SD-in-the-Head rust implementation and optimization

Hugh Benjamin Zachariae, 201508592
Magnus Jensen, 201708626

Master's Thesis, Computer Science
November 2024
Advisor: Diego F. Aranha

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

►in English. . . ◄

Resumé

▶in Danish. . . ◀

Acknowledgments

►. . . ◄

Magnus Jensen,
Aarhus, November 2024.

Contents

Chapter 1

Introduction

few pages. INtroduce what we have done and how the paper is structured

►motivate and explain the problem to be addressed◄

►example of a citation: [?]◄ ►get your bibtex entries from `https://dblp.org/`◄

Chapter 2

Description of the algorithm

Abstract level description of the algorithm own words MPCitH Syndrome detection problem Fiat-Shamir heuristic

2.1 Syndrome Decoding Problem

Describe the syndrome decoding problem introducing why we need ZKAoK, LSSS, Beaver triples and MPC construction from [1] section 3.3.

The syndrome decoding problem is used in many code based cryptosystems. A syndrome is the result of multiplying a vector x with a parity-check matrix H, finding the original x based only on the syndrome is then without the parity-check matrix is NP-complete.
A good description of $p \neq Np$ and the "*coset weight*" problem and the reduction of this problem into a NP-complete problem [2]

2.2 Galois Finite Field

A group for bytes

►Describe the arithmetic operations in GF256 and the extension field◄

2.2.1 Extension field

F_q^η

Polynomial evaluation

For polynomial evalution in the multiparty computation seen in **??**, we need to evaluate a polynomial $P(x)$ at a point $r \in F_q^\eta$. This is done by evaluating the polynomial at each coefficient and summing the results. The polynomial $P(x)$ is defined as

$$\begin{aligned} &\bigcup_{|Q|}(F_q)^{|Q|} \times F_q^\eta \to F_q^\eta \\ &\qquad Q(r) = \sum_{i=0}^{|Q|} Q_i \cdot r^{i-1} \qquad\qquad Q_i \in F_q, r \in F_q^\eta \end{aligned} \tag{2.1}$$

2.3 Linear Secret Sharing Scheme (LSSS)

Secret sharing schemes (SSS) are a type of cryptographic protocol that allows for the distribution of a secret amongst a group of participants. The secret can only be reconstructed when a sufficient number of shares are combined together. The threshold variant of the SDitH protocol relies on a low-threshold linear secret sharing scheme (*LSSS*). Threshold secret sharing schemes allow for the reconstruction of a secret from a subset of shares of length ℓ, where ℓ is the threshold. The threshold allows for the SDitH protocol to be more communication efficient, as the amount of shares needed to reconstruct the secret is low.

Definition 2.3.1. *S is a* (ℓ,n) *threshold SSS if it satisfies the following properties*:

- **Share generation**: Given a secret s, the scheme generates n shares $\texttt{share}(s) = [[s]] = [[s_1, s_2, \ldots, s_n]]$.
- **Reconstruction**: Given a subset of shares $[[s']]$ of size ℓ, the scheme can reconstruct the secret $s = \texttt{open}([[s']])$.

Linear secret charing schemes, or $(+,+)$-homomorphic schemes, refers to secret sharing schemes that are linearly homomorphic over some field $\mathbb{F}$ (say the galois field `GF256` described in section 2.2). This means that given shares $[[a]]$ and $[[b]]$ we have that

Definition 2.3.2. A (ℓ,n) threshold SSS is $(+,+)$-homomorphic if for any two secrets s_1 and s_2 and their shares $[[s_1]]$ and $[[s_2]]$, the sum of the shares $[[s_1]] + [[s_2]] = [[s_{11} + s_{21}, s_{12} + s_{22}, \ldots, s_{1n} + s_{2n}]]$ is equal to the share of the sum of the secrets $[[s_1 + s_2]]$ for the same subset of shares.

2.3.1 Shamir's Secret Sharing

Shamir's Secret Sharing scheme [4]. Shamir's Secret Sharing is a method for distributing a secret amongst a group of participants, each of which is given a share of the secret. The secret can only be reconstructed when a sufficient number of shares are combined together.

For a secret s and a given security threshold ℓ, the share $[[s]]$ is generated by sampling a random polynomial of degree $t-1$ with s as the free coefficient. Each participant is given a share of the polynomial evaluated at a point. The secret can be reconstructed by interpolating the polynomial from a sufficient number of shares.

The (t,n) Shamir's polynomial based secret sharing scheme is $(+,+)$-homomorphic in which the addition of two polynomials secrets equals the Lagrange's interpolation of the sum-of-shares for the same subset of shares.

►Write about creating shares, addition◄

2.3.2 Multiplication and Beaver Triples

2.4 Zero-Knowledge Proofs

Zero-Knowledge Arguments of Knowledge (ZKAoK) allow a prover to convince a verifier that they knows a certain witness for a statement without revealing any additional information about the witness. **►What are they used for..◄**

ZKAoK protocols have been introduced in multiple variants that each solve specific relational problems with varying levels of communication complexity and prover running times. While many of these perform well for these specific problems, there has long been a need for a well performing generic ZKAoK protocol.

2.4.1 MPC-in-the-Head

Secure Multiparty Computation or MPC describes a cryptographic protocol that allows multiple parties to run a circuit C, while no information on inputs is leaked outside of the intended output of C. These algorithms can be augmented to ensure correctness, even in the presence of malicious parties. **►skriv mere her. Check threshold paper mÃ¥ske?◄**

The MPC-in-the-Head (MPCitH) framework, introduced by [3], uses MPC protocols to construct generic ZKAoK protocols. First, the statement is turned into a circuit C. On input x, C outputs y iff. $x = w$ for some witness w which is a correct witness for the statement to be proven. The prover simulates the MPC protocol of all the parties *in the head*. The parties each evalutate C on a secret sharing of w

The framework benefits from previous efficiency improvements in MPC protocols to improve the efficiency of ZKAoK protocols. Particularly, its possible to achieve *constant-rate* zero knowledge. For an arbitrary circuit C of size s and a bounded fan-in, it is possible to construct a ZKAoK protocol with communication complexity $O(s) + \text{poly}(k)$, for a security parameter k. This is an improvement over previous ZKAoK protocols, which had communication complexity $O(ks)$. **►write about this in the previous section◄**.

2.4.2 MPC preprocessing

We first generalize the idea of [KKW18] to work over arithmetic circuits using a variant of the SPDZ MPC protocol [DPSZ12,LN17] and provide a formal proof of security to their "cut-and-choose" preprocessing heuristic. Then, we present a new construction where we replace the "cut-and-choose" mechanism with a "sacrificing"-based approach. While both approaches have similar cost per MPC instance, our "sacrificing"-based approach yields a smaller cheating probability, which means that the number of MPC instances simulated in the proof can be significantly smaller, thus reducing the overall communication footprint. Our scheme is highly flexible in its choice of parameters. In particular, by changing the number of parties in the underlying MPC protocol, one can alternate between achieving low communication and low running time. Our construction only requires efficient standard symmetric primitives, and thus is plausibly post-quantum secure even in the non-interactive case [DFMS19]. The two constructions can be found in [1] **►Copied from citation. rewrite this◄**

2.4.3 Verification of a multiplication triple using another

Given a random triple $([[a]], [[b]], [[c]])$, it is possible to verify the correctness of a triple $([[x]], [[y]], [[z]])$, i.e., that $z = x \cdot y$, without revealing any information on either of the triples, in the following way [1].

1. The parties generate a random $\varepsilon \in \mathbb{F}$.
2. The parties locally set $[[\alpha]] = \varepsilon[[x]] + [[a]], [[\beta]] = [[y]] + [[b]]$.
3. The parties run $\texttt{open}([[\alpha]])$ and $\texttt{open}([[\beta]])$ to obtain α and β.
4. The parties locally set $[[v]] = \varepsilon[[z]] - [[c]] + \alpha \cdot [[b]] + \beta \cdot [[a]] - \alpha \cdot \beta$.
5. The parties run $\texttt{open}([[v]])$ to obtain v and accept iff $v = 0$.

Observe that if both triples are correct multiplication triples (i.e., $z = xy$ and $c = ab$) then the parties will always accept since

$$\begin{aligned}
v &= \varepsilon \cdot z - c + \alpha \cdot b + \beta \cdot a - \alpha \cdot \beta && (2.2)\\
&= \varepsilon \cdot xy - ab + (\varepsilon \cdot x + a)b + (y+b)a - (\varepsilon \cdot x + a)(y+b) && (2.3)\\
&= \varepsilon \cdot xy - ab + \varepsilon \cdot xb + ab + ya + ba - \varepsilon \cdot xy - \varepsilon \cdot xb - ay - ab && (2.4)\\
&= (\varepsilon \cdot xy - \varepsilon \cdot xy) + (ab - ab) + (\varepsilon \cdot xb - \varepsilon \cdot xb) + (ya - ay) + (ba - ab) && (2.5)\\
&= 0 && (2.6)
\end{aligned}$$

2.4.4 Error probability

Lemma 2.4.1. *If* $([[a]], [[b]], [[c]])$ *or* $([[x]], [[y]], [[z]])$ *is an incorrect multiplication triple then the parties output* `accept` *in the sub-protocol above with probability* $\frac{1}{|\mathbb{F}|}$.

►Proof of lemma and solution◄

2.5 Fiat-Shamir Heuristic

►Turning the ZKAoK into a signature scheme◄

Chapter 3

Specification

more detailed description of the algorithm e.g. how we sampled I[e] witness challenge table of spec params (with our code naming and different categories)

3.1 Merkle Tree Commitment

3.2 Hashing and XOF

►Write about Shake, Keccak◄ and how we initiate the hash function and XOF

3.3 MPC computation

The computation is based on HVZKAoK Protocol using imperfect preprocessing and sacrificing. Section 3.3 [1].

A toy example of the computation protocol computations, can be seen in Figure 3.1. We have the two computation methods. Here for 1 split and 1 evaluation point. This means that we have only value for each challenge and beaver triple. Note that any arithmetic is run in GF256, so addition and subtraction are both `XOR` and multiplication is modulus $x^8 + x^4 + x^3 + x + 1$. Furthermore, for negation we have that $-a = a$.

Note that the

If we first instantiate an input i and one random input i^* (like the `input_coef`).Then the input share is generated by adding the two. Similar, but simpler, to the input share generation of Algorithm 12, line 13 of the specification.

$$\begin{aligned}
i &= (s_a, Q, P, a, b, c) \\
i^* &= ({s_a}^*, Q^*, P^*, a^*, b^*, c^*) \\
[i] = i + i^* &= (s_a + {s_a}^*, Q + Q^*, P + P^*, a + a^*, b + b^*, c + c^*) \\
&= ([s_a], [Q], [P], [a], [b], [c]) \\
\mathtt{chal} &= (\varepsilon, r) \\
\mathtt{pk} &= (H', y)
\end{aligned}$$

```
PartyComputation

    Input:
    $(s_a, Q', P, a, b, c), (\overline{\alpha}, \overline{\beta}), (H', y)$
    $(\varepsilon, r)$, with_offset
    Output:
    $(\alpha, \beta, v)$

    $Q = Q'_1$ if with_offset else $Q'_0$
    $S = (s_a | y + H's_a)$ if with_offset else $(s_a | H's_a)$
    $v = -c$
    $\alpha = \varepsilon \cdot Q(r) + a$
    $\beta = S(r) + b$
    $v \mathrel{+}= \varepsilon \cdot F(r) \cdot P(r)$
    $v \mathrel{+}= \overline{\alpha} \cdot b + \overline{\beta} \cdot a$
    $v \mathrel{+}= -\alpha \cdot \beta$    if with_offset
```

```
InverseComputation

    Input:
    $(s_a, Q', P), (\alpha, \beta, v), (\overline{\alpha}, \overline{\beta}), (H', y)$
    $(\varepsilon, r)$, with_offset
    Output:
    $(a, b, c)$

    $Q = Q_1$ if with_offset else $Q_0$
    $S = (s_a | y + H's_a)$ if with_offset else $(s_a | H's_a)$
    $c = -v$
    $a = \alpha - \varepsilon \cdot Q(r)$
    $b = \beta - S(r)$
    $c \mathrel{+}= \varepsilon \cdot F(r) \cdot P(r)$
    $c \mathrel{+}= \overline{\alpha} \cdot b + \overline{\beta} \cdot a$
    $c \mathrel{+}= -\alpha \cdot \beta$    if with_offset
```

Figure 3.1: Simplified version of the MPC party computation and inverse computation. Q_0 means that Q is completed with a 0 for leading coefficient. Furthermore, F is precomputed. Note that all arithmetic is done in $\mathbb{F}_q = GF256$. All elements are in $\mathbb{F}_q^\eta$ except for the coefficients of Q, S and P which are in $\mathbb{F}_q$.

We also compute the plain broadcast share of the input as per Algorithm 12, line 18. Note that $\overline{v}$ is computed to zero and therefore removed from the computation in the implementation.

$$(\overline{\alpha},\ \overline{\beta}) = \texttt{PartyComputation}(i,\ (\overline{\alpha},\overline{\beta}),\ \texttt{chal},\ \texttt{pk},\ \texttt{true})$$

$$\begin{aligned}
\overline{\alpha} =& \varepsilon \cdot Q_1(r) + a \\
\overline{\beta} =& S_y(r) + b \\
\overline{v} =& -c + \varepsilon \cdot F(r) \cdot P(r) + \overline{\alpha} \cdot b + \overline{\beta} \cdot a - \overline{\alpha} \cdot \overline{\beta} \\
\overline{v} =& -c + \varepsilon \cdot F(r) \cdot P(r) + (\varepsilon \cdot Q_1(r) + a) \cdot b + (S_y(r) + b) \cdot a - (\varepsilon \cdot Q_1(r) + a) \cdot (S_y(r) + b) \\
\overline{v} =& -c + \varepsilon \cdot F(r) \cdot P(r) \\
& + \varepsilon \cdot Q_1(r) \cdot b + c + S_y(r) \cdot a + c \\
& - \varepsilon \cdot Q_1(r) \cdot S_y(r) - \varepsilon \cdot Q_1(r) \cdot b - a \cdot S_y(r) - c \\
\overline{v} =& 0
\end{aligned}$$

We then compute a broadcast share from the randomness and the broadcast, as per Algorithm 12, line 21.

$$(\alpha^*, \beta^*, v^*) = \texttt{PartyComputation}(i^*, (\overline{\alpha}, \overline{\beta}), \texttt{chal}, \texttt{pk}, \texttt{false})$$

$$\begin{aligned}
\alpha^* &= \varepsilon \cdot {Q^*}_0(r) + a^* \\
\beta^* &= {S^*}_0(r) + b^*
\end{aligned}$$

This broadcast share is sent to the verifier along with the truncated input share (removing the beaver triples). The verifier then needs to recompute the input share beaver triples using the `InverseComputation` function. First we add the input share to the broadcast share as per Algorithm 13, line 8.

$$(\alpha', \beta', v') = (\alpha^*, \beta^*, v^*) + (\overline{\alpha},\ \overline{\beta}, 0) = (\alpha^* + \overline{\alpha}, \beta^* + \overline{\beta}, v^* + 0)$$

$$\begin{aligned}
\alpha' &= \varepsilon \cdot {Q^*}_0(r) + a^* + \overline{\alpha} \\
&= \varepsilon \cdot {Q^*}_0(r) + a^* + \varepsilon \cdot Q_1(r) + a \\
\beta' &= {S^*}_0(r) + b^* + \overline{\beta} \\
&= {S^*}_0(r) + b^* + S_y(r) + b
\end{aligned}$$

Next, the verifier computes the inverse of the broadcast share to recompute $([a], [b], [c])$

using the `InverseComputation` function. This is done as per Algorithm 13, line 10.

$$(a',b',c') = \texttt{InverseComputation}([i],(\alpha',\beta',v'),(\overline{\alpha},\overline{\beta}),\texttt{chal},\texttt{pk},\texttt{true})$$

$$\begin{aligned}
a' &= \alpha' - \varepsilon\cdot[Q]_1(r) \\
&= \varepsilon\cdot Q^*{}_0(r) + a^* + \varepsilon\cdot Q_1(r) + a - \varepsilon\cdot[Q]_1(r) \\
&= \varepsilon\cdot Q^*{}_0(r) - \varepsilon\cdot[Q]_1(r) + \varepsilon\cdot Q_1(r) + [a] \\
&= \varepsilon\cdot(Q^*{}_0(r) - [Q]_1(r) + Q_1(r)) + [a] \\
&= \varepsilon\cdot(Q_0^*(r) + Q_0^*(r)) + [a] && \textit{Equation } 2.1 \\
&= [a] && \textit{Equation } 2.1 \\
b' &= \beta' - [S]_y(r) \\
&= S^*{}_0(r) + b^* + S_y(r) + b - [S]_y(r) \\
&= S^*{}_0(r) - [S]_y(r) + S_y(r) + [b] \\
&= S^*{}_0(r) - S_0^*(r) + [b] && \textit{Equation } 2.1 \\
&= [b] && \textit{Equation } 2.1
\end{aligned}$$

►**Want to explain the above in a more detailed manner? Specifically** $Q^*{}_0(r) - [Q]_1(r) = Q_1(r)$◄

Chapter 4

Implementation

Tooling and language feaures (rust, criterion) code sections code re-usability with traits for categories.

const generics vs inline mutability (benchmarking?, nightly?) traits for categories

Chapter 5

Benchmarks

diaries of benchmarks. discussion of results

Chapter 6

Conclusion

wrap up and pose future work what should people continue with point to round 2 NIST work in context of timeline

►conclude on the problem statement from the introduction◄

Bibliography

[1] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In *IACR International Conference on Public-Key Cryptography*, pages 495–526. Springer, 2020.

[2] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[3] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 21–30, 2007.

[4] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

Appendix A

The Technical Details

►...◄