214.69	1288.14	2969.32	17815.94
	$\Pi^{\text{Scalable}}_{\text{mpc}}$ (GB)	0.89	5.35	1.46	8.73	43.53	261.16	602.00	3611.99

† Bandwidth numbers are the sum of the outgoing traffic of all parties during a specific application.

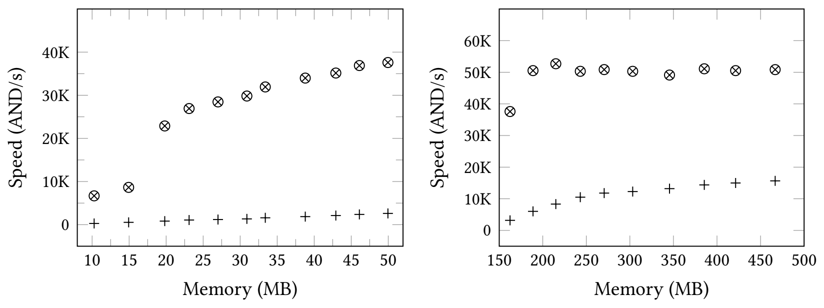

Figure 11: Memory's impact on performance
(⊗ for LAN, + for WAN)

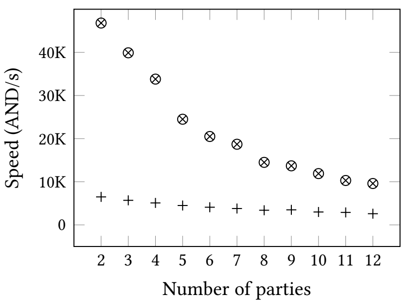

Figure 12: Scale up with # of parties
(⊗ for LAN, + for WAN)

Table 6: Comparing to Pool-JIMU and WRK in executing two-party random circuits (B refers to cut-and-choose bucket size.)

Memory Budget		**20 MB**			**200 MB**			**2 GB**		
Protocol		**This Work** ($B = 4$)	**Pool-JIMU** (B=6)	**WRK** (only if $\lvert C_f \rvert \le 3.8$K)	**This Work** (B=3)	**Pool-JIMU** ($B = 4$)	**WRK** (only if $\lvert C_f \rvert \le 38$K)	**This Work** ($B = 3$)	**Pool-JIMU** ($B = 4$)	**WRK** (only if $\lvert C_f \rvert \le 500$K)
Speed (AND/s)	**20 Mbps 40 ms**	795.03	561	1.73K	2.73K	787.72	2.75K	3.12K	925.04	3.23K
	200 Mbps 40 ms	825.18	4.12K	2.76K	6.94K	6.21K	12.94K	20.94K	7.20K	22.38K
	2 Gbps <1 ms	20.27K	46.6K	20.53K	46.66K	69.77K	46.84K	49.34K	88.01K	50.64K
Bandwidth (Byte/AND)		505	3.3K	504	380	2.2K	504	379	1.8K	378

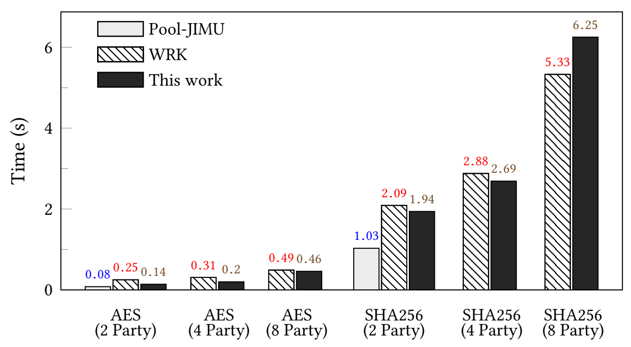

Figure 13: Application-specific performance comparisons. (Pool-JIMU was run with 2.6M pool, $B = 4$, and check rate $r_c = 3\%$. This work was run with 479K pool, $B = 3$, and stage of 128K AND.)

[30] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE Symposium on Security and Privacy*.

[31] Benjamin Mood, Lara Letaw, and Kevin Butler. 2012. Memory-efficient garbled circuit generation for mobile devices. In *International Conference on Financial Cryptography and Data Security*. Springer, 254–268.

[32] Greg Morrisett, Matthias Felleisen, and Robert Harper. 1995. Abstract Models of Memory Management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA)*. ACM, New York, NY, USA, 66–77.

[33] Moni Naor and Benny Pinkas. 2001. Efficient oblivious transfer protocols. In *SODA*.

[34] 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*.

[35] Jesper Nielsen, Peter Nordholt, Claudio Orlandi, and Sai Burra. 2012. A new approach to practical active-secure two-party computation. In *CRYPTO*.

[36] Jesper Nielsen, Thomas Schneider, and Roberto Trifiletti. 2017. Constant Round Maliciously Secure 2PC with Function-independent Preprocessing using LEGO. In *NDSS*.

Protocol Π_{2pc}

Inputs: the parties agree on a function $f : \{0,1\}^{|\mathcal{I}_1|} \times \{0,1\}^{|\mathcal{I}_2|} \to \{0,1\}^{|\mathcal{O}|}$. P_1 holds $x \in \{0,1\}^{|\mathcal{I}_1|}$ and P_2 holds $y \in \{0,1\}^{|\mathcal{I}_2|}$.

Function-independent preprocessing:

(1) P_1 and P_2 send **init** to $\mathcal{F}_{\text{Pre}}$, which sends Δ_1 to P_1 and Δ_2 to P_2.

(2) For each wire $w \in \mathcal{I}_1 \cup \mathcal{I}_2 \cup \mathcal{W}$ where $\mathcal{W}$ denotes the set of internal wires, P_1 and P_2 send **abit** to $\mathcal{F}_{\text{Pre}}$. In return, $\mathcal{F}_{\text{Pre}}$ sends $(r_w, \mathsf{M}[r_w], \mathsf{K}[s_w])$ to P_1 and $(s_w, \mathsf{M}[s_w], \mathsf{K}[r_w])$ to P_2. Define $\lambda_w = s_w \oplus r_w$. P_1 picks a uniform κ-bit string L_w^0 and sets $\mathsf{L}_w^1 := \mathsf{L}_w^0 \oplus \Delta_1$.

Function-dependent preprocessing:

(3) For each gate $\mathcal{G} = (\alpha, \beta, \gamma, \oplus)$, P_1 computes $(r_\gamma, \mathsf{M}[r_\gamma], \mathsf{K}[s_\gamma]) := (r_\alpha \oplus r_\beta, \mathsf{M}[r_\alpha] \oplus \mathsf{M}[r_\beta], \mathsf{K}[s_\alpha] \oplus \mathsf{K}[s_\beta])$, and sets $L_\gamma^0 := L_\alpha^0 \oplus L_\beta^0$ and $L_\gamma^1 := L_\gamma^0 \oplus \Delta_1$. Similarly, P_2 computes $(s_\gamma, \mathsf{M}[s_\gamma], \mathsf{K}[r_\gamma]) := (s_\alpha \oplus s_\beta, \mathsf{M}[s_\alpha] \oplus \mathsf{M}[s_\beta], \mathsf{K}[r_\alpha] \oplus \mathsf{K}[r_\beta])$. Define $\lambda_\gamma = \lambda_\alpha \oplus \lambda_\beta$.

(4) For each gate $\mathcal{G} = (\alpha, \beta, \gamma, \wedge)$:

(a) P_1 (resp., P_2) sends $(\textbf{aAND}, (r_\alpha, \mathsf{M}[r_\alpha], \mathsf{K}[s_\alpha]), (r_\beta, \mathsf{M}[r_\beta], \mathsf{K}[s_\beta]))$ (resp., $(\textbf{aAND}, (s_\alpha, \mathsf{M}[s_\alpha], \mathsf{K}[r_\alpha]), (s_\beta, \mathsf{M}[s_\beta], \mathsf{K}[r_\beta]))$) to $\mathcal{F}_{\text{Pre}}$. In return, $\mathcal{F}_{\text{Pre}}$ sends $(r_\sigma, \mathsf{M}[r_\sigma], \mathsf{K}[s_\sigma])$ to P_1 and $(s_\sigma, \mathsf{M}[s_\sigma], \mathsf{K}[r_\sigma])$ to P_2, where $s_\sigma \oplus r_\sigma = \lambda_\alpha \wedge \lambda_\beta$.

(b) P_1 computes the following locally:

$$\begin{array}{llll}
(r_{\gamma,0}, \mathsf{M}[r_{\gamma,0}], \mathsf{K}[s_{\gamma,0}]) := (r_\sigma \oplus r_\gamma, & \mathsf{M}[r_\sigma] \oplus \mathsf{M}[r_\gamma], & \mathsf{K}[s_\sigma] \oplus \mathsf{K}[s_\gamma] &) \\
(r_{\gamma,1}, \mathsf{M}[r_{\gamma,1}], \mathsf{K}[s_{\gamma,1}]) := (r_\sigma \oplus r_\gamma \oplus r_\alpha, & \mathsf{M}[r_\sigma] \oplus \mathsf{M}[r_\gamma] \oplus \mathsf{M}[r_\alpha], & \mathsf{K}[s_\sigma] \oplus \mathsf{K}[s_\gamma] \oplus \mathsf{K}[s_\alpha] &) \\
(r_{\gamma,2}, \mathsf{M}[r_{\gamma,2}], \mathsf{K}[s_{\gamma,2}]) := (r_\sigma \oplus r_\gamma \oplus r_\beta, & \mathsf{M}[r_\sigma] \oplus \mathsf{M}[r_\gamma] \oplus \mathsf{M}[r_\beta], & \mathsf{K}[s_\sigma] \oplus \mathsf{K}[s_\gamma] \oplus \mathsf{K}[s_\beta] &) \\
(r_{\gamma,3}, \mathsf{M}[r_{\gamma,3}], \mathsf{K}[s_{\gamma,3}]) := (r_\sigma \oplus r_\gamma \oplus r_\alpha \oplus r_\beta, & \mathsf{M}[r_\sigma] \oplus \mathsf{M}[r_\gamma] \oplus \mathsf{M}[r_\alpha] \oplus \mathsf{M}[r_\beta], & \mathsf{K}[s_\sigma] \oplus \mathsf{K}[s_\gamma] \oplus \mathsf{K}[s_\alpha] \oplus \mathsf{K}[s_\beta] \oplus \Delta_1 &)
\end{array}$$

(c) P_2 computes the following locally:

$$\begin{array}{llll}
(s_{\gamma,0}, \mathsf{M}[s_{\gamma,0}], \mathsf{K}[r_{\gamma,0}]) := (s_\sigma \oplus s_\gamma, & \mathsf{M}[s_\sigma] \oplus \mathsf{M}[s_\gamma], & \mathsf{K}[r_\sigma] \oplus \mathsf{K}[r_\gamma] &) \\
(s_{\gamma,1}, \mathsf{M}[s_{\gamma,1}], \mathsf{K}[r_{\gamma,1}]) := (s_\sigma \oplus s_\gamma \oplus s_\alpha, & \mathsf{M}[s_\sigma] \oplus \mathsf{M}[s_\gamma] \oplus \mathsf{M}[s_\alpha], & \mathsf{K}[r_\sigma] \oplus \mathsf{K}[r_\gamma] \oplus \mathsf{K}[r_\alpha] &) \\
(s_{\gamma,2}, \mathsf{M}[s_{\gamma,2}], \mathsf{K}[r_{\gamma,2}]) := (s_\sigma \oplus s_\gamma \oplus s_\beta, & \mathsf{M}[s_\sigma] \oplus \mathsf{M}[s_\gamma] \oplus \mathsf{M}[s_\beta], & \mathsf{K}[r_\sigma] \oplus \mathsf{K}[r_\gamma] \oplus \mathsf{K}[r_\beta] &) \\
(s_{\gamma,3}, \mathsf{M}[s_{\gamma,3}], \mathsf{K}[r_{\gamma,3}]) := (s_\sigma \oplus s_\gamma \oplus s_\alpha \oplus s_\beta \oplus 1, & \mathsf{M}[s_\sigma] \oplus \mathsf{M}[s_\gamma] \oplus \mathsf{M}[s_\alpha] \oplus \mathsf{M}[s_\beta], & \mathsf{K}[r_\sigma] \oplus \mathsf{K}[r_\gamma] \oplus \mathsf{K}[r_\alpha] \oplus \mathsf{K}[r_\beta] &)
\end{array}$$

(d) P_1 computes $\mathsf{L}_\alpha^1 := \mathsf{L}_\alpha^0 \oplus \Delta_1$ and $\mathsf{L}_\beta^1 := \mathsf{L}_\beta^0 \oplus \Delta_1$, and then sends the following to P_2:

$$\begin{array}{lll}
G_{\gamma,0} := \mathsf{H}(\mathsf{L}_\alpha^0, \mathsf{L}_\beta^0, \gamma, 0) \oplus (r_{\gamma,0}, & \mathsf{M}[r_{\gamma,0}], & \mathsf{L}_\gamma^0 \oplus \mathsf{K}[s_{\gamma,0}] \oplus r_{\gamma,0}\Delta_1) \\
G_{\gamma,1} := \mathsf{H}(\mathsf{L}_\alpha^0, \mathsf{L}_\beta^1, \gamma, 1) \oplus (r_{\gamma,1}, & \mathsf{M}[r_{\gamma,1}], & \mathsf{L}_\gamma^0 \oplus \mathsf{K}[s_{\gamma,1}] \oplus r_{\gamma,1}\Delta_1) \\
G_{\gamma,2} := \mathsf{H}(\mathsf{L}_\alpha^1, \mathsf{L}_\beta^0, \gamma, 2) \oplus (r_{\gamma,2}, & \mathsf{M}[r_{\gamma,2}], & \mathsf{L}_\gamma^0 \oplus \mathsf{K}[s_{\gamma,2}] \oplus r_{\gamma,2}\Delta_1) \\
G_{\gamma,3} := \mathsf{H}(\mathsf{L}_\alpha^1, \mathsf{L}_\beta^1, \gamma, 3) \oplus (r_{\gamma,3}, & \mathsf{M}[r_{\gamma,3}], & \mathsf{L}_\gamma^0 \oplus \mathsf{K}[s_{\gamma,3}] \oplus r_{\gamma,3}\Delta_1)
\end{array}$$

Input processing:

(5) For each $w \in \mathcal{I}_2$, P_1 sends $(r_w, \mathsf{M}[r_w])$ to P_2, who checks that $(r_w, \mathsf{K}[r_w], \mathsf{M}[r_w])$ is valid. If so, P_2 computes $\lambda_w := r_w \oplus s_w$ and sends $y_w \oplus \lambda_w$ to P_1. Finally, P_1 sends $\mathsf{L}_w^{y_w \oplus \lambda_w}$ to P_2.

(6) For each $w \in \mathcal{I}_1$, P_2 sends $(s_w, \mathsf{M}[s_w])$ to P_1, who checks that $(s_w, \mathsf{K}[s_w], \mathsf{M}[s_w])$ is valid. If so, P_1 computes $\lambda_w := r_w \oplus s_w$ and sends $x_w \oplus \lambda_w$ and $\mathsf{L}_w^{x_w \oplus \lambda_w}$ to P_2.

Circuit evaluation:

(7) P_2 evaluates the circuit in topological order. For each gate $G = (\alpha, \beta, \gamma, T)$, P_2 initially holds $(z_\alpha \oplus \lambda_\alpha, \mathsf{L}_\alpha^{z_\alpha \oplus \lambda_\alpha})$ and $(z_\beta \oplus \lambda_\beta, \mathsf{L}_\beta^{z_\beta \oplus \lambda_\beta})$, where z_α, z_β are the underlying values of the wires.

(a) If $T = \oplus$, P_2 computes $z_\gamma \oplus \lambda_\gamma := (z_\alpha \oplus \lambda_\alpha) \oplus (z_\beta \oplus \lambda_\beta)$ and $\mathsf{L}_\gamma^{z_\gamma \oplus \lambda_\gamma} := \mathsf{L}_\alpha^{z_\alpha \oplus \lambda_\alpha} \oplus \mathsf{L}_\beta^{z_\beta \oplus \lambda_\beta}$.

(b) If $T = \wedge$, P_2 computes $i := 2(z_\alpha \oplus \lambda_\alpha) + (z_\beta \oplus \lambda_\beta)$ followed by $(r_{\gamma,i}, \mathsf{M}[r_{\gamma,i}], \mathsf{L}_\gamma^0 \oplus \mathsf{K}[s_{\gamma,i}] \oplus r_{\gamma,i}\Delta_1) := G_{\gamma,i} \oplus \mathsf{H}(\mathsf{L}_\alpha^{z_\alpha \oplus \lambda_\alpha}, \mathsf{L}_\beta^{z_\beta \oplus \lambda_\beta}, \gamma, i)$. Then P_2 checks that $(r_{\gamma,i}, \mathsf{K}[r_{\gamma,i}], \mathsf{M}[r_{\gamma,i}])$ is valid and, if so, computes $z_\gamma \oplus \lambda_\gamma := (s_{\gamma,i} \oplus r_{\gamma,i})$ and $\mathsf{L}_\gamma^{z_\gamma \oplus \lambda_\gamma} := (\mathsf{L}_\gamma^0 \oplus \mathsf{K}[s_{\gamma,i}] \oplus r_{\gamma,i}\Delta_1) \oplus \mathsf{M}[s_{\gamma,i}]$.

Output revelation:

(8) For each $w \in \mathcal{O}$, P_1 sends $(r_w, \mathsf{M}[r_w])$ to P_2, who checks that $(r_w, \mathsf{K}[r_w], \mathsf{M}[r_w])$ is valid. If so, P_2 outputs $z_w := (z_w \oplus \lambda_w) \oplus r_w \oplus s_w$.

Figure 14: The original two-party WRK in the $\mathcal{F}_{\text{Pre}}$-hybrid model. (excerpted from [44])

[37] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. 2013. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE Symposium on Security and Privacy.*

[38] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. 2009. Secure two-party computation is practical. In *ASIACRYPT.*

[39] Peter Rindal and Mike Rosulek. 2016. Faster malicious 2-party secure computation with online/offline dual execution. In *USENIX Security Symposium.*

[40] University of Virginia Security Research Group. 2015. *Obliv-C: A Language for Extensible Data-Oblivious Computation.* https://oblivc.org/

[41] 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.*

[42] Xiao Wang, Alex Malozemoff, and Jonathan Katz. 2016. *EMP-toolkit: Efficient MultiParty computation toolkit.* https://github.com/emp-toolkit

[43] Xiao Wang, Alex Malozemoff, and Jonathan Katz. 2017. Faster Secure Two-Party Computation in the Single-Execution Setting. In *EUROCRYPT.*

[44] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated garbling and efficient maliciously secure two-party computation. In *ACM CCS.*

[45] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-scale secure multiparty computation. In *ACM CCS.* ACM.

[46] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA.

[47] Ruiyu Zhu and Yan Huang. 2017. JIMU: Faster LEGO-based Secure Computation using Additive Homomorphic Hashes. In *ASIACRYPT.*

[48] Ruiyu Zhu, Yan Huang, and Darion Cassel. 2017. Pool: scalable on-demand secure computation service against malicious adversaries. In *ACM CCS.*