zkPass Protocol based on TLS, MPC and ZKP

Technical Whitepaper 2.0

zkPass Team

May 2023

This whitepaper is tailored to technically proficient readers interested in integrating zkPass into their development stacks and who want to understand this technology's reasoning and strategic decisions. It aims to provide insight into the conceptual frameworks that drive zkPass, making it valuable for developers involved in component creation, tool development, or building businesses around zkPass within the broader ecosystem.

Abstract

In this whitepaper, we introduce the zkPass protocol, an innovative cryptographic structure leveraging the power of three-party Transport Layer Security (TLS), Multi-Party Computation (MPC), and Interactive Zero-Knowledge Proof (IZK). The ubiquity of HTTPS has facilitated users to access private data with robust end-to-end confidentiality and integrity. However, a shortfall exists in that they cannot inherently substantiate to third parties that the data originated from a specific website. To rectify this, we propose the zkPass protocol. This empowers users to convincingly demonstrate that data accessed via HTTPS was obtained from a particular website, asserting statements about this data within a zero-knowledge context, thereby preserving privacy and averting the exposure of sensitive information. Serving as a conduit between Web 2.0 and Web 3.0, the zkPass protocol liberates private data from the confines of centralized web servers, paving the way for seamless migration into a decentralized era.

Notation and Preliminaries

- Prover(P): or user, or client who is required to generate proof for his statement.
- Verifier(V): or platform, which is responsible for verifying the Prover's statement.
- Node(N): zkPass Node, is a part of the protocol execution used to verify the authenticity and integrity of the Prover's private data.
- DataSource(S): stands for https server, is the trusted data source that owns the Prover's data.

1 Overview of the Solution

In the traditional data validation and confirmation process, the Prover submits their information to the Verifier. The Verifier, in turn, retrieves this data and performs authentication checks in collaboration with the DataSource. Thus, the Verifier serves merely as an intermediary or broker in this model.

Each party faces unique challenges in this scenario: for the Prover, there's a risk of disclosing excessive personal information; for the DataSource, while it's a trusted data provider, it's incapable of offering personalized verification services; and for the Verifier, they're privy to all the customers' private data, gaining total access, which presents significant potential risks of data leakage.

We propose a new approach that repositions these three entities, positioning the Prover between the Verifier and the DataSource. Rather than the traditional method, the Prover uses their access token to directly locate and retrieve their data from the DataSource, subsequently generating a Zero-Knowledge Proof (ZKP) for the Verifier to check. This process ensures the Verifier remains unaware of the Prover's personal information.

In order to implement this structure, we incorporate 3P-TLS, MPC, and IZK technologies.

1.1 3P-TLS

Transport Layer Security (TLS) is the secure protocol for HTTPS, supported by almost all DataSources. It is a two-party protocol designed for client/server structure. We have built the 3P-TLS protocol based on elliptic curve DH protocol and combined it with MPC and Oblivious Transfer(OT) to prevent cheating.

1.2 MPC

One challenge we face is that the Prover may forge proof information provided by the DataSource to deceive the Verifier. The solution to this problem is through MPC, where both Prover and Verifier hold half of a session MAC key, which is a data integrity key used to maintain data integrity. Since the Prover cannot forge or tamper with information responses provided by the DataSource, it cannot deceive the Verifier. MPC can also prevent the Verifier from knowing any private information about the Prover. During MPC handshake, there is no encryption key (data confidentiality key) for the Verifier, so it cannot decrypt any data and therefore does not know any private information for the Prover.

1.3 IZK

After obtaining information from the DataSource, Prover needs to prove certain statements of the response in a secure and private manner. We use Zero-Knowledge Proofs(ZKP) to achieve this goal. More specifically, we use Interactive Commit and Prove Zero-Knowledge Proofs(ICP-ZKP) to deal with the large scale of circuit. Prover gives the commitment as $d = commit(m, r)$, where m is the message and r is the randomness. Then Verifier check $T/F = verify(m, r, d)$. d wouldn't reveal any information about m, and Prover can't find different and m' (and r and r') such that $verify(m, r, d) = verify(m', r', d) = T$. During the protocol, Prover needs to commit its private witness, then prove the input wires and the computation of gate one by one, and continue to commit the output wires until the final one.

2 TLS

2.1 Overview

TLS is one of the most widely used protocols for secure communication over the Internet. It encrypts data from plaintext to ciphertext and vice versa, providing data security and privacy by encrypting traffic to prevent sensitive data from being leaked by third parties. The process consists of two sub-protocols: handshake and record layer. The goal of the first sub-protocol is to negotiate a secure key between two endpoints, while the second uses an agreed key to protect communication.

2.2 Handshake

We refer to the endpoint initiating the connection as the ”client,” while the term ”server” denotes the responding endpoint. Additionally, we define the ”transcript” as the comprehensive record of all messages exchanged over the connection up until the point of analysis. The Handshake protocol itself can be divided into three distinct phases: the key exchange phase, the authentication phase, and the key derivation phase.

2.2.1 Key Exchange Phase

The protocol is started by the client sending a ClientHello to the server, containing the cryptographic algorithms supported by the client. The server replies with a ServerHello, containing the cryptographic algorithms selected from those offered by the client and a Diffie-Hellman public key. When the initial exchange is concluded, the two endpoints perform a Diffie-Hellman key exchange using the keys specified in the Hello messages.

2.2.2 Authentication phase

Starting from the server, the endpoints exchange their certificates (ServerCertificate) and a signature on the transcript using the key specified on them (ServerCertificateVerify). Finally, they send an HMAC on the transcript using the MAC keys obtained from the key derivation (ServerFinished and ClientFinished). The client uses the first key, the server uses the second one. The endpoints verify all the signatures and the MAC provided by the other endpoint. If any check fails, the connection is closed.

2.2.3 Keys Derivation phase

At the end of the Handshake, the endpoints feed the new messages of the transcript into the key derivation function. The key derivation function maintains an internal state, therefore all its outputs depend on the The key exchange phase. At the end, the parties obtain application keys(encrypted keys/MAC keys). From that moment, the Record layer protects the client-to-server flow through those keys. As it was proven in the authentication phase, all the values output by the Handshake are computationally independent, even slight modifications in the transcripts input in the key derivation function would lead to completely different and unpredictable outputs.

2.2.4 3P Handshake

To illustrate the 3P handshake, we first assign specific roles to the three parties involved: Prover (P) and Node (N) jointly act as the Client, while the DataSource (S) assumes the role of the Server.

This handshake process is based on the fundamental principle of Diffie-Hellman key exchange. Given the Client's public key, all participating parties should be capable of computing a secret-sharing of the exchanged secret. It is crucial to emphasize that if any party gains access to the exchanged secret, it could compute the symmetric keys and engage in unrestricted communication with the client. Therefore, in order to design a secure multiparty Diffie-Hellman protocol, it is essential for the parties to possess a secret-shared private key. It is worth noting that the public key does not require confidentiality. The protocol proceeds as follows:

The 3P-HS protocol between $\mathcal{P}$,$\mathcal{V}$ and $\mathcal{N}$

Define: EC be the Elliptic Curve used in ECDHE over $\mathbb{F}_p$, G a parameter, and Y_S the server public key. $\mathcal{P}$ and $\mathcal{N}$ output $k_{\mathcal{P}}^{MAC}$ respectively, while the TLS server outputs $k^{MAC} = k_{\mathcal{P}}^{MAC} + k_{\mathcal{V}}^{MAC}$. Besides, both $\mathcal{S}$ and $\mathcal{P}$ outputs k^{Enc}.

Process:

- $\mathcal{P}$ samples $r_c \leftarrow \$\{0,1\}^{256}$ and sends ClientHello(r_c) to $\mathcal{S}$.
- $\mathcal{S}$ sends ServerHello(r_s), ServerKeyEx($Y, \sigma, cert$) to $\mathcal{P}$.
- $\mathcal{P}$ verifies the certificate and check σ is a valid signature over(r_c, r_s, Y_S) signed by private key in certificate.
- $\mathcal{P}$ commit $(r_c, r_s, Y_S, \sigma, cert)$ to $\mathcal{N}$.
- $\mathcal{N}$ do the same check as what P do.
- $\mathcal{N}$ samples $s_N \leftarrow \mathbb{F}_p$ and computes $Y_N = s_N \cdot G$ then sends Y_N to $\mathcal{P}$.
- $\mathcal{P}$ samples $s_P \leftarrow \mathbb{F}_p$ and computes $Y_P = s_P \cdot G$ then sends $Y_P + Y_N$ to $\mathcal{S}$.

- $\mathcal{P}$ and $\mathcal{N}$ run the process to compute a sharing of the x-coordinate of $Y_P + Y_N$ (specified below), denoted x_P, x_N.

- $\mathcal{P}$ (and $\mathcal{N}$) send x_P (and x_N) to a process of key derivation (specified below) to compute shares of session keys and the master secret. $\mathcal{P}$ receives($k^{Enc}, k_{\mathcal{P}}^{MAC}, m_{\mathcal{P}}$), while $\mathcal{N}$ receives $(\mathcal{N}^{MAC}, m_{\mathcal{N}})$.

- $\mathcal{P}$ runs 2PC-PRF through GC with $\mathcal{N}$ to compute $s = PRF(m_{\mathcal{P}} \oplus m_{\mathcal{N}}, "clientfinished", h)$ and sends a Finished(s) to $\mathcal{S}$.

- $\mathcal{S}$ receives the client finished data and verifies its correctness then sends the server finished data.

- $\mathcal{P}$ and $\mathcal{N}$ run a 2PC to check $s \overset{?}{=} PRF(m_{\mathcal{P}} \oplus m_{\mathcal{N}}, "serverfinished", h)$ and abort if not.

- 3P-HS is done.

Compute a Sharing of the X-coordinate

Define: $P_1 = (x_1, y_1)$ from $\mathcal{P}$, $P_2 = (x_2, y_2)$ from $\mathcal{N}$. M_A, M_B, N_A, N_B, S_p is random mask. p is the prime number for the curve. $E(x)$ is a homomorphic encryption function which supports additive homomorphism. $\mathcal{P}$ and $\mathcal{N}$ outputs s_1 and s_2 such that $s_1 + s_2 = x$ where $(x, y) = P_1 + P_2$.

Process:

- $x = (y_2^2 - 2y_2y_1 + y_1^2)(x_2 - x_1)^{-2} - x_2 - x_1$. According to Fermat's little theorem, $x = (y_2^2 - 2y_2y_1 + y_1^2)(x_2 - x_1)^{p-3} - x_2 - x_1$.
- Make $A = (y_2^2 - 2y_2y_1 + y_1^2)$, $B = (x_2 - x_1)^{p-3}$, $C = -x_p - x_q$.
- Compute $A = (y_2^2 - 2y_2y_1 + y_1^2)$:
  - $\mathcal{N}$ sends $E(y_2^2)$, $E(-2y_2)$ and $E(x_2)$ to $\mathcal{P}$.
  - $\mathcal{P}$ computes $E(y_1^2)$ then $E(A) = E(y_2^2) + E(-2y_2) * y1 + E(y_1^2)$. $\mathcal{P}$ computes $E(A * M_A + N_A)$ then sends $E(A * M_A + N_A)$ and $(N_A \ mod \ p)$ to $\mathcal{N}$.
  - $\mathcal{N}$ decrypts and gets $(A * M_A + N_A)$. Then $\mathcal{N}$ reduces $(A * M_A + N_A) \ mod \ p$ and computes $A * M_A \ mod \ p = (A * M_A + N_A) \ mod \ p - N_A \ mod \ p$
- Similar algorithm could be applied for the computation of B. After the computation, $\mathcal{P}$ picks M_B and $\mathcal{N}$ gets $(B * M_B)$
- Compute x:
  - $\mathcal{N}$ sends $E(A * M_A * B * M_B)$ and $E(-x_2)$ to $\mathcal{P}$.
  - $\mathcal{P}$ computes $E(A*B+C) = E(A*B)+E(-x_2)+E(-x_1) = E(A*M_A*B*M_B)*(M_A*M_B)^{-1}+E(-x_2)+E(-x_1)$ and applies S_p then sends $E(A*B+C+s_p)$ to $\mathcal{N}$. $\mathcal{P}$ gets his share $s1 = (S_p \ mod \ p)$
  - $\mathcal{N}$ decrypts and gets $A * B + C + S_p$, then computes his share $s2 = (A * B + C + Sq) \ mod \ p$

Key Derivation

Define: nonce r_c, r_s, p_P from $\mathcal{P}$, p_N from $\mathcal{N}$;

Process:

- $p = p_P + p_N$
- $m = PRF(p, "mastersecret", r_c||r_s)$
- $k^{MAC}, k^{Enc} = PRF(m, "keyexpansion", r_s||r_c)$
- Sample $r_k, r_m \leftarrow \mathcal{F}_p$. Send$(k^{Enc}, r_k, r_m)$ to $\mathcal{P}$, and $(r_k \oplus k^{MAC}, r_m \oplus m)$ to $\mathcal{N}$ privately.

2.3 The Record Layer

The Record layer provides fragments as application data, which signifies the actual communication between the two endpoints. User-sensitive information is contained within this application data, usually in the form of HTTP requests/responses, and is encrypted/authenticated using the application keys. The security of this system is maintained if each party keeps their respective application keys private. This ensures the protection of user data throughout the communication process, minimizing the potential risks of data leakage and unauthorized access.

2.3.1 Multi-Record

TLS is performed on streams, and the data in those streams are put into one or more fragments (of max 2^{14} bytes). Client message boundaries are not preserved in the record layer (i.e., multiple client messages of the same ContentType MAY be coalesced into a single TLSPlaintext record, or a single message MAY be fragmented across several records). This means that we need to authenticate all records.

Define: R is the response from DataSource. $R = r_1||...||r_i||...||r_n$, where r_i is the record for $i \in \{1...n\}$.$r_i = header_i||data_i||mac_i$, where $header_i$ is the record header for i-th record, $data_i$ is the fragement data for i-th record, mac_i is the authentication mac for i-th record.

Verifier: compute $s_i = auth(r[i], mac[i])$, and check if $s_i = true$ for $i \in \{1...n\}$. If any check fails, output Abort.

2.3.2 Multi-Response

In specific usage scenarios, it may be necessary for the Prover to retrieve information from multiple responses originating from different APIs, potentially requiring multiple TLS sessions with a single data source. This implies that the Prover would need to perform multiple rounds of MPC and ZKP. To handle this situation efficiently and reduce overhead, we propose leveraging the same handshake session.

HTTP keep-alive is a mechanism designed to reuse the same connection for multiple HTTP requests and responses. This inherently reduces the number of TLS handshakes, as only one TLS handshake is performed per TCP connection. In other words, with HTTP keep-alive, a single TCP and TLS handshake can accommodate multiple HTTP requests. Consequently, we can transmit different application data pertaining to various requests within a single TLS session.

Furthermore, resuming the TLS session allows the Prover and the DataSource to utilize the same set of keys. The TLS protocol provides a method for session resumption known as Session Tickets, as defined in RFC 5077. The server can utilize a key rotation algorithm to encrypt and decrypt tickets using different keys over time. Additionally, the server can send new tickets to the client after each successful resumption. By implementing this standard, we can ensure an efficient and secure data retrieval process.

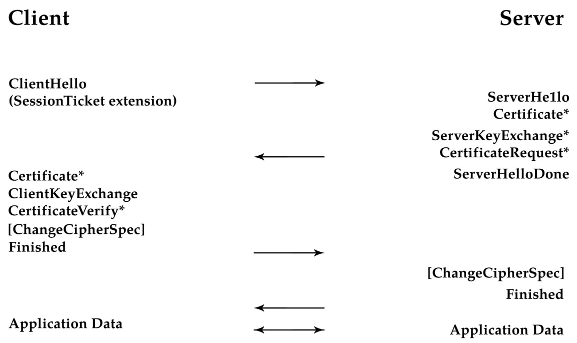

Figure 1: Resuming TLS Session with Session Ticket

2.4 Additional Explanation

The content we've presented here partially explains TLS. We still need to cover numerous additional procedures and features in this discussion. For a more comprehensive understanding and deeper insight into these topics, we highly recommend referring to RFC5246 and RFC8446, specific Internet standards that provide detailed information about the various aspects and applications of TLS.

3 MPC

3.1 OT

The OT protocol is a technique first introduced by Rabin in 1981. Coupled with Garbled Circuits (GC), it can facilitate secure Multi-Party Computation (MPC) protocols. Parties involved in secure computation employ OT to establish a secure communication channel or to exchange encrypted data. This allows them to construct the GC and execute the computation without exposing sensitive information. To illustrate our protocol, we'll concentrate on the chosen 1-out-of-2 OT, often called standard OT, though it can easily be extended to 1-out-of-N OT.

There are two roles for OT: sender and receiver. Specifically, the sender has two messages m_1 and m_2, the receiver has an index $c \in \{0, 1\ \}$ and the receiver wants to receive the sender's $c - th$ message without letting the sender know c. At the same time, the sender wants to ensure that the recipient can only receive one of these two messages.

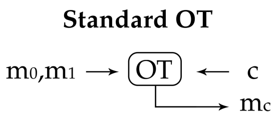

Figure 2: Standard OT

In practice, we often define $m_1 = m_0 \oplus \Delta$, such that $m_c = m_0 \oplus c \cdot \Delta$, this particular variation is known as correlated OT. The Following outlines the specifics of implementing basic OT using the Diffie-Hellman key exchange:

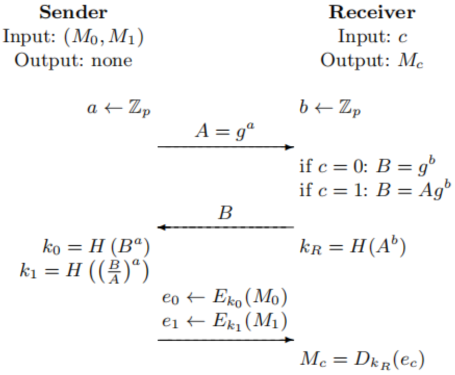
Sender
Input: (M_0, M_1)
Output: none
Receiver
Input: c
Output: M_c
$a \leftarrow \mathbb{Z}_p$
$b \leftarrow \mathbb{Z}_p$
$A = g^a$
if $c = 0$: $B = g^b$
if $c = 1$: $B = Ag^b$
B
$k_0 = H\left(B^a\right)$
$k_1 = H\left(\left(\frac{B}{A}\right)^a\right)$
$k_R = H(A^b)$
$e_0 \leftarrow E_{k_0}(M_0)$
$e_1 \leftarrow E_{k_1}(M_1)$
$M_c = D_{k_R}(e_c)$

Figure 3: The Simplest Protocol for Oblivious Transfer

Given that OT serves as a prerequisite for both Garbled Circuits (GC) and Interactive Zero-Knowledge Proofs (IZKP), and considering our need for large-scale utilization of OT, we require an efficient methodology for generating OT instances that are optimally cost-effective. Additionally, to ensure adequate security of the entire zkPass protocol, it's crucial that our OT protocol possesses active security.

In the following section, we detail the OT protocol we are utilizing, which requires merely 128 basic OT instances. This approach eliminates the need for costly public key encryption for basic one-time passwords and aids in distributing network and computational load effectively, thereby optimizing our protocol's overall efficiency and security.

Protocol for $ROT^{\kappa,\iota}$

The protocol uses an arbitrary stretch pseudorandom generator, PRG, and a hash function $H : \{0,1\}^* \to \{0,1\}^\kappa$, modeled as a random oracle.

Initialize:

- R samples κ pairs of random $\kappa - bit$ seeds,$\{(k_0^i, k_1^i)\}_{i=1}^{\kappa}$.
- S samples a random $\Delta = (\Delta_1, ..., \Delta_\kappa) \in \mathbb{F}_2^\kappa$.
- The parties call $\kappa \times OT_\kappa$ with inputs Δ and k_0, k_1.
- S receives $k_{\Delta_i}^i$ for $i = 1, ..., \kappa$.

Extend: R inputs the choice bits $x_1, ...x_\iota \in \mathbb{F}_2$. Let $\iota^{'} = \iota + s$, and assume that $s|\iota$.

- R picks random $x_{\iota+1}, ..., x_{\iota+s} \in \mathbb{F}_2$ and lets $x = (x_1, ..., x_{\iota^{'}})$.
- Expand k_i^0 and k_i^1, each using the next $\iota^{'}$ bits from PRG, obtaining $t_0^i = PRG(k_i^0) \in \mathbb{F}_2^{\iota^{'}}$ and $t_1^i = PRG(k_i^1) \in \mathbb{F}_2^{\iota^{'}}$, $i = 1, ..., \kappa$.
- R computes and sends $u^i = t_0^i + t_1^i + x \in \mathbb{F}_2^{\iota^{'}}$, for $i = 1, ..., \kappa$.
- S computes $q^i = \Delta_i \cdot u^i + t_{\Delta_i}^i \in \mathbb{F}_2^{\iota^{'}}$. Write $t^i = t_0^i$, so that $q^i = t^i + \Delta_i \cdot x$, for $i = 1, ..., \kappa$.

Consistency check: Let $m = \iota/s$. We divide the $\iota^{'}$ OTs into $m+1$ blocks of s bits, writing $x = (\hat{x}_1, ..., \hat{x}_{m+1}) \in \mathbb{F}_{2^S}^{m+1}$, and similary $t^i = (\hat{t}_1^i, ..., \hat{t}_{m+1}^i) \in \mathbb{F}_{2^S}^{m+1}$, $q^i = (\hat{q}_1^i, ..., \hat{q}_{m+1}^i) \in \mathbb{F}_{2^S}^{m+1}$, for $i = 1, ..., \kappa$.

- S samples and sends $(\mathcal{X}_1, ..., \mathcal{X}_m) \overset{\$}{\leftarrow} \mathbb{F}_{2^S}^m$.
- R computes the following values over $\mathbb{F}_{2^S}$ and sends them to S. $x = \sum_{j=1}^{m} \hat{x}_j \cdot \mathcal{X}_j + \hat{x}_{m+1}, t^i = \sum_{j=1}^{m} \hat{t}_j^i \cdot \mathcal{X}_j + \hat{t}_{m+1}^i$, for $i = 1, ..., \kappa$.
- S computes $q^i = \sum_{j=1}^{m} \hat{q}_j^i \cdot \mathcal{X}_j + \hat{q}_{m+1}^i \in \mathbb{F}_{2^S}$ and checks that $q^i = t^i + \Delta_i \cdot x$, for all $i = 1, ..., \kappa$. If any check fails, output Abort.

Transpose and randomize:

- Let q_j denote th $j - th$ row of the $\iota^{'} \times \kappa$ bit matrix $[q^1|...|q^\kappa]$ held by S, and similarly let t_j be the $j - th$ row of $[t_0^1|...|t_0^\kappa]$, held by R.
- R outputs $v_{x_j,j} = H(j\|t_j), j \in [\iota]$.
- S outputs $\mathbf{v}_{0,j} = H(j\|q_j)$ and $\mathbf{v}_{1,j} = H(j\|q_j + \Delta), j \in [\iota]$.

3.1.1 GC

GC is an encryption protocol that facilitates secure computation between two parties. It allows two participants, who may not trust each other, to jointly evaluate a function on their respective private inputs without necessitating a trusted third party. For the GC protocol, the function to be evaluated must be defined as a Boolean circuit.

The function underlying this, such as the comparison function in the millionaire problem, is presented as a Boolean circuit with two input gates. Both parties are aware of this circuit. What follows is an illustration of the GC workflow.

A component of a garbling scheme $G = (G_b, E_n, D_e, E_v, e_v)$. The function G_b maps f and k to (F, e, d), where strings encode the garbled function, encoding function and decoding function. With e and x one can compute the garbled input $X = E_n(e, x)$; with F and X one can compute the garbled output $Y = E_v(F, X)$; knowing d and Y allows for recovery of the final output $y = D_e(d, Y)$, which must be equal to $e_v(f, x)$.

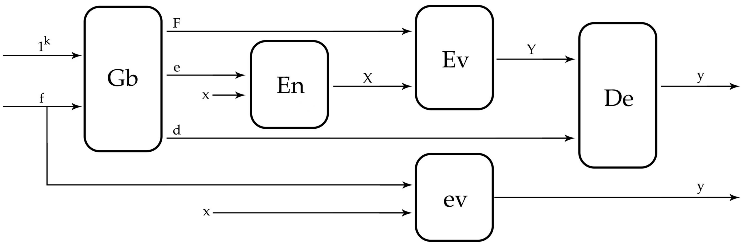

Figure 4: Foundations of Garbled Circuits

Considering that both the Prover and the Node maintain their respective private keys during the key exchange phase of the 3P handshake, both parties need to use their own private keys to compute the Pre-Master Secret (PMS). This PMS is then further derived into a session key. We employ Garbled Circuits (GC) for this calculation. Below, we detail the specific implementation of GC as used in zkPass.

```
procedure $Gb(1^k, f)$ :
    $R \twoheadleftarrow \{0,1\}^{k-1}$
    for $i \in$ Inputs($f$) do
        $W_i^0 \twoheadleftarrow \{0,1\}^k$
        $W_i^1 \leftarrow W_i^0 \oplus R$
        $e_i \leftarrow W_i^0$
    for $i \notin$ Inputs($f$)$\{in \ \ topo.\ order\}$ do
        $\{a, b\} \leftarrow$ GateInputs($f, i$)
        if $i \in$ XorGates($f$) then
            $W_i^0 \leftarrow W_a^0 \oplus W_b^0$
        else
            $(W_i^0, T_{G_i}, T_{E_i}) \leftarrow$ GbAnd($W_a^0, W_b^0$)
        $F_i \leftarrow (T_{G_i}, T_{E_i})$
        else if
        $W_i^1 \leftarrow W_i^0 \oplus R$
    for $i \in$ Onputs($f$) do
        $d_i \leftarrow lsb(W_i^0)$
    return $\{\hat{F}, \hat{e}, \hat{d}\}$

private procedure $GbAnd(W_a^0, W_b^0)$ :
    $p_a \leftarrow lsb\ W_a^0$
    $p_b \leftarrow lsb\ W_b^0$
    $j \leftarrow$ NextIndex()
    $j' \leftarrow$ NextIndex()
    $\{First \ \ half \ \ gate\}$
    $T_G \leftarrow H(W_a^0, j) \oplus H(W_a^1, j) \oplus p_b R$
    $T_G^0 \leftarrow H(W_a^0, j) \oplus p_a T_G$
    {Second half gate}
    $T_E \leftarrow H(W_b^0, j') \oplus H(W_b^1, j') \oplus W_a^0$
    $W_E^0 \leftarrow H(W_b^0, j') \oplus p_b(T_E \oplus W_a^0)$
    {Combine halves}
    $W^0 \leftarrow W_G^0 \oplus W_E^0$
    return $\{W^0, T_G, T_E\}$
```

```
procedure $En(\hat{e}, \hat{x})$ :
    for $e_i \in \hat{e}$ do
        $X_i \leftarrow e_i \oplus x_i R$
    return $\hat{X}$

procedure $Dn(\hat{d}, \hat{Y})$ :
    for $d_i \in \hat{d}$ do
        $y_i \leftarrow d_i \oplus lsb\ Y_i$
    return $\hat{y}$

procedure $Ev(\hat{F}, \hat{X})$ :
    for $i \in$ Inputs($\hat{F}$) do
        $W_i \leftarrow X_i$
    for $i \notin$ Inputs($\hat{F}$)$\{in \ \ topo.\ order\}$do
        $\{a, b\} \leftarrow$ GateInputs($\hat{F}, i$)
        if $i \in$ XorGates($\hat{F}$) then
            $W_i \leftarrow W_a \oplus W_b$
        else
            $s_a \leftarrow lsb\ W_a$
            $s_b \leftarrow lsb\ W_b$
            $j \leftarrow$ NextIndex()
            $j' \leftarrow$ NextIndex()
            $(T_G, T_{E_i}) \leftarrow F_i$
            $W_{G_i} \leftarrow H(W_a, j) \oplus s_a T_{G_i}$
            $W_{E_i} \leftarrow H(W_b, j') \oplus s_b(T_{E_i} \oplus W_a)$
            $W_i \leftarrow W_{G_i} \oplus W_{E_i}$
        end if
            $W_i^1 \leftarrow W_i^0 \oplus R$
    for $i \in Onputs(\hat{F})$ do
        $Y_i \leftarrow W_i$
    return $\hat{Y}$
```

4 IZK

In Non-interactive Zero-Knowledge (NIZK) Proof systems, such as zk-SNARK and zk-STARK, computations are represented as circuits, and the gate constraints within the circuit are depicted as a set of polynomials. If a computation requires multiple circuits, all these circuits need to be amalgamated into a single, large circuit that is then submitted. Despite the trust assumptions associated with this approach, it necessitates a very large memory space which is typically not feasible within browser environments.

To tackle the issue, we employ VOLE (Vector Oblivious Linear Evaluation)-based IZKP. Its linear nature allows us to submit circuits individually, effectively balancing memory size. Moreover, IZKP doesn't require a trusted setup, thereby enabling the generation of zero-knowledge proofs in a browser environment.

4.1 VOLE

VOLE is the arithmetic counterpart of string OT. Specifically, the VOLE functionality is a two-party functionality that takes a pair of vectors from the sender P_0, and allows the receiver P_1

to learn a chosen linear combination of these vectors. More formally, given a finite field $\mathbb{F}$, the VOLE functionality takes a pair of vectors $(u, v) \in \mathbb{F}_n \times \mathbb{F}_n$ from P_0 and a scalar $x \in \mathbb{F}$ from P_1. It outputs $w = ux + v$ to P_1. We will also consider a randomized version of VOLE where the sender's inputs (u, v) are picked randomly by the functionality and delivered as outputs to the sender. The deterministic VOLE functionality can be easily reduced to the randomized one, analogous to the reduction of OT to random OT.

Below is how we generating the VOLE instances:

VOLE Generator G_{primal}

- **Parameters:** dimension $k = k(\lambda)$, noise parameter $t = t(\lambda)$
- **Building blocks:** a code generator C, such that $C(k, n, \mathbb{F})$ defines a public matrix $C_{k,n} \in \mathbb{F}^{k \times n}$, and a multi-point function secret sharing $MPFSS = (MPFSS.Gen, MPFSS.Eval, MPFSS.fullEval)$
- G_{primal}.**Setup**$(1^\lambda, \mathbb{F}, n, x)$**:** pick a random size-t subset S of $[n]$, two random vectors $(a, b) \xleftarrow{R} \mathbb{F}^k \times \mathbb{F}^k$, and a random vector $y \xleftarrow{R} \mathbb{F}^t$. Let $s_1 < s_2 < ... < s_t$, denote the elements of S. Set $c \leftarrow ax + b$. Compute $(K_0, K_1) \xleftarrow{R} MPFSS.Gen(1^\lambda, f_{S,xy})$. Set $seed_0 \leftarrow (\mathbb{F}, n, K_0, S, y, a, b)$ and $seed_1 \leftarrow (\mathbb{F}, n, K_1, S, x, c)$. $Output(seed_0, seed_1)$.
- G_{primal}.**Expand**$(\sigma, seed_\sigma)$**:**
  - If $\sigma = 0$, parse $seed_0$ as $(\mathbb{F}, n, K_0, S, y, a, b)$. Set $\mu \leftarrow spread_n(S, y)$. Compute $v_0 \leftarrow MPFSS.FullEval(0, K_0)$. Output $(u, v) \leftarrow (a \cdot C_{k,n} + \mu, b \cdot C_{k,n} - v_0)$.
  - If $\sigma = 1$, parse $seed_1$ as $(\mathbb{F}, n, K_1, S, x, c)$. Compute $v_1 \leftarrow MPFSS.FullEval(1, K_1)$ and set $w \leftarrow (c \cdot C_{k,n} + v_1)$. Output w.

4.2 ICP-ZKP

We employ ICP-ZKP, a variant of interactive protocols that primarily uses symmetric key operations, thus achieving high efficiency and scalability. It can process millions of gates per second and manage large circuits consisting of billions of gates. The protocol includes a method for associating commitment schemes with proof commitment values. The commitment scheme actually serves as an Information-Theoretic Message Authentication Code (IT-MAC). This can be efficiently implemented using VOLE, and its homomorphic properties help to reduce the cost associated with addition gates.

The functionality of the ICP-IZK is described as follows:

Procotol $\Pi_{ZK}^{p,r}$

Inputs: The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ hold a circuit $\mathcal{C}$ over any field $\mathbb{F}_p$ with t multiplication gates. Prover $\mathcal{P}$ also holds a witness w such that $C(w) = 1$ and $|w| = n(i.e., |\mathcal{I}| = n)$.

Preprocessing phase: Both the circuit and witness are unknown.

- $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}_{ext-sVOLE}^{p,r}$, which returns a uniform $\Delta \in \mathbb{F}_{p^r}$ to $\mathcal{V}$.
- $\mathcal{P}$ and $\mathcal{V}$ send (extend,$n+t$) to $\mathcal{F}_{ext-sVOLE}^{p,r}$, which returns authenticated values $\{[\mu_i]\}_{i \in [n]}$ and $\{[v_i]\}_{i \in [t]}$ to the parties.
- $\mathcal{P}$ and $\mathcal{V}$ send (VOPE,1) to $\mathcal{F}_{ext-sVOLE}^{p,r}$, which returns uniform (A_0^*, A_1^*) to $\mathcal{P}$ and B^* to $\mathcal{V}$, such that $B^* = A_0^* + A_1^* \cdot \Delta$.

Online phase: Now the circuit and witness are known by the parties.

- For $i \in \mathcal{I}_in$, $\mathcal{P}$ sends $\delta_i := w_i - \mu_i \in \mathbb{F}_p$ to $\mathcal{V}$, and then both parties compute $[w_i] := [\mu_i] + \delta_i$.
- For each gate $(\alpha, \beta, \gamma, T) \in \mathcal{C}$, in a topological order:
  - If $T =$ Add, then two parites locally compute $[w_\gamma] := [w_\alpha] + [w_\beta]$.
  - If $T =$ Mult and this is the i-th multiplication gate, $\mathcal{P}$ sends $d_i := w_\alpha \cdot w_\beta - v_i \in \mathbb{F}_p$ to $\mathcal{V}$, and then both parties compute $[w_\gamma] := [v_i] + d_i$ (with $w_\gamma = w_\alpha \cdot w_\beta$ in the honest case).
- For the i-th multiplication gate, two parties hold an authenticated triple($[w_\alpha], [w_\beta], [w_\gamma]$)(with $k_i = m_i + w_i \cdot \Delta$ for $i \in \{\alpha, \beta, \gamma\}$) from the previous step and execute the following:
  - $\mathcal{P}$ computes $A_{0,i} := m_\alpha \cdot m_\beta \in \mathbb{F}_{p^r}$ and $A_{1,i} := w_\alpha \cdot m_\beta + w_\beta \cdot m_\alpha - m_\gamma \in \mathbb{F}_{p^r}$.
  - $\mathcal{V}$ computes $B_i := k_\alpha \cdot k_\beta - k_\gamma \cdot \Delta \in \mathbb{F}_{p^r}$.
- $\mathcal{P}$ and $\mathcal{V}$ perform the following check to verify that $B_i = A_{0,i} + A_{1,i} \cdot \Delta$ for all $i \in [t]$.
  - $\mathcal{V}$ samples $\mathcal{X} \leftarrow \mathbb{F}_{p^r}$ and sends it to $\mathcal{P}$.
  - $\mathcal{P}$ computes $U := \sum_{i \in [t]} B_i \cdot \mathcal{X}^i + A_0^*$ and $V := \sum_{i \in [t]} A_{1,i} \cdot \mathcal{X}^i + A_i^*$, and sends (U, V) to $\mathcal{V}$.
  - Then $\mathcal{V}$ computes $W := \sum_{i \in [t]} B_i \cdot \mathcal{X}^i + B^*$ and checks that $W = U + V \cdot \Delta$. If the check fails, $\mathcal{V}$ outputs false and aborts.
- For the single output wire h in the circuit $\mathcal{C}$, both parties hold $[w_h]$ with $k_h = m_h + w_h \cdot \Delta$, and check that $w_h = 1$ as follows:
  - In parallel with the previous step, $\mathcal{P}$ sends m_h to $\mathcal{V}$.
  - $\mathcal{V}$ checks that $k_h = m_h + \Delta$. If the check fails, then $\mathcal{V}$ outputs false. Otherwise, $\mathcal{V}$ outputs true.

4.3 Circuit Factory

4.3.1 Bristol Fashion Circuit

We utilize Bristol-style circuits, which are composed of AND, XOR, and INV gates. The format is defined as follows:

Bristol Format

- A line defining the number of gates and then the number of wires in the circuit.
- Two numbers defining the number n1 and n2 of wires in the inputs to the function given by the circuit. If the function has only one input then the second inputs size is set to zero.
- On the same line comes the number of wires in the output n3.
- The wires are ordered so that the first n1 wires correspond to the first input value, the next n2 wires correspond to the second input value. The last n3 wires correspond to the output of the circuit.
- Gates are listed in the format:
  - Number input wires
  - Number output wires
  - List of input wires

- List of output wires
- Gate operation (XOR, AND or INV)

4.3.2 Modular Circuit

When it comes to Modular Circuit design, the size of the response from the DataSource isn't always certain and can be approximately 1k. This implies that the scale of the whole circuit for authentication and assertion can be quite large, containing millions of gates. For a browser-like system, this is impractical to load the circuit and provide proof due to the significant memory consumption.

However, we can notice that the main circuit is not just a combination of many sub-circuits, but certain sub-circuits are reused repeatedly during the evaluation. Therefore, we can construct a proof system in a modular manner by connecting small specialized "gadget" circuits in a lightweight fashion. Thanks to the schema of ICP-ZK, we can recycle the memory: once a certain modular circuit is no longer needed in the calculation, all parties can remove the promise of that circuit from the memory.

From a theoretical perspective, modular circuit designs are flexible and reusable. If a computation naturally presents different "components", a general-purpose scheme will homogenize them into a single representation, which might lead to performance overhead. Thanks to the schema of ICP-ZK, we can recycle the memory: once a particular modular circuit is no longer needed in the calculation, all parties can remove the promise of that circuit from memory.

With a CP scheme one can prove statements of the form "$commit(x)$ contains x such that $R(x, w)$" where $commit(x)$ is a commitment. To see how the CP capability can be used for modular composition consider the following example of sequential composition in which one wants to prove that $\exists w, z = h(x, w)$, where $h(x, w) = g(f(x, w), w)$. Such a proof can be built by combining two CP systems Πf and Πg for its two building blocks, i.e., respectively f and g: the prover creates a commitment $commit(y)$ of y, and then uses $\Pi f(resp, \Pi g)$ to prove that "$commit(y)$ contains $y = f(x, w)$ ($resp$ contains y such that $z = g(y, w)$".

Here is what we have done for the sub circuit generation:

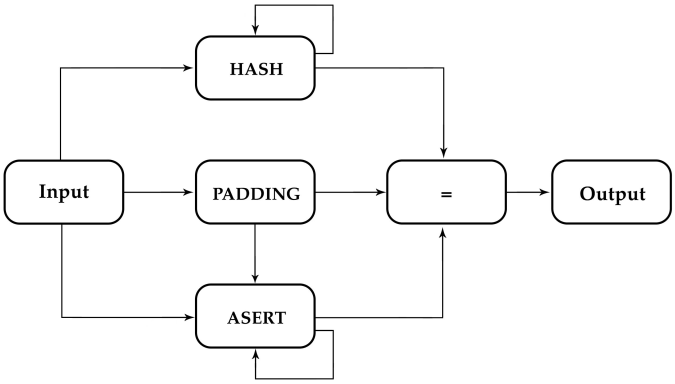

Figure 5: Sub Circuit Generation

4.3.3 Authentication of Response

The authentication is to check whether the corresponding HMAC is correct. HMAC is defined as follows:

$HMAC(K, m) = H((K^{'} \oplus opad) \| H((K^{'} \oplus ipad) \| m))$

$$K' = \begin{cases} \text{H(K)} & \text{if K is larger than block size} \\ \text{K} & \text{otherwise} \end{cases}$$

where

- H is a cryptography hash function.
- m is the message to be authenticated.
- K is the secret key.
- $K^{'}$ is a block-sized key derived from the secred key K; ether by padding to the right with 0s up to the block size, or by hashing down to less than or equal to the block size first and then padding to the right with zeros.
- $\|$ is denotes concatenation.
- $\oplus$ denotes bit wise XOR.
- *opad* is the block-sized outer padding, consisting of repeated bytes valued 0x5c.
- *ipad* is the block-sized inner padding, consisting of repeated bytes valued 0x36.

From a definitional standpoint, an HMAC proof typically requires two hash operations. However, in the context of IZKP (Interactive Zero-Knowledge Proofs), the computational cost of performing these hashes, typically using SHA256, is high due to the uncertainty of the response length. Therefore, it becomes necessary to introduce certain optimizations to mitigate this issue.

Our optimization exploits the Merkle–Damgård structure. Suppose b_1 and b_2 are two correctly sized blocks. Then $H(b_1 \| b_2)$ is computed as $f_H(f_H(S_i, b_1), b_2)$ where f_h denotes the one-way compression function of H, and S_i is the initial state for each round of compression. Based on the above analysis, we divide the HMAC into 4 steps:

1. calculate outer hash state: $Outer_{hs} = f_H(S_i, k' \oplus opad)$
2. calculate inner hash state: $Inner_{hs} = f_H(S_i, k' \oplus ipad)$
3. calculate inner hash: $Inner_h = f_H(Inner_{hs}, R)$
4. calculate HMAC: $HMAC = f_H(Outer_{hs}, Inner_h)$

Revealing the outer hash state or inner hash state would not reveal $k^{'}$ since the f_H is assumed to be one way. Thus during the 3P handshake, node can reveal the $Inner_{hs}$ to Prover, Prover can use $Inner_{hs}$ as public signal and its R as the witness to generate proof that: $\Pi = ZK-PoK\{R, Inner_{hs} : Inner_h = f_H(Inner_{hs}, R)\}$. $Inner_h$ also needs to be committed. Then the node commits the $Outer_{hs}$, so the check of $HMAC$ can be out of the ZK circuits which can significantly reduce cost.

Now we will reduce the number of hash operations in HMAC to one. We can outline the logic for the circuit as follows:

Define: R is the response, which is padding outside the circuit. $R = b_1||...||b_i||...||b_n$, b_i is the block of R, for $i \in 1...n$. iState is the init state of hash, fState is the final result of hash.

```
Procedure C($R, iState, fState$):
    $state = iState$
    for $i \in 1...n$ do:
```

```
    $b_i$ ← **GetBlock**(R)
    state← **hash**(state, b$_i$)
if $state = fState$ then
    return 1
return 0
```

4.3.4 Parse Response

In IZK, parsing JSON poses a significant computational burden. The syntax tree of each node contains numerous potential branches, and the parser must traverse each branch. Consequently, as the length of the JSON string increases, the total number of branches grows exponentially. This exponential growth makes ZKP circuits designed for parsing JSON documents impractical due to their sheer size.

However, in practical scenarios, JSON structures typically do not contain sensitive information. The response from the data source often exhibits a similar structure across different users, with user-specific sensitive data typically represented as scalar JSON values. Consequently, we can alleviate this issue by performing JSON parsing outside of the circuit.

Define: R is the response from DataSource and is the private input. R' is the mask of R which give all scalar to a garble value. q is the query for some key in JSON, for example, jq. GC is grammar checker function. VC is valid checker function. CE is a equal circuit.

Procedure F(R, q):

- $R' \leftarrow Trans(R)$
- $(Ind_1, ..., Ind_i, ...Ind_n) = GC(R', q)$, where $i \in \{1, deepPath(R)\}$. n is the height for the last leaf in grammar tree. Ind_i is range dual for node start index S_i and end index E_i in bit in R
- $s_1 = VC(ind_i)$, for $\forall i, j, ind[i]$∈ ind[j] & ind[i] ≠ ind[j] & ind[i] ind[j]
- $s_2 = CE(R, ind_i, pit)$, pit is the grammar path define in template
- check s_1 and s_2 are both true. If the check fails, abort.

4.3.5 Assert Response

Once we have got the ind_n, the corresponding value can be found at the index $E_n + 1$, so we could extract the value by the wire from $E_n + 1$ to $E_n + Len(value)$, where Len is the bit length function for value. We can verify the assertion by checking if $Assert(R, E_n + 1, E_n + Len(value)) = 1$.

To facilitate proof of assertions, we utilize a primitive query circuit called Assert. It is designed to easily provide proof of various assertions. Since each case may have different assertions, constructing a large combination circuit with specific query selectors is not feasible. Such an approach would make the main circuit excessively large and difficult to maintain. To address this, our circuit factory offers a range of common circuits. Enterprise users can choose and combine these circuits to fulfill their specific requirements. Currently, we provide support for the following assertion circuits:

- equal or not equal
- greater/less than or equal
- include/all different
- average
- sum
- arithmetic with scalar

- order
- regexp/like
- logical operation
- between or out of range

5 Benchmark

We conducted a performance evaluation of the complete protocol implementation using specific hardware setups. The Prover was executed on a MacBook Pro 15-inch Mid 2015, equipped with 16GB of 1600MHz DDR3 memory and a 2.5GHz Intel Core i7 processor. This configuration closely resembles the actual user environment. On the other hand, the Verifier was deployed on an AWS c6a.2xlarge instance, which offers 8 virtual CPUs and 16GB of memory (GiB).

By utilizing these hardware setups, we aimed to assess the protocol's performance under conditions that closely resemble real-world scenarios.

ZK	**Block**	**Setup Time**	**Prove Time**	**Verify Time**	**Memory**	**Gates**
SNARK	4	0s	22000ms	100ms	630M	540,292
ZKPASS	4	1.2s	210ms	60ms	80M	540,292
ZKPASS	10	2.2s	340ms	100ms	130M	1,350,730
ZKPASS	20	3.4s	580ms	160ms	180M	2,701,460

6 Future Direction

The fields of MPC and ZK related technologies are constantly evolving, with significant advancements being made each year. To ensure the continuous optimization of the zkPass protocol, we are actively staying abreast of the latest technologies. Here are some noteworthy references that we are considering: [FNO14], [NNOB11], [HK20], [HYDK22], [ADST21], [BDOZ10], [FKL$^+$21], [DIO20], [JKO13], [GAZ$^+$21].

OT. Traditional OT protocols like [IKNP03] and [KOS15] require security parameter λ bits of communication per OT instance. However, we propose building a new maliciously secure OT extension based on VOLE([BCGI19]/[SGRR19]) which only needs λ/k bits. for any k, at the expense of requiring $2^{k-1}/k$ times the computation.

GC. Existing GC evaluation methods such as [ZRE14] and [KS08] evaluate GC functions gate-by-gate using encrypted truth tables. The GC evaluator decrypts the corresponding output label based on input labels. However, interactive protocols offer more sophisticated techniques. For example, we can expose a (masked) private value to a party, allowing them to perform local computation and feed the resulting cleartext value back into the MPC.

IZK. VOLE-based IZK([WYKW20]/[YSWW21]) suffers from high communication overhead, often linear to the circuit size. We are constructing a new ZK protocols with communication sublinear to the circuit size, while maintaining a similar level of computational efficiency.

References

[ADST21] Damiano Abram, Ivan Damgård, Peter Scholl, and Sven Trieflinger. Oblivious tls via multi-party computation. Cryptology ePrint Archive, Paper 2021/318, 2021. https://eprint.iacr.org/2021/318.

[BCGI19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector ole. Cryptology ePrint Archive, Paper 2019/273, 2019. https://eprint.iacr.org/2019/273.

[BDOZ10] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. Cryptology ePrint Archive, Paper 2010/514, 2010. https://eprint.iacr.org/2010/514.

[DIO20] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. Cryptology ePrint Archive, Paper 2020/1446, 2020. https://eprint.iacr.org/2020/1446.

[FKL$^+$21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for ram programs. Cryptology ePrint Archive, Paper 2021/979, 2021. https://eprint.iacr.org/2021/979.

[FNO14] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. Cryptology ePrint Archive, Paper 2014/598, 2014. https://eprint.iacr.org/2014/598.

[GAZ$^+$21] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. Cryptology ePrint Archive, Paper 2021/1022, 2021. https://eprint.iacr.org/2021/1022.

[HK20] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. Cryptology ePrint Archive, Paper 2020/136, 2020. https://eprint.iacr.org/2020/136.

[HYDK22] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast zk processor with cached ram for ansi c programs. Cryptology ePrint Archive, Paper 2022/810, 2022. https://eprint.iacr.org/2022/810.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 145–161, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: How to prove non-algebraic statements efficiently. Cryptology ePrint Archive, Paper 2013/073, 2013. https://eprint.iacr.org/2013/073.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure ot extension with optimal overhead. Cryptology ePrint Archive, Paper 2015/546, 2015. https://eprint.iacr.org/2015/546.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*, ICALP '08, page 486–498, Berlin, Heidelberg, 2008. Springer-Verlag.

[NNOB11] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. Cryptology ePrint Archive, Paper 2011/091, 2011. https://eprint.iacr.org/2011/091.

[SGRR19] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-ole: Improved constructions and implementation. Cryptology ePrint Archive, Paper 2019/1084, 2019. https://eprint.iacr.org/2019/1084.

[WYKW20] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. Cryptology ePrint Archive, Paper 2020/925, 2020. https://eprint.iacr.org/2020/925.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. Cryptology ePrint Archive, Paper 2021/076, 2021. https://eprint.iacr.org/2021/076.

[ZRE14] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. Cryptology ePrint Archive, Paper 2014/756, 2014. https://eprint.iacr.org/2014/756.