zkSalary: A Privacy-Preserving System for Salary Verification Using zk-SNARK

Pham Duc Hao[a]

[a] *Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), Ho Chi Minh City, Vietnam*

Abstract

Privacy is playing a crucial role in the finance-related industries, especially the payment card industry, where verifying salary information securely and transparently becomes an important challenge. Traditional salary verification processes often involve exchanging sensitive financial data (such as payrolls and employment contracts), increasing the concerns about data leakages and other privacy violations. In this study, we address these concerns by proposing a system for salary verification called *zkSalary*, based on the Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK).

The *zkSalary* system provides a secure, transparent and privacy-enhancing solution for both employers and employees without revealing sensitive salary details during the verification process. The proofs can be generated within a few seconds and succint to transfer to the verifier via plaintext or QR code.

We also make our system portable by gather all important modules into a single Javascript library, making it easier to integrate into other existing systems.

Keywords: privacy, zk-SNARK, Groth16, salary verification

Acknowledgements

We would like to express our gratitude to Mr. Nguyen Duy Hieu for his invaluable contributions in proposing the idea and providing us a straightforward approach for studying ZKP. Without his guidance, we would have spent a significant amount of time to complete this project.

Additionally, we extend our sincere thanks to Mr. Nguyen Anh Khoa for offering numerous invaluable pieces of advice, both in the technical and personal aspects of life and work. His insights have proven to be strongly beneficial to our project and personal growth.

1. Introduction

In digital era, where privacy and data security are paramout concerns, the needs for innovative solutions to protect sensitive financial information have never been more pressing. Salary verification processes, a crucial component of almost every people in daily financial activities, often involve exchanging sensitive data such as payrolls and employment contracts. These processes create significant risks about data breaches and privacy violations. In fact, many financial activities (such as opening a credit card) do not require people to show the exact salary amount, instead they only need to prove that their incomes are in a certain range, or maybe just larger or lower than a number. Traditional proofs cannot afford this requirement without showing the exact salary amount in a signed contract or payroll. But zero-knowledge proofs can overcome this challenge. Mathematically, it is possible that a zero-knowledge proof can prove $a \leq x \leq b$ without exposing the actual value of x.

By the inspiration of zero-knowledge proof, we propose *zk-Salary*, a privacy-preserving system for salary verification using one of the most popular zk-SNARK protocols - Groth16 [7]. Employees of a company can generate a proof of their salary in a specified interval range. If the range is valid, a succint, constant-sized proof will be generated within a few seconds and transfer to the verifier via QR code. Unlike any traditional proofs, the proof generated by our system exposes only fundamental details such as the name, ID, the desired range,... while not losing its integrity.

We also try to make our system be as portable as possible by writing all important functionalities into a library, which can be integrated into other bigger systems easily.

2. Related Works

To date, several protocols have been proposed to solve the problem called ”range proof”. A range proof protocol is basically used to prove that a commited value lies in a specified interval range [9]. The idea of range proof was first developed in [3]. A range proof proving a commited value is in interval $[0, 2^n]$ have been proposed in [8] by simply checking the bit length of that number in binary representation. In 2017, another proof system called Bulletproof[4] was published by Bünz et al. and applied in Monero cryptocurrency, which is now running a more robust version called Bulletproof+ [5].

Unfortunately, these methods is inefficient and not suitable for our system's scenerio. Instead, we find that zk-SNARK, which was initial introduced in [2], allow us to solve any problems whenever we can translate these problems into an arithmetic circuit. Many zk-SNARK constructions were proposed including Plonk [6] and Groth16 [7]. These tools give us a more flexible way to solving our problem comparing to range proofs, because we can add other functionalities into the circuit to make it more useful (like including Merkle proof for proving of existence). The Plonk protocol is now more popular than Groth16 because of its smaller verification time complexity in blockchain space, where the proof generation can be done offline on off-chain, no need to worry about proof time complexity. In our scenerio, where the proof generation happens in a centralized server and the verification can be done by any verifier's trusted party, the latter seems much more suitable.

3. Preliminaries

In this section, we introduce definitions of several cryptographic tools that are used throughout the paper.

3.1. Hash functions and the MiMC hash

3.1.1. Hash functions

A hash function is any function H that can be used to map data of arbitrary size to k-length fixed-size values:

$$H : \{0,1\}^* \rightarrow \{0,1\}^k$$

A hash function should hold 3 properties:

1. **Uniformity**: It means that the hash function should map input values as evenly as possible over its output range.
2. **Preimage resistance**: It means that, given a string y from $\{0,1\}^k$ it is infeasible to find a string $x \in \{0,1\}^*$ such that $H(x) = y$.
3. **Collision resistance**: It means that it is infeasible to find two inputs $x_1, x_2 \in \{0,1\}^*$ such that $H(x_1) = H(x_2)$

3.1.2. MiMC hash

MiMC [1] is a "ZK-friendly" hash function over a finite field $\mathbb{F}$, which can be converted to a more efficient arithmetic circuit (3.3.1) for zk-SNARK (3.4).

To hash a single number x, we calculate the following function:

$$MiMC(x) = (F_{r-1} \circ F_{r-2} \circ ... \circ F_0)(x) + k$$

where r is the number of rounds, k is the key and F_i is the round function for round i:

$$F_i(x) = (x + k + c_i)^3$$

where c_i is the round constant ($c_0 = 0$).
Note that all operations in *MiMC* hash are done over $\mathbb{F}$.

3.2. Merkle tree

A Merkle tree is a complete binary tree equipped with a hash function *hash* and an assignment ϕ [10]. For the two child nodes, n_{left} and n_{right}, of any interior node, n_{parent}, the assignment ϕ is required to satisfy

$$\phi(n_{parent}) = hash(\phi(n_{left}) \| \phi(n_{right})) \qquad (1)$$

The function *hash* is a one-way function, such as MiMC.

For each leaf l, the value $\phi(l)$ may be chosen arbitrarily, and then equation 1 determines the values of all the interior nodes. In our application, for simplicity, we choose ϕ be an identity function that maps its input to itself.

This tree is used to verify that the hash of a leaf node is part of the hash of the root node in an efficient way, since we only need a small set of the hashes of the tree (called the Merkle proof) to carry out this verification. In other words, there is a function $MerkleCheck$ such that if $MerkleCheck(h, \text{Merkle proof}) == True$, then h is a hash of a leaf node.

3.3. Statement Representations

The zk-SNARK constructions of any computational statement require converting the computation into an appropriate form, which is the Quadratic Arithmetic Program form, such that the prover can show they have a valid input (witness w) to the computation. To do that, we must first "flatten" the computation into arithmetic circuit (3.3.1). Then we convert this circuit to a Rank-1 Quadratic Constraint System (3.3.2) as an intermediate step to the QAP form (3.3.3).

3.3.1. Arithmetic Circuits

An arithmetic circuit has input gates, output gates, intermediate gates, and directed wires between them. Each gate computes an arithmetic expression of addition or multiplication.

3.3.2. Rank-1 Quadratic Constraint Systems

Rank-1 Quadratic Constraint System (R1CS) is defined as the following set of k many equations:

$$(a_0^i + \Sigma_{j=1}^n a_j^i.I_j + \Sigma_{j=1}^m a_{n+j}^i.W_j) \times (b_0^i + \Sigma_{j=1}^n b_j^i.I_j + \Sigma_{j=1}^m b_{n+j}^i.W_j) \\ = c_0^i + \Sigma_{j=1}^n c_j^i.I_j + \Sigma_{j=1}^m c_{n+j}^i.W_j$$

where $i = 1..k$; n, m and $k \in \mathbb{N}$; and a_j^i, b_j^i and c_j^i are constants from a field $\mathbb{F}$. The parameter k is called the number of constraints, and each equation is called a constraint.

If a pair of field elements $(< I_1, ..., I_n >; < W_1, ..., W_m >)$ satisfies theses equations, $< I_1, ..., I_n >$ is called a instance and $< W_1, ..., W_m >$ is called a witness of the system.

It can be shown that every bounded computation is expressible as a R1CS.

3.3.3. Quadratic Arithmetic Programs

Let $\mathbb{F}$ be a field and R a R1CS over $\mathbb{F}$, a Quadratic Arithmetic Program (QAP) associated to the R1CS R is the following set of polynomials over $\mathbb{F}$:

$$QAP(R) = \{T \in \mathbb{F}[x], \{A_j, B_j, C_j \in \mathbb{F}[x]\}_{j=0}^{n+m}\}$$

where $T(x) = \Pi_{l=1}^k (x - m_l)$ with $m_1, ..., m_j$ are the distinct constants. $T(x)$ is called the target polynomial. A_j, B_j and C_j are the unique degree $k-1$ polynomials defined by the following equation, using Lagrange Interpolation:

$$A_j(m_l) = a_j^l, B_j(m_l) = b_j^l, C_j(m_l) = c_j^l$$

for $l = 1..k$ and $j = 0..(n+m)$.

A pair $(< I_1, ..., I_n >; < W_1, ..., W_m >)$ satisfies R if and only if the following polynomial is divisible by the target polynomial $T(x)$:

$$\begin{aligned} P(x) = &(A_0 + \Sigma_{j=1}^{n} A_j(x) I_j + \Sigma_{j=1}^{m} A_{n+j}(x) W_j) \\ &\times (B_0 + \Sigma_{j=1}^{n} B_j(x) I_j + \Sigma_{j=1}^{m} B_{n+j}(x) W_j) \\ &-(C_0 + \Sigma_{j=1}^{n} C_j(x) I_j + \Sigma_{j=1}^{m} C_{n+j}(x) W_j) \end{aligned}$$

In other words, it means that if there is a low-degree polynomial $H(x) \in \mathbb{F}[x]$ such that $H(x) = \frac{P(x)}{T(x)}$, then all constraints are met.

3.4. Non-interactive Zero-Knowledge Arguments of Knowledge

Non-interactive Zero-Knowledge Arguments of Knowledge (zk-SNARK) allows prover to prove that for a public instance $\mathcal{I} =< I_1, ..., I_n >$, she knows the private witness $\mathcal{W} =< W_1, ..., W_m >$ such that $(\mathcal{I}, \mathcal{W})$ satisfies the a defined statement x. An efficient prover has the following probabilistic polynomial algorithms $(SETUP, PROVE, VERIFY)$:

- $(pk, vk) \leftarrow SETUP$: The setup procedure produces public parameters (1) pk (proving key): a common reference string (CRS) that defines statement x, and (2) vk (verification key): a simulation trapdoor.
- $\pi \leftarrow PROVE(pk, \mathcal{I}, \mathcal{W})$: The prover takes pk and $(\mathcal{I}, \mathcal{W})$ and returns an argument π.
- $0/1 \leftarrow VERIFY(pk, \mathcal{I}, \pi)$: The verifier rejects (0) or accepts (1) the given proof π. The function will return 1 if $(\mathcal{I}, \mathcal{W})$ is satisfied.

4. Proposed System

4.1. System Architecture

The *zkSalary* system consists of three parties:

- **Employer's server**: This is where to store the payrolls and generate zk-SNARK proofs. Each server should be owned by only one employer in order to keep its payroll's secrecy.
- **Employee's app**: Employees use this app to create proof request and receive proof from employer's server. This application requires user authentication to verify the validity of the proof request (preventing unauthorized proof generation).
- **Verifier's app**: This application is used by verifiers to verify the proof sent by an employee. It does not require user authentication from verifiers, since there is no harm if a proof is verified by someone inadvertently.

The system architecture is presented in Fig.1.

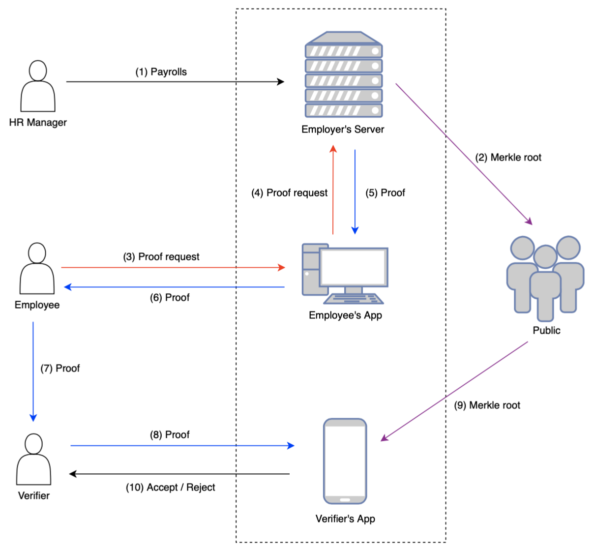

Figure 1: System Architecture

4.2. System Workflow

First, (1) the HR Manager of the employer pushes payrolls to Employer's Server. Then, (2) Employer's Server builds a Merkle Tree for this payrolls and publishes the Merkle root to public, allows any party to use this root for verification processes. Whenever an employee need a salary proof for verification, (3) she logs in to the Employee's App, then submits a proof request which is a form containing two values called *lowerbound* and *upperbound*. (4) The proof request is then sent to Employer's Server. In Employer's Server, (5) it checks the validity of this request, then generates a zk-SNARK proof which is sent back to the Employee's App. At this step, (6) the proof is sent back to the employee via QR code for convenience. With the proof in hand, (7) she can come to any verifier and shows him the proof for verification. Then, (8) the verifier uses his own app to scan the QR code and verify the proof. After verifying the validity of the proof, (9) the app compares the Merkle root in the proof with the Merkle root of this employer which can be found in public. Finally, (10) the Verifier's App returns Accept or Reject status, together with some public data such as Employee's name, *lowerbound*, *upperbound* and the Merkle root.

4.3. Arithmetic Circuits

This subsection describes arithmetic circuits used for proof generation in our system.

In an arithmetic circuit, the public inputs and the private inputs correspond to zk- SNARK's witness $\mathcal{I}$ and instance $\mathcal{W}$, respectively. Private inputs are hidden, and the proof π exposes no information about them. The public inputs is visible to everyone. The verifier uses public inputs and π to verify the relations between public and private inputs declared in the arithmetic circuit.

The details for the circuit are described in Algorithm 1. First, the circuit checks the identification of employee using the Merkle Tree built by payrolls. Then it checks the validity of the request form, which are lowerbound (lower) and upperbound (upper). Finally, it checks if lower ≤ salary ≤ upper, which equivalent to (upper – salary)(salary – lower) ≤ 0 (when lower ≤ upper).

```
Algorithm 1: Arithmetic Circuit
  Public inputs: identifier, root, lower, upper
  Private inputs: Merkle proof, salary
  Output: Decision bit $b$
    1. $h \leftarrow MiMC$(identifier||salary)
    2. Check if $MerkleCheck(h,$ Merkle proof$) == True$
    3. Check if lower ≤ upper
    4. Check if (upper – salary)(salary – lower) ≤ 0
    5. If lines 2, 3 and 4 are $True$ then $b = 1$, else $b = 0$.
```

An implementation of these circuits in Circom language could be found in Appendix A.

5. Implementation

We have implemented a simple web-based version for *zkSalary*, which is available in this repository: HaoPham23/zkSalary. In this system, employer, employee and verifier use a same website which contains three buttons representing the three parties mentioned in 4.1. The employee authentication (step 3 in 4.2) is skipped for simplicity, but it must be implemented for real applications. An example workflow for this system is illustrated in this video.

6. Conclusions

We have proposed *zkSalary* system which solves the problem of privacy-preserving salary verification by utilizing zk-SNARK protocols. We implemented a web-based system for demonstrating the workflow of *zkSalary*, which is shown to be simple and efficient. We also made this demo system to be portable by combining all important functions into a single library, meaning it can be easily integrated into other more complex systems.

In summary, this is a simple system, yet to be implemented in practical applications, and it requires significant refinement to meet the complex demands of the financial sector. However, this system has, to some extent, demonstrated the feasibility of applying Zero-Knowledge Proofs to preserve the individual privacy in the financial sector. It has shown that such an approach is not only possible but potentially efficient.

Appendix A. Circuits Implementation

```
pragma circom  2.0.0;

include "./utils/circuits/comparators.circom";
include "./utils/circuits/mimcsponge.circom";

template HashLeftRight() {
    signal input left;
    signal input right;
    signal output hash;
    // MiMCSponge(nInputs, nRounds, nOutputs)
    component hasher = MiMCSponge(2, 220, 1);
    hasher.ins[0] <== left;
    hasher.ins[1] <== right;
    hasher.k <== 0;
    hash <== hasher.outs[0];
}

template DualMux() {
    signal input in[2];
    signal input s;
    signal output out[2];

    s * (1 - s) === 0;
    out[0] <== (in[1] - in[0])*s + in[0];
    out[1] <== (in[0] - in[1])*s + in[1];
}

template MerkleTreeChecker(levels) {
    signal input leaf;
    signal input pathElements[levels];
    signal input pathIndices[levels];
    signal input root;

    component selectors[levels];
    component hashers[levels];

    for (var i = 0; i < levels; i++) {
        selectors[i] = DualMux();
        selectors[i].in[0] <== i == 0
            ? leaf
            : hashers[i - 1].hash;
        selectors[i].in[1] <== pathElements[i];
        selectors[i].s <== pathIndices[i];

        hashers[i] = HashLeftRight();
        hashers[i].left <== selectors[i].out[0];
        hashers[i].right <== selectors[i].out[1];
    }

    root === hashers[levels - 1].hash;
}

template CheckSalary(levels) {
    signal input identifier;
    signal input salary;

    signal input root;
    signal input pathElements[levels];
    signal input pathIndices[levels];

    signal input lower;
    signal input upper;
    signal output out;

    component hasher = MiMCSponge(2, 220, 1);
    hasher.ins[0] <== identifier;
    hasher.ins[1] <== salary;
    hasher.k <== 0;

    component tree = MerkleTreeChecker(levels);
    tree.leaf <== hasher.outs[0];
    tree.root <== root;
    for (var i = 0; i < levels; i++) {
        tree.pathElements[i] <== pathElements[i];
        tree.pathIndices[i] <== pathIndices[i];
    }
    component validInputChecker = LessEqThan(40);
    validInputChecker.in[0] <== lower;
    validInputChecker.in[1] <== upper;
    validInputChecker.out === 1;

    component checker = LessEqThan(80);
    checker.in[0] <== 0;
    checker.in[1] <==
        (upper - salary) * (salary - lower);

    out <== checker.out;
}

component main {public
    [identifier, root, lower, upper]
                } = CheckSalary(20);
```

References

[1] Martin Albrecht et al. “MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2016, pp. 191–219.

[2] Nir Bitansky et al. “From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again”. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. 2012, pp. 326–349.

[3] Ernest F Brickell et al. “Gradual and verifiable release of a secret”. In: *Advances in Cryptology—CRYPTO’87: Proceedings 7*. Springer. 1988, pp. 156–166.

[4] Benedikt Bünz et al. “Bulletproofs: Short proofs for confidential transactions and more”. In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 315–334.

[5] Heewon Chung et al. “Bulletproofs+: Shorter proofs for a privacy-enhanced distributed ledger”. In: *IEEE Access* 10 (2022), pp. 42081–42096.

[6] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. “Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge”. In: *Cryptology ePrint Archive* (2019).

[7] Jens Groth. “On the size of pairing-based non-interactive arguments”. In: *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer. 2016, pp. 305–326.

[8] Wenbo Mao. “Guaranteed correct sharing of integer factorization with off-line shareholders”. In: *International Workshop on Public Key Cryptography*. Springer. 1998, pp. 60–71.

[9] Eduardo Morais et al. “A survey on zero knowledge range proofs and applications”. In: *SN Applied Sciences* 1 (2019), pp. 1–17.

[10] Michael Szydlo. “Merkle tree traversal in log space and time”. In: *Advances in Cryptology-EUROCRYPT 2004: International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004. Proceedings 23*. Springer. 2004, pp. 541–554.