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 are 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 suite of benchmark applications, including an actively-secure four-party logistic 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

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 $\Pi_{\text{abit}}$ (for producing authenticated bits), $\Pi_{\text{aAND}}$ (for producing authenticated ANDs), but most importantly, an undocumented missing step to bridge the gap between the use of function-independent $\Pi_{\text{aAND}}$ and $\Pi_{\text{abit}}$ sub-protocols and its function-dependent pi2pc protocol, which needs extra rounds of communication. In addition, their authenticated garbling protocol $\Pi_{\text{2pc}}$ 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 $\Pi_{\text{abit}}$ would need to be invoked multiple times. However, the global secret $\Delta$ used in $\Pi_{\text{abit}}$ to MAC abits will change across different invocations of $\Pi_{\text{abit}}$. 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. 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 $\Pi_{\text{2pc}}$ (resp. $\Pi_{\text{mpc}}$) to $\Pi_{\text{2pc}}^{\text{ Scalable}}$ (resp. $\Pi_{\text{mpc}}^{\text{ Scalable}}$), 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 $\Pi_{\text{abit}}$ into $\Pi_{\text{abit}}^{\text{ Scalable}}$, where values of $\Delta$ 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 $\Pi_{\text{aAND}}$ into $\Pi_{\text{aAND}}^{\text{ Scalable}}$ 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 (c5.1arge, 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.1

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 \( L_0^i, L_1^i \) (also known as wire-labels) for the \( i \)-th wire in the circuit (\( L_0^i \) encodes a 0-bit and \( L_1^i \) 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

\[
\text{Enc}_{L_k^i, L_k^j}(g(b_i, b_j))
\]

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 \( L_{i,j}, L_j^i \) 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. [35] 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 \( b_i \otimes \lambda_i \) can be revealed to the evaluator to index the garbled entry for decryption. More specifically, let \( \lambda_i, \lambda_j, \lambda_k, \lambda_i \) be the permutation bits of the two input-wires and the output-wire of a gate. Then \( L_0^i \) and \( L_j^i \) should be used to encrypt output key \( L_k^i(\lambda_i, \lambda_j, \lambda_k) \). Thus, a garbled table for \( g = \text{AND} \) can be expressed as the forth column of Table 1. Also, because the evaluator doesn’t know \( \lambda_i, \lambda_j, \lambda_k, \lambda_i \), it is safe to send the garbled table without further permutation. Note that in the random oracle model, \( \text{Enc}_{x, y}(z) \) can be realized as \( H(x, y) \oplus z \) 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 \( \Delta \) and dictate that for every wire \( i \) whose 0-label is \( L_0^i \), its 1-label \( L_1^i \) is always defined as \( L_1^i = L_0^i \oplus \Delta \). Further, for an XOR gate with input wires \( i, j \) and output wire \( k \), the garbler will always set \( L_k^i = L_0^i \oplus L_0^j \). 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, a 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 semi-technical 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 \( O(n|C|) \) 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 \( \Lambda_1 \in \{0, 1\} \). We denote by \([b]^1 \) an authenticated bit \( b \) of party \( P_1 \), which refers to a distributed tuple \((b, M[b], K[b])\) such that \( M[b] = K[b] \odot b\Lambda_1 \) where \( P_1 \) has \( (b, M[b]) \), and \( P_2 \) knows \( K[b] \). We call \( M[b] \in \{0, 1\} \) the Message Authentication Code (MAC) of \( b \). An authenticated AND triple is 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 \( (a_1 \oplus a_2) \cdot (b_1 \oplus b_2) = c_1 \oplus c_2 \). 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, \([\lambda_i]^1, [\lambda_j]^1, [\lambda_k]^1 \) and \([\lambda_i]^2, [\lambda_j]^2, [\lambda_k]^2 \), held by \( P_1 \) and \( P_2 \), respectively, such that \( \lambda_i \oplus \lambda_i^1 = \lambda_i^2 \oplus \lambda_i^2 = \lambda_k \). Now the first question is, to produce the first garbled row of Table 1, how could the circuit generator (call it \( P_1 \)) compute \((\lambda_i \lambda_j \lambda_k) \Delta\) without actually knowing \( \lambda_i, \lambda_j, \lambda_k \)? WRK addresses this challenge by dividing \((\lambda_i \lambda_j \lambda_k) \Delta\) into two XOR-shares \( S_P^1 \) and \( S_P^2 \) such that \( S_P^1 \oplus S_P^2 = (\lambda_i \lambda_j \lambda_k) \Delta\), 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 \((\lambda_1 \lambda_2 \lambda_3) \Delta = \lambda_1, \lambda_2, \lambda_3 \Delta\), then \( P_1, P_2 \) can learn \( c_1, c_2 \), respectively, with \( c_1 \oplus c_2 = \lambda_i \lambda_j \lambda_k \). Consequently, \( P_1 \) can compute \( c_1 \lambda_i \lambda_k \Delta \) from \( c_1, \lambda_i \lambda_k \) and \( \Delta \), all

1https://github.com/nanoPMPC/nanoPMPC.
Table 1: Garbling in Honest-but-curious Adversary Model

<table>
<thead>
<tr>
<th>$b_i \oplus \lambda_i$</th>
<th>$b_j \oplus \lambda_j$</th>
<th>$b_k \oplus \lambda_k$</th>
<th>Point &amp; Permute</th>
<th>With Random Oracle $H$</th>
<th>Free-XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$z_{00} = \lambda_i \lambda_j \lambda_k$</td>
<td>$\text{Enc}<em>{\lambda_i}^{k_0} \left( L_k^{z</em>{00}} \cdot z_{00} \right)$</td>
<td>$H_{0,0} \oplus (L_k^{z_{00}} \cdot z_{00})$</td>
<td>$H_{0,0} \oplus (L_k^{z_{00}} \cdot z_{00})$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$z_{01} = \lambda_i \lambda_j \lambda_k$</td>
<td>$\text{Enc}<em>{\lambda_i}^{k_0} \left( L_k^{z</em>{01}} \cdot z_{01} \right)$</td>
<td>$H_{0,1} \oplus (L_k^{z_{01}} \cdot z_{01})$</td>
<td>$H_{0,1} \oplus (L_k^{z_{01}} \cdot z_{01})$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$z_{10} = \lambda_i \lambda_j \lambda_k$</td>
<td>$\text{Enc}<em>{\lambda_i}^{k_0} \left( L_k^{z</em>{10}} \cdot z_{10} \right)$</td>
<td>$H_{1,0} \oplus (L_k^{z_{10}} \cdot z_{10})$</td>
<td>$H_{1,0} \oplus (L_k^{z_{10}} \cdot z_{10})$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$z_{11} = \lambda_i \lambda_j \lambda_k$</td>
<td>$\text{Enc}<em>{\lambda_i}^{k_0} \left( L_k^{z</em>{11}} \cdot z_{11} \right)$</td>
<td>$H_{1,1} \oplus (L_k^{z_{11}} \cdot z_{11})$</td>
<td>$H_{1,1} \oplus (L_k^{z_{11}} \cdot z_{11})$</td>
</tr>
</tbody>
</table>

Table 2: WRK’s Authenticated Garbling in Malicious Adversary Model

<table>
<thead>
<tr>
<th>Share of $P_1$’s Garbled Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_{0,0} \oplus \left( L_k^{\lambda_i} \oplus (c_1 \oplus \lambda_i^2 \lambda_j) \Delta \oplus K[c_2] \oplus K[\lambda_j^2] \right)$, c_1 \oplus \lambda_i^2 \lambda_j^2, M[c_1] \oplus M[\lambda_j^2]</td>
</tr>
<tr>
<td>$H_{0,1} \oplus \left( L_k^{\lambda_i} \oplus (c_1 \oplus \lambda_i^2 \lambda_j \lambda_k) \Delta \oplus K[c_2] \oplus K[\lambda_j^2] \oplus K[\lambda_k^2] \right)$, c_1 \oplus \lambda_i^2 \lambda_j^2 \lambda_k, M[c_1] \oplus M[\lambda_j^2] \oplus M[\lambda_k^2]</td>
</tr>
<tr>
<td>$H_{1,0} \oplus \left( L_k^{\lambda_i} \oplus (c_1 \oplus \lambda_i^2 \lambda_j \lambda_k) \lambda \Delta \oplus K[c_2] \oplus K[\lambda_j^2] \oplus K[\lambda_k^2] \right) \oplus \Delta$, c_1 \oplus \lambda_i^2 \lambda_j^2 \lambda_k, M[c_1] \oplus M[\lambda_j^2] \oplus M[\lambda_k^2]</td>
</tr>
<tr>
<td>$H_{1,1} \oplus \left( L_k^{\lambda_i} \oplus (c_1 \oplus \lambda_i^2 \lambda_j \lambda_k) \lambda \Delta \oplus K[c_2] \oplus K[\lambda_j^2] \oplus K[\lambda_k^2] \right) \oplus \Delta$, c_1 \oplus \lambda_i^2 \lambda_j^2 \lambda_k, M[c_1] \oplus M[\lambda_j^2] \oplus M[\lambda_k^2]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Share of $P_2$’s Garbled Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\left( M[c_2] \oplus M[\lambda_j^2] \right)$, M[c_2] \oplus M[\lambda_j^2]</td>
</tr>
<tr>
<td>$\left( M[c_2] \oplus M[\lambda_j^2] \right) \oplus \left( M[c_2] \oplus M[\lambda_j^2] \right)$, M[c_2] \oplus M[\lambda_j^2] \oplus M[\lambda_j^2]</td>
</tr>
<tr>
<td>$\left( M[c_2] \oplus M[\lambda_j^2] \right) \oplus \left( M[c_2] \oplus M[\lambda_j^2] \right)$, M[c_2] \oplus M[\lambda_j^2] \oplus M[\lambda_j^2]</td>
</tr>
</tbody>
</table>

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

To resolve this, WRK exploited the fact that $P_2$ already knows $M[c_2]$, $M[\lambda_j^2]$, both generated from the same $\Delta$ ($P_1$’s global secret for authenticating $P_2$’s bits). Because $K[b] \oplus M[b] = b \lambda_1$, for any authenticated bit $b$, it suffices to require $P_1$ to set $\Delta = \lambda_1$, and define

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

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

so that $S^P_1 \oplus S^P_2 = (\lambda_i \lambda_j \lambda_k) \Delta$ while the parties can each locally compute $S^P_1$ and $S^P_2$, respectively. Finally, to prevent a malicious $P_1$ from replacing $c_1 \oplus \lambda_i^2 \lambda_j$ with an arbitrary bit, WRK requires $P_1$ to provide $M[c_1] \oplus M[\lambda_i^2]$, the MAC of $c_1 \oplus \lambda_i^2 \lambda_j$, so that $P_2$ can verify the correctness of a garbled row. Further, observing that the MACs are XOR-homomorphic and $\lambda_i \lambda_j \lambda_k = \lambda_j \lambda_i \lambda_k$, $\lambda_i \lambda_j \lambda_k = \lambda_j \lambda_i \lambda_k$, and $\lambda_i \lambda_j \lambda_k = 1 \oplus \lambda_i \oplus \lambda_j \oplus \lambda_k$, 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 $\Pi_{\text{aAND}}$, dominates the overall cost of the WRK protocols. WRK’s $\Pi_{\text{aAND}}$ works in two high-level steps:

1. Generating leaky-aANDs using $\Pi_{\text{aAND}}$. A leaky-aAND has the same property as aAND except that a cheating party is able to correctly guess a honest party’s first bit 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 omissions 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 found 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.

Note that even on a nowadays powerful cloud server with 128GB memory, (roughly the $c5. 18x1ar.ge$ instance at $3/3/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 an 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...
4 Party LAN
2 Party WAN
1B
32M
1M
4B
2B
4M
16M
256M
128M
512M
8M
0.5M

Thus, new secure mechanisms are needed to better reconcile the
beiteness tradeoff. We chose to underestimate by setting $k := \min_{\text{all observed}} \left( \frac{m_1 - m_2}{g_1} \right)$ and $\hat{b} := \min_{\text{all observed}} \left( \frac{m_1 - m_2}{g_1 - g_2} \right)$.

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

In order to quantify the performance penalty, we have tried
to naive modification to WRK and experimentally measured
its performance in various setups. As Figure 2 shows, the naive
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 pro-
tocols 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 com-
nunity 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.

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 $k := \min_{\text{all observed}} \left( \frac{m_1 - m_2}{g_1} \right)$ and $\hat{b} := \min_{\text{all observed}} \left( \frac{m_1 - m_2}{g_1 - g_2} \right)$.

Figure 2: Performance degradation of naively 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.

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 gen-
erated 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 par-
allel. (2) The call to abit in step (2) of protocol $\Pi_{2pc}$ (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 $\Pi_{aAND}$
is constant-round but uses $O(|C_f|)$-space. (2) Even if WRK’s $\Pi_{aAND}$
was efficiently executable in constant-space, the way aAND
gets used in WRK’s $\Pi_{2pc}$ also prevents $\Pi_{2pc}$ 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 $\Pi_{2pc}$ [44] as an ex-
ample. The ideal aAND of $\Pi_{2pc}$ (which we excerpted from [44]) as Figure 5) is actually different from their $F_{aAND}$ func-
tionality (which we copied below for easy comparison):

$F_{aAND}$

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

The gap is that: $F_{aAND}$ does not allow the participants to control
the values of the abits, whereas the aAND of $F_{2pc}$ does allow $P_1$
ways to construct the function. WRK obtains this wire-connection
P + bytes per AND because n expanded for L data to run through the AES cipher.

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 λ, β 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 λ, β are consistent with
those returned by F_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-JMU 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-JMU 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.  

```c
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 `x` 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-JMU, 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 each 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 $\Pi_{\text{abits}}$ and $\Pi_{\text{aAND}}$ in the preparation phase are plagued by the space-round dilemma. We first show that naïvely converting WRK’s $\Pi_{\text{abits}}$ 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 $\Pi_{\text{abits}}$. The basic idea to scale up $\Pi_{\text{abits}}$ is to break the constant-round, single-batch of $\ell$ abits generation process into $\ell/k$ batches, each producing $k$ abits ($k$ is set based on available resource). However, realizing this by naïvely calling WRK’s $\Pi_{\text{abits}}$ 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) $\ell$ random correlated message pairs $\{(m_i^0, m_i^1)\}_{1 \leq i \leq \ell}$ so that $\forall \Delta \in \{0, 1\}^n, \forall i, m_i^0 \oplus m_i^1 = \Delta$; and returns $P_2$ (the RCOT receiver) $\{(b, m_i^b)\}_{1 \leq i \leq \ell}$. Note that there is no guarantee that the same $\Delta$ 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-$\Delta$ attacks!

To see how an inconsistent-$\Delta$ attack works, note that with all but negligible probability, $P_1$ as an RCOT sender will get two values of the correlation-difference, say $\Delta_1$ and $\Delta_2$, from two calls to RCOT protocol. So with high probability, in some iterations of step (4a), the abit $r_a$ of a particular aAND is authenticated with $\Delta_2$, whereas the rest of the abits involved in the same aAND call are authenticated with $\Delta$. By definition, it is easy to verify that an aAND will fail if $r_a = 1$ but succeed if $r_a = 0$. Since everyone knows which abits are associated with which $\Delta$ 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** \(\Pi_{\text{Scalable}}\). To avoid the inconsistent-\(\Delta\) issue, we propose two minor modifications to WRK’s \(\Pi_{\text{ablit}}\): (1) we change the underlying RCOT protocol so that it takes a predefined \(\Delta\) as input from the sender which is used to form random correlated messages; and (2) to prevent an adversary from deliberately using different \(\Delta\) in different RCOT batches, we add a consistency check step at the end of every batch of RCOTs to ensure that identical \(\Delta\) values are used across different batches. Our abit protocol, called \(\Pi_{\text{Scalable}}\), can efficiently produce an arbitrary number of abits using constant-space. \(\Pi_{\text{Scalable}}\) is specified in Figure 8.

**Fast, Scalable** \(\Pi_{\text{Scalable}}\). 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 \(O(|C_p| \cdot B)\) leaky-ANDs before randomly grouping every \(B\) leaky-ANDs into a bucket, we can carry out the random grouping always within a pool of \(p\) leaky-ANDs while refilling a used leaky-AND as soon as it is marked to be converted into a fully-secure AND triple. Thus, WRK’s cut-and-choose-based \(\Pi_{\text{AND}}\) is modified to efficiently run in constant space using pool-based cut-and-choose. Comparing to Pool-JMU, the differences here are: (1) every leaky-AND is checked for validity with a constant fault-detection rate \(1/2\), and (2) every leaky-AND can be checked and combined with other leaky-ANDs 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 \(\Pi_{\text{AND}}\) runs.) Our aAND protocol \(\Pi_{\text{Scalable}}\) can efficiently generate any number of aANDs using constant-space. \(\Pi_{\text{Scalable}}\) 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 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.

**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 (\(\Pi_{\text{ablit}}\)) without overly penalizing its performance due to network roundtrips. As a proof-of-concept, we formalize the subset-of-C circuit description language using the grammar below:

| Programs | \(p \equiv d_1; d_2; \ldots; f_1; f_2; \ldots; b\) |
| Variable Declarations | \(d \equiv \text{bit } x | \text{bit[n]} \times x) |
| Function Declarations | \(F \equiv f(d_1, d_2, \ldots)(b)\) |
| Blocks | \(b \equiv d_1; d_2; \ldots; x_1; x_2; \ldots\) |
| Circuit Variables | \(X \equiv x | x[i]\) |
| Iteration Variables | \(i \equiv 0 | 1 | 2 \ldots\) |
| Operators | \(\odot \equiv \text{nand } | \text{nor } | \ldots\) |
| Statements | \(s \equiv X = \text{true} | X = \text{false}\) |

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

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) unravel 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 syntactic of the resulting scoped circuit description language can thus be simplified to the following:

Programs p ::= b
Blocks b ::= d1;d2;...;s1;s2;...
Variable Declarations d ::= x = true | x = false
| x = x1 ∘ x2 | {b}
Statements s ::= x = true | x = false
| x = x1 ∘ x2 | {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;
  t = t ∨ x;
  r = r ∨ t;
}
rs0 = r;
{ // second call to f
  bit x;
  x = xs1;
  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 ← ℓ to produce a new environment ρ[x ← ℓ]. 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:

(ρ[x ← ℓ](y) =
  ℓ if x = y
  ρ(y) otherwise

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 σ[ℓ ← 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:

D[bit x](ρ, σ) = (ρ[x ← ℓ], σ[ℓ ← false])

where ℓ is fresh in σ.

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

D[bit x](ρ, σ) = D[bit x](ρ, σ)

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 B[ ] and S[ ] defined below. Each of these functions takes a current environment and store and returns only a store.

B[ds](ρ, σ) = S[ss](D[ds](ρ, σ))
S[x = true](ρ, σ) = σ[p(x) ← true]
S[x = false](ρ, σ) = σ[p(x) ← false]
S[x = y ⊙ z](ρ, σ) = σ[p(x) ← σ(p(y) ⊙ σ(p(z)))]
S[ ](ρ, σ) = B[ ](ρ, σ)

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, y, z, x, y, t. Upon exiting the scope, the environment reverts to only having entries for r, rs0, rs1, xs0, xs1, y, z, x, y, t. 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 Π2gc, 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\". 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:

```plaintext
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 \(\eta\), between the local and non-local instances of \(r\). Specifically, upon exiting the inner scope, an update operation uses \(\eta\) to copy the contents of the local instance of \(r\) back into the non-local instance. See below for more explanation about \(\eta\).

Second, we split the semantic definition of staged execution into two functions that communicate through a shared queue containing delayed \textit{thanks} [16] or \textit{closures} [23]. One function (see Figure 5) traverses the program as usual but instead of executing expressions like \("\text{bit } x = y ⊕ z\"\) immediately, it instead allocates a fresh location \(\ell\), extends the environment with a binding of the declared variable to \(\ell\), extends the store with a marker \(●\) indicating that \(\ell\) is currently uninitialized, constructs a closure \(\ell \leftarrow \rho(y) ⊕ \rho(z)\) 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 \(\eta\) is maintained to connect local and non-local instances of the same variable. Note that updates to \(\eta\) are in-place, namely, \("\eta[x/\ell]\"\) denotes updating the location where the non-local variable \(x\)'s latest value is stored to \(\ell\). The update operations introduced in the previous stage use this \(\eta\) 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, \(\eta\) is updated when processing declarations (by \(C_\eta[\Gamma]\)) and used for interpreting \"update\" instructions (by \(C_s[\Gamma]\)).

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 \(0x100\) for \(x\) 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 \(0x100\). 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:

\[
\begin{align*}
S\{ x = \text{true} \}(\rho, \sigma) &= \sigma[\rho(x) \leftarrow \text{true}] \\
\end{align*}
\]

and compare it with the new staged semantics for assignments:

\[
\begin{align*}
C_s\{ x = \text{true} \}(\rho, \sigma, \eta, L) &= \langle \sigma, \eta, L + (\rho(x) \leftarrow \text{true}) \rangle \\
\end{align*}
\]

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

\[
\begin{align*}
D\{ \text{bit } x = \text{true} \}(\rho, \sigma) &= \langle \rho[x \leftarrow \ell], \sigma[\ell \leftarrow \text{true}] \rangle \\
\end{align*}
\]

and the new staged semantics is:

\[
\begin{align*}
C_d\{ \text{bit } x = \text{true} \}(\rho, \sigma, \eta, L) &= \langle \rho[x \leftarrow \ell], \sigma[\ell \leftarrow ●], \eta, L + (\ell \leftarrow \text{true}) \rangle \\
\end{align*}
\]

where \(\ell\) is fresh in \(\sigma\) in both cases. Again the difference is that the update \(\ell \leftarrow \text{true}\) is associated with \(L\) instead of \(\sigma\).

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 \(L\) are performed in the original order, i.e., by treating \(L\) 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
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.

<table>
<thead>
<tr>
<th>line #</th>
<th>( \rho )</th>
<th>( \sigma )</th>
<th>( \eta )</th>
<th>( \mathcal{L} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>1</td>
<td>{ r } \leftarrow 0x1000 }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>2</td>
<td>{ rs0 } \leftarrow 0x1004 }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>3</td>
<td>{ rs1 } \leftarrow 0x1008 }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>4</td>
<td>{ xs0 } \leftarrow 0x100c }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>5</td>
<td>{ xs1 } \leftarrow 0x1010 }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>6</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>7</td>
<td>{ x } \leftarrow 0x1014 }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>8</td>
<td>{ t } \leftarrow 0x1018 }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>9</td>
<td>{ t } \leftarrow 0x101c }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>10</td>
<td>{ r } \leftarrow 0x1020 }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>11</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>12</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>13</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>14</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>15</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>16</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
</tbody>
</table>

If a staged execution kicks in here, then it updates
\( \sigma_{16} = \{ (0x1000, \text{false}), (0x1014, \text{true}), (0x101c, \text{false}), (0x1020, \text{false}), (0x1024, \text{false}), (0x1028, \text{true}) \} \) and \( \mathcal{L}_{16} = \{ \} \).

\[ \langle \rho, \sigma, \eta, \ell \leftarrow \text{true} \rangle \mathcal{L} \propto = \langle \rho, \sigma[\ell \leftarrow \text{true}], \eta, \mathcal{L} \rangle \]
\[ \langle \rho, \sigma, \eta, \ell \leftarrow \text{false} \rangle \mathcal{L} \propto = \langle \rho, \sigma[\ell \leftarrow \text{false}], \eta, \mathcal{L} \rangle \]
\[ \langle \rho, \sigma, \eta, \ell \leftarrow \ell_1 \cup \ell_2 \rangle \mathcal{L} \propto = \langle \rho, \sigma[\ell \leftarrow \sigma(\ell_1) \cup \sigma(\ell_2)], \eta, \mathcal{L} \rangle \]

Figure 6: Formal semantics for executing delayed closures.

The sequence of atomic actions \( \ell \leftarrow \nu_i \) 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 \( \rho \) and the delayed closures \( \mathcal{L} \) contain all the live locations. (The locations in the association \( \eta \) are already in the environment \( \rho \).) Any method that shrinks \( \sigma \) by collecting all other locations, i.e., all locations that do not occur in \( \rho \) 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

\( \sigma_{16} = \{ (0x1000, \text{false}), (0x1014, \text{true}), (0x101c, \text{false}), (0x1020, \text{false}), (0x1024, \text{false}), (0x1028, \text{true}) \} \) and \( \mathcal{L}_{16} = \{ \} \).

Figure 6: Formal semantics for executing delayed closures.
We need a rule:
\[ \langle \rho, \sigma | \ell \leftarrow v, \eta, \mathcal{L} \rangle = \langle \rho, \sigma, \eta, \mathcal{L} \rangle \]
if \( \ell \) occurs in neither \( \rho \) nor \( \mathcal{L} \). The rule specifies that the configuration consisting of the environment, store, association \( \eta \), and closures on the left is indistinguishable from the configuration on the right. First the location \( \ell \) 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 \( \sigma \) without the location \( \ell \). 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 \( \ell \) is already present in either \( \rho \) 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 \( \Pi_{2pc}^{abst} \)

We modify WRK’s \( \Pi_{2pc} \) so that it uses \( \mathcal{F}_{\text{aAND}} \) instead of \( \mathcal{F}_{\text{pke}} \)’s aAND. This is done by replacing the step (4a) of Figure 14 with the following:

- (4a) \( P_1 \) and \( P_2 \) invoke \( \mathcal{F}_{\text{aAND}} \). 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 \( r_{\sigma} \oplus s_{\sigma} \) 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 \( r_\sigma \oplus s_\sigma = \lambda_\sigma \land \lambda_\beta \).

The modified step securely aligns two secret bits \( a, b \), each of which is already divided into two abits, e.g., \([a_1]^1, [a_2]^1\), \([b_1]^2, [b_2]^2\). 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 \( P_2 \) resets \( K[a_1] = K[a_1] \oplus \Delta_2 \).

**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 \( \mathcal{F}_{\text{abst}} \) and \( \mathcal{F}_{\text{aAND}} \).

Intuitively, because both \( x_1 \) and \( r_\sigma \) are uniform, revealing \( x_1 \oplus r_\sigma \) won’t give \( P_2 \) any advantage in guessing either \( x_1 \) or \( r_\sigma \). The proposition above is proved in a hybrid model with ideal \( \mathcal{F}_{\text{abst}} \) and \( \mathcal{F}_{\text{aAND}} \). 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 \( \Pi_{\text{Scalable}}^{abst} \) in the \( \mathcal{F}_{\text{Leaky-RCOT}} \)-hybrid model. \( \mathcal{F}_{\text{Leaky-RCOT}} \) functionality is given in Figure 7, which can be efficiently implemented based on actively-secure OT extension by Keller et al. [18].

**Proposition 5.4.** \( \Pi_{\text{Scalable}}^{abst} \) of Figure 8 can securely realize \( \mathcal{F}_{\text{abst}} \).

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

**Proposition 5.4.** \( \Pi_{\text{Scalable}}^{abst} \) can securely realize \( \mathcal{F}_{\text{abst}} \).
Therefore, once $P_1$’s input is $\Lambda$ and $P_2$ has no input. 

Output: $P_1$ gets (up to) infinitely many uniform $K_i$ where $K_i \in \{0, 1\}^n$, $P_2$ gets (up to) infinitely many random $(b_i, M_i)$ where $M_i \in \{0, 1\}^n$ such that $M_i = \Lambda \oplus b_i \cdot K_i$ 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 $\Delta' \in \{0, 1\}^{n\times n}$ subject to $A \times \Delta' = \Delta$. $P_1$ commits $A$.

(2) $P_2$ uses $\Delta'$ to run $\mathcal{F}_{\text{leaky-RCOT}}(n + s, s)$ with $P_2$ so that $P_1$ gets $\{K_i\}_{i \in [s]}$ and $P_2$ obtains $\{(b_i, M_i)\}_{i \in [s]}$.

(3) $P_1$ opens $A$ to $P_2$. $P_1$ computes $K_i' = A \times K_i$ for all $i \in [s]$. $P_2$ computes $M_i := A \times M_i'$ 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 $\Delta' \in \{0, 1\}^{n\times n}$ subject to $A \times \Delta' = \Delta$. $P_1$ commits $A$.

(b) $P_1$ uses the refreshed $\Delta'$ to run $\mathcal{F}_{\text{leaky-RCOT}}(n + s, p + s)$ (with $P_2$) so $P_1$ learns $\{K_i\}_{i \in [p]}$ and $P_2$ learns $\{(b_i, M_i)\}_{i \in [p]}$.

Then $P_1$ opens the refreshed $A$ to $P_2$. For every $i \in \{p, \ldots, p + s\}$, $P_1$ and $P_2$ are authenticated parties. $P_1$ first computes $K_i := A \times K_i'$ and $P_2$ computes $M_i' := A \times M_i'$.

(c) Repeat the following check $s$ times (for $i \in [s]$).

(i) $P_2$ sends $b_i' + b_{j+i}$ to $P_1$.

(ii) $P_1$ computes $K_i' = K_i', K_i + K_i + K_i'$. $P_2$ computes $M_i' = M_i', M_i + M_i + M_i'$.

P in invoke $\mathcal{F}_{\text{equal}}$ using $K_i$ and $M_i$, respectively. They abort if $\mathcal{F}_{\text{equal}}$ returns unequal.

(d) To refresh $\{\{K_i', b_i', M_i'\}_{i \in [s]}\}$ for every $i \in \{s\}$.

(i) $P_1$ sets $K_i' := K_i$.

(ii) $P_2$ sets $b_i' := b_{i+1}$.

(e) $P_1$ outputs $\{K_i\}_{i \in [p]}$, $P_2$ outputs $\{b_i, M_i\}_{i \in [p]}$.

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

\[ \text{Set } i \triangleq \infty, b_0 \triangleq \cdots \triangleq b_0 = 2, \text{ and repeat the steps below until it exits from (2)}: \]

(1) Use the recurrence above to compute $P_B(0, k)$ for all $0 \leq k \leq b_0$.

(2) If $\max_{k \in \{1, \ldots, b_0\}} 2^{-k} \cdot P_B(p, k) \leq 2^{-b}$, then output $B$ and halt; otherwise, set $B := B + 1$.

(3) If $B > p$, exit with output $\bot$.

**Proposition 5.5.** Protocol $\Pi^{n\times(n+s)}_{\text{abt}}$ of Figure 9 securely realizes $\mathcal{F}_{\text{abt}}$.

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.

\section{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 $P_1$, denoted as $[b_1]$, refers to the tuple $\{b_1, [M_1[j]]_{j \neq i}, [K_1[j]]_{j \neq i}\}$ where $P_1$ holds $\{b_1, [M_1[j]]_{j \neq i}\}$ and $P_i$ ($\forall i \neq i$) holds $K_1[i]$. Similarly, an $n$-party sub-protocol is used to securely compute $n$-party authenticated AND triples

\[ (\{a\}^n \oplus \cdots \oplus \{a\}^n) \cdot (\{b\}^n \oplus \cdots \oplus \{b\}^n) = \{c\}^n \]

where 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 $\Pi_{\text{mpc}}$ so it works in a hybrid model with ideal $\mathcal{F}^n_{\text{abt}}$ and $\mathcal{F}^n_{\text{abt}}$. Then we show how to efficiently scale up $\mathcal{F}^n_{\text{abt}}$ and $\mathcal{F}^n_{\text{abt}}$. For the convenience of the readers, we copied the definition of WRK’s $\mathcal{F}^n_{\text{abt}}$ and $\mathcal{F}^n_{\text{abt}}$ below.

\[ \mathcal{F}^n_{\text{abt}} \]

**Honest Case:** The box receives (input, $i, \ell$) from all parties and picks random bit-string $x \in \{0, 1\}^\ell$. For each $j \in \{0, \ldots, m\}$, the box picks random $K_x[j]$, and computes $[M_K[x]] := [K_x[j] \oplus x_K[j]]_{k \neq i}$ and sends them to parties. That is, for each $j \in \{0, \ldots, m\}$, it sends $\{M_K[x]\}_{k \neq i}$ to $P_i$ and sends $K_x[j]$ to $P_k$ for each $k \neq i$.

**Corrupted parties:** A corrupted party can choose their output from the protocol.
6.1 Fixing WRK’s $\Pi_{mpc}$

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

4 (a) All $P_i(1 \leq i \leq n)$ jointly invoke $F_{\text{AND}}$ so that $P_i$ receives $[x_1^i]^l, [x_2^i]^l, [x_3^i]^l$ where $\bigoplus_{i} x_1^i \land \bigoplus_{i} x_2^i = \bigoplus_{i} x_3^i$. $P_i$ securely aligns $\bigoplus_{i} x_1^i$ with $\bigoplus_{i} r_{i1}^f$, and securely aligns $\bigoplus_{i} x_2^i$ with $\bigoplus_{i} r_{i2}^f$. Finally, $P_i$ sets $r_{i3}^f := x_3^i \land \bigoplus_{i} r_{i1}^f = \lambda_{ia} \land \lambda_{ib}$.

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. $a_d(1 \leq d \leq l)$ and $b_d(1 \leq d \leq l)$. The alignment is done by letting $P_i$ broadcast $[a_d \oplus b_d]^l$. This allows them to check whether $\bigoplus_{i} (a_d \oplus b_d) = 0$ which is equivalent to $\bigoplus_{i} a_d = \bigoplus_{i} b_d$. If the equality doesn’t hold, then $P_i$ flips $a_d$ while every $P_i(2 \leq i \leq n)$ setting $K_i[a_d] := K_i[a_d] \oplus \Delta_i$. Since $x_1^i$ and $r_{i3}^f$ are uniform, revealing $X_i$ won’t leak any information on $x_1^i$ or $r_{i3}^f$.

6.2 Efficiently Scalable $\Pi_{\text{Scalable}, n}^\text{abit}(s, p)$

WRK [45]’s $F_{\text{Scalable}, n}^\text{abit}$ is built from the two-party $F_{\text{abit}}$. So we can scale up their $\Pi_{abit}^n$ protocol by instantiating every call to $F_{\text{abit}}$ in their $\Pi_{abit}^n$ using our $\Pi_{\text{Scalable}, n}^\text{abit}$.

Note that $\Pi_{\text{Scalable}, n}^\text{abit}(s, \ell)$ in Figure 10 guarantees that each $P_i$ uses consistent choice bits across corresponding instances $F_{\text{abit}}$. $F_{\text{abit}}$ guarantee that a consistent $\Delta$ is used between different executions.

6.3 Efficiently Scalable $\Pi_{\text{Scalable}, n}^\text{abit}(s, p)$

The efficient scalability of our multi-party $\Pi_{\text{Scalable}, n}^\text{abit}(s, p)$ protocol is derived from the same idea of employing pool-based cut-and-choose. The multi-party $\Pi_{\text{Scalable}, n}^\text{abit}$ is the same as the two-party $\Pi_{\text{Scalable}, n}^\text{abit}$, 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 c5.1large to run most of our experiments. The c5.1large 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.

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 descent (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-server setting.

7.2 Performance

As is evidenced from the benchmark applications (Table 5), our two-party protocols executed at fairly consistent speeds: roughly
We considered three kinds of memory budgets in combination with will shift from roundtrip latency to computation. This shows that this is because switching from (200Mbps, 40ms) to (20Mbps, 40ms) runs 86% faster than ours while either reducing or boosting the 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 opensourcens 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

[8] Ivan Damgård, Marcel Keller, Enrique Larraza, Valerio Pastro, Peter Scholl, and Nigel Smart. 2013. Practical covertly secure MPC for dishonest majority—or: breaking the SPDZ limits. In ESORICS.
[27] Yehuda Lindell, Benny Pinkas, Nigel P Smart, and Ariadnya 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.
Table 5: Experiments on selected applications

<table>
<thead>
<tr>
<th>Protocol</th>
<th>This Work (B = 4)</th>
<th>Pool-JIMU (B = 6)</th>
<th>WRK (only if</th>
<th>This Work (B = 3)</th>
<th>Pool-JIMU (B = 4)</th>
<th>WRK (only if</th>
<th>This Work (B = 3)</th>
<th>Pool-JIMU (B = 4)</th>
<th>WRK (only if</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 MB 40 ms</td>
<td>795.03</td>
<td>561</td>
<td>1.73K</td>
<td>2.75K</td>
<td>787.72</td>
<td>2.75K</td>
<td>3.12K</td>
<td>925.04</td>
<td>3.23K</td>
</tr>
<tr>
<td>200 MB 40 ms</td>
<td>825.18</td>
<td>4.12K</td>
<td>2.76K</td>
<td>6.94K</td>
<td>6.21K</td>
<td>12.94K</td>
<td>20.94K</td>
<td>7.20K</td>
<td>22.34K</td>
</tr>
<tr>
<td>2 GB  &lt;1 ms</td>
<td>20.27K</td>
<td>46.6K</td>
<td>20.53K</td>
<td>46.66K</td>
<td>69.77K</td>
<td>46.84K</td>
<td>49.34K</td>
<td>88.01K</td>
<td>50.64K</td>
</tr>
</tbody>
</table>

| Bandwidth (Byte/AND) | 505 | 3.3K | 504 | 380 | 2.2K | 504 | 379 | 1.8K | 378 |

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

Figure 11: Memory’s impact on performance

Figure 12: Scale up with # of parties

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


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


[41] Xiao Wang, Yan Huang, Yongan Zhao, Haixu Tang, Xiaofeng Wang, and Diyye Bu. 2015. Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In ACM CCS.


