Split: A Hash-based Memory Optimization Method for Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK)

Huayi Qi, Ye Cheng, Minghui Xu, *Member, IEEE,* Dongxiao Yu, *Senior Member, IEEE,*
Haipeng Wang, Weifeng Lyu

***Abstract*—Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) is a practical zero-knowledge proof system for Rank-1 Constraint Satisfaction (R1CS), enabling privacy preservation and addressing the previous scalability concerns on zero-knowledge proofs. Existing constructions of zk-SNARKs require huge memory overhead to generate proofs in that the size of the zk-SNARK circuit can be large even for a very simple use case, which limits the applications for regular resource-constrained users. To reduce the memory utilization of zk-SNARKs, this paper presents a hash-based method "Split". Concretely, Split intends to partition the zk-SNARK circuits so that components can be processed sequentially while ensuring strong security properties leveraging hash circuits. As a zk-SNARK circuit is partitioned, obsolete variables are no longer preserved in the memory. We further propose an enhanced Split as n-Split, which leads to better optimization by properly choosing multiple splits. Our experimental results validate the effectiveness and efficiency of Split in conserving memory usage for resource-constrained provers as long as the circuit can be partitioned to a Good Split, indicating that via Split zk-SNARKs can be brought one step closer to practical applications.**

***Index Terms*—memory optimization, zk-SNARKs, zero-knowledge proof, privacy**

I. Introduction

Zero-knowledge proofs bring a privacy solution to enable a prover to convince verifiers of a series of statements, without leaking essential information. For instance, one can demonstrate that he/she has permission to enter a room by proving that the identity or biological information satisfies specific access rules, while keeping the information private. There exist a number of privacy-preserving techniques in blockchain-enabled applications based on zero-knowledge proofs, especially in financial fields, such as Zerocash [1], Decentralized Conditional Anonymous Payments [2], and BlockMaze [3]. These applications can protect the information (e.g., identities of sender and receiver, transfer amount, etc.) from leakage without eliminating the transparency brought by blockchain. Apart from finance, zero-knowledge proofs have been widely adopted in other fields such as decentralized file systems [4], smart grids [5], traffic management [6], data trading [7], key management [8], COVID-19 contact tracing [9], and so on. Privacy solutions based on zero-knowledge proofs don't require the support of special hardware such as TEE [10].

Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) [11] is one of the main classes of zero-knowledge constructions. zk-SNARKs provide general solutions for any problem defined in NP languages, and thus can be more widely adopted compared to other approaches [12], [13]. In zk-SNARKs, a problem is described in Rank-1 Constraint Satisfaction (R1CS), an NP-complete language that generalizes arithmetic circuit satisfiability. One of the main features of R1CS is that each variable in R1CS must be preserved during the whole procedure, even if it is a temporary one that only exists for a short period under the view of traditional programming. A garbage collector that cleans out obsolete variables is not applicable in R1CS.

Thus, zk-SNARKs are usually inefficient in memory usage. Provers are supposed to be servers that have plenty of memory modules installed. The huge memory consumption limits the applications of zk-SNARKs in areas where provers don't have sufficient memory, e.g., personal computers and cellphones. For example, regular smart phone users are unable to be a prover in large scale zero-knowledge proofs, although the privacy issues in their smart phones have attracted serious concerns. Take a QAP-based zk-SNARK construction proposed in [14] as an example. Consider a simple circuit that is a loop iteration updating a 32-bit integer $x \leftarrow a \cdot x + b$. It takes 1.173 GB memory for iterating 10,000 times.[1] When the number of iterations grows to 300,000, 32.513 GB memory is occupied. This memory burden grows more severe for deep learning techniques based on zk-SNARKs. To remedy this drawback, SafetyNets [15] uses average pooling instead of max pooling to save memory, despite that the latter performs better; ZEN [16] parallelizes the steps across multiple machines in order to overcome the large memory consumption of the verifiable model training.

Even though the memory overhead of zk-SNARKs is high, few works focus on optimizing the memory usage. In fact, the majority of zk-SNARK constructions do not pay attention to memory consumption. The benchmark for zk-SNARK constructions presented by [17] only focuses on CPU time and

Huayi Qi, Ye Cheng, Minghui Xu (Corresponding Author), and Dongxiao Yu are with School of Computer Science and Technology, Shandong University, Qingdao 266237, China (e-mail: qi@huayi.email, chengye0311@163.com, {mhxu, dxyu}@sdu.edu.cn).

Haipeng Wang is with Institute of Information Fusion, Naval Aviation University, Yantai 264001, China (e-mail: whp5691@163.com).

Weifeng Lyu is with School of Computer Science and Technology, Beihang University, Beijing 100191, China (e-mail: lwf@nlsde.buaa.edu.cn).

[1]The data are retrieved from our evaluation in Sec. V.

proof size, ignoring the memory usage, although the authors claimed that insufficient memory can obviously affect the performance according to the test results. Due to the memory constraints of large circuits, Wu et al. [18] asked a question: *can zk-SNARKs be used for circuits of much larger sizes, and at what cost?* Their solution is to distribute the circuits into multiple clusters, which improves the performance, but for regular personal computers or laptops with no clusters, their scheme is not applicable. This motivates us to ask the following question: ***can zk-SNARKs be used for circuits of much larger sizes on a single machine, and at what cost measured in both CPU time and memory?***

To answer these questions, we propose Split, which can partition a zk-SNARK circuit into two components that can be processed sequentially. As the circuit is partitioned, obsolete variables are no longer needed to be preserved in memory. We further provide an enhanced Split, denoted as n-Split, which contributes to better optimization by properly choosing multiple splits. To ensure the integrity of the partition and guarantee strong security, hash circuits are involved to prevent malicious users from changing the input/output variables of each component. In our proposed scheme, the challenge is to partition the circuit. A zk-SNARK circuit can be considered as a directed acyclic graph, where variables and constraints are presented as nodes. Our proposed scheme partition the graph into two disjoint subsets. For every edge across two subsets there is an intermediate variable as the input of the hash function, which brings additional CPU and memory overhead. It is challenging to partition the graph in order to reduce the memory usage while having a reasonable CPU overhead.

Our contributions are summarized as follows:

- We introduce Split, a hash-based memory optimization method for zk-SNARKs that can efficiently reduce the memory usage as long as it can be partitioned as a Good Split, especially for loop-based circuits. To our best knowledge, Split is the first method for zk-SNARKs aiming at improving memory usage on a single machine without sufficient memory.
- We rigorously define Split to partition a circuit into two components which can be processed sequentially in memory, and introduce the concept of Good Split. These definitions provide guidance in properly choosing a split mechanism. We further extend the concept of Split to n-Split that partitions a circuit into n components.
- We provide analysis to demonstrate that splitting a circuit does not compromise its security properties compared to the original one. We also conduct experiments to evaluate the proof-of-concept of Split and n-Split. Our experimental results demonstrate that Split can reduce the memory usage of a commonly used loop structure roughly to 50% for 2-Split with little time increase, and to 27% for 5-Split with about 20% of additional CPU time, enabling zk-SNARKs to be applicable for resource-constrained provers.

This paper is organized as follows. In Sec. II we introduce the most related work and the components of popular zk-SNARK constructions. In Sec. III, we define Split, Good Split, n-Split, and the corresponding zero-knowledge proof procedures, explain how the memory usage gets reduced via Split, and demonstrate Split via two examples. Security and performance analysis, as well as the experimental results, are reported in Sec. IV and Sec. V, respectively. Finally in Sec. VI, we conclude this paper and discuss future research.

II. Related Work and Preliminaries

A. Related Work

In this subsection, we first introduce zk-SNARK constructions on which our Split scheme can be applied, namely QAP-based zk-SNARKs and IOP-based zk-SNARKs. Then we overview previous works aiming at decreasing memory usage, including optimization in specific areas and distributing the memory into a cluster.

QAP-based zk-SNARKs. Such constructions [19] [14] [20] reduce R1CS to Quadratic Arithmetic Program (QAP) instances. The proof size is constant, and the verification procedure is efficient in milliseconds, making them suitable for blockchain-based applications, especially wireless blockchain [10], [21], where both persistent storage space and CPU resource are extremely valuable and the memory capacity of verifiers is constrained. Thus, QAP-based zk-SNARK is the most widely-used class of zk-SNARK constructions.

IOP-based zk-SNARKs. zk-SNARKs can be constructed from interactive oracle proofs (IOPs) via BCS transformation [22]. IOP-based constructions [23] [24] [25] support the R1CS language and have better performance for provers measured in CPU time compared to QAP-based zk-SNARKs.

Memory optimization in specific areas. A lightweight authentication scheme, TinyZKP [26], based on zero-knowledge proofs, was implemented on TinyOS-based sensor nodes. This scheme reduces memory usage by getting rid of the complex key management in authentication. Khernane et al. [27] presented the design and evaluation of a secure lightweight and energy-efficient authentication scheme, BANZKP, based on zero-knowledge proof and a commitment scheme. BANZKP decreases the size of a pre-distributed set of keys and the memory usage by 56% compared with TinyZKP. Their optimizations are directly performed by optimizing the authentication algorithm to reduce the size of zero-knowledge proof codes, which are only applicable in authentication schemes. An application-independent scheme is needed to minimize memory usage for various zero-knowledge proof applications.

Extending the memory occupation into clusters. Wu et al. [18] presented a construction that distributes the generation of a zero-knowledge proof across multiple machines in a computer cluster, mainly because of the memory resource limitation. Such a mechanism should facilitate the adoption of zk-SNARKs in cloud-based applications [28].

B. Preliminaries

In this subsection, we first define zk-SNARK, introducing its main procedures and security properties. Then we present R1CS, the "interface" between the front end and the back end of a zk-SNARK system, as well as circuit, an equivalent

representation of R1CS. Finally, we introduce high-level programming languages and QAP, which are the corresponding typical implementations of the front and back end.

1) zk-SNARKs: A zk-SNARK system is used to prove that a prover knows a set of values such that $\mathbf{y}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{priv}} = F(\mathbf{x}_{\mathsf{pub}}, \mathbf{x}_{\mathsf{priv}})$, where F refers to a circuit that acts like a function, $\mathbf{x}_{\mathsf{pub}}$, $\mathbf{x}_{\mathsf{priv}}$ refers to the public/secret inputs, and $\mathbf{y}_{\mathsf{pub}}$, $\mathbf{y}_{\mathsf{priv}}$ the public/secret outputs.[2] No information about the values of secret inputs/outputs is leaked.

A zk-SNARK system usually consists of the following three procedures.

- $\mathsf{Setup}(1^{\mathcal{K}}, F) \rightarrow (\mathsf{pk}, \mathsf{vk})$.
  Given a security parameter $1^{\mathcal{K}}$ and a zk-SNARK circuit F, a trusted party generates public parameters consisting of a proving key pk and a verification key vk. The memory usage of this procedure is high, but considering that for an F, the setup is needed to be run only once, and it is executed by a trusted party that is supposed to have sufficient hardware, the memory usage issue in this procedure can be tolerated.
- $\mathsf{Prove}(F, \mathbf{x}_{\mathsf{pub}}, \mathbf{x}_{\mathsf{priv}}, \mathbf{y}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{priv}}, \mathsf{pk}) \rightarrow \pi$.
  Given a zk-SNARK circuit F, public/secret inputs $\mathbf{x}_{\mathsf{pub}}$, $\mathbf{x}_{\mathsf{priv}}$, public/secret outputs $\mathbf{y}_{\mathsf{pub}}$, $\mathbf{y}_{\mathsf{priv}}$, and a proving key pk, a prover outputs a proof π that proves $\mathbf{y}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{priv}} = F(\mathbf{x}_{\mathsf{pub}}, \mathbf{x}_{\mathsf{priv}})$ without revealing any information about $\mathbf{x}_{\mathsf{priv}}$ and $\mathbf{y}_{\mathsf{priv}}$. This is an expensive procedure that requires a huge amount of resources in both memory and CPU time. As end users that need privacy protection may not have sufficient memory capacity, memory optimization is needed for them to run this procedure.
- $\mathsf{Verify}(\mathbf{x}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{pub}}, \pi, \mathsf{vk}) \rightarrow \{0, 1\}$.
  Given public inputs/outputs $\mathbf{x}_{\mathsf{pub}}$, $\mathbf{y}_{\mathsf{pub}}$, a proof π, and a verification key vk, a verifier can verify the proof π and output 1 if and only if π passes the verification. This procedure is efficient; it takes time in milliseconds for QAP-based zk-SNARKs.

zk-SNARKs satisfy the following three properties:

- *Completeness*. Knowing $\mathbf{x}_{\mathsf{priv}}, \mathbf{y}_{\mathsf{priv}}$ to the validity of a statement $\mathbf{y}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{priv}} = F(\mathbf{x}_{\mathsf{pub}}, \mathbf{x}_{\mathsf{priv}})$, the prover is able to convince the verifier.
- *Soundness*. A malicious prover is not able to convince the verifier in the case that $\mathbf{y}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{priv}} \neq F(\mathbf{x}_{\mathsf{pub}}, \mathbf{x}_{\mathsf{priv}})$.
- *Zero-knowledge*. The verifier learns nothing about $\mathbf{x}_{\mathsf{priv}}$ and $\mathbf{y}_{\mathsf{priv}}$.

To better illustrate how zk-SNARK systems work, we present the basic structure of a zk-SNARK system considered in this paper in Fig. 1.

2) High-level Programming Languages: Developers use high-level programming languages such as xJsnark [29], ZoKrates [30], and Leo [31] to express the programming logic of a zk-SNARK, which is further compiled into R1CS language by compiler. A typical programming code contains condition statements, loop statements, function calls, and recursions. Due to the limitation of the R1CS language, these statements are unrolled by the compiler, which is introduced in Sec. II-B5. Thus, even a simple program could be compiled into a very large R1CS instance, and therefore requires a huge amount of memory.

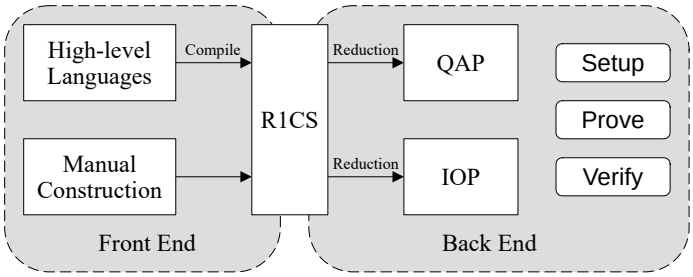


Fig. 1: The basic structure of zk-SNARK systems

3) Rank-1 Constraint Satisfaction (R1CS) and the Arithmetic Circuit: An R1CS instance ϕ over a prime field $\mathbb{F}$ is a tuple $(k, N, M, \mathbf{a}, \mathbf{b}, \mathbf{c})$, where $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are three matrices over $\mathbb{F}$, M is the number of constraints, N is the number of all variables, and k is the number of public variables. We define the size of a zk-SNARK circuit F as $|F| = N$.

For a solution vector $\mathbf{s} = (s_1, s_2, s_3, \ldots, s_N)$ (the first k values are visible to everyone), the following condition is met for all $i \in [1, M]$:

$$(\sum_{j=1}^{N} \mathbf{a}_{i,j} \cdot \mathbf{s}_j) \cdot (\sum_{j=1}^{N} \mathbf{b}_{i,j} \cdot \mathbf{s}_j) - (\sum_{j=1}^{N} \mathbf{c}_{i,j} \cdot \mathbf{s}_j) = 0$$

Note that although not mandatory, it is often assumed that the first variable is a public one with value $s_1 = 1$. Each row in the three matrices stands for a constraint that consists of addition, subtraction, and multiplication operations. For example, the following matrices

$$\mathbf{a} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}, \mathbf{c} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix},$$

require a solution vector $(1, s_2, s_3)$ with the following two constraints: $s_2 = 1 + s_3$ and $s_3 = -s_2$.

R1CS is widely used in nearly all zk-SNARK constructions.[3] Note that the matrix definition of R1CS is good to express constraints, thus easy to be handled by back-end implementations, but it is not straightforward enough to front end. Although a prover needs to provide such a solution vector in order to prove a statement, in the front end, the variable values are derived from calculations instead of "guessing". A prover actually fills the input values, carries out calculations, and finally obtains all values. An equivalent of R1CS, a (arithmetic) circuit, is more straightforward to the front-end implementations, which is introduced here.

An arithmetic circuit is a directed, connected, unweighted, acyclic graph $G = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ contains two kinds of nodes: variables $\mathbf{V}_{\mathsf{var}}$ (also known as wires) and basic constraints $\mathbf{V}_{\mathsf{con}}$ (also known as gates). An edge only exists for the connection between a variable and a constraint: $\forall (v_1, v_2) \in$

[2]In other studies, a zk-SNARK system is often defined as proving $F(\mathbf{x}, \mathbf{w}) = 1$, while $\mathbf{x}$ and $\mathbf{w}$ stands for all public and secret variables respectively. These two definitions are equivalent. Our choice is for the sake of simplicity in scheme design, as our work resides in the front end.

[3]For those zk-SNARK back ends that do not aim at R1CS, e.g., PLONK, R1CS wrappers can be adopted.

$\mathbf{E}, (v_1 \in \mathbf{V}_{\mathsf{var}} \wedge v_2 \in \mathbf{V}_{\mathsf{con}}) \vee (v_1 \in \mathbf{V}_{\mathsf{con}} \wedge v_2 \in \mathbf{V}_{\mathsf{var}})$. All source nodes are called input variables: $\mathbf{V}_{\mathsf{input}} = \{v \in \mathbf{V} | v \neq v_2, \forall (v_1, v_2) \in \mathbf{E}\} \subset \mathbf{V}_{\mathsf{var}}$, and all sink nodes are called output variables: $\mathbf{V}_{\mathsf{output}} = \{v \in \mathbf{V} | v \neq v_1, \forall (v_1, v_2) \in \mathbf{E}\} \subset \mathbf{V}_{\mathsf{var}}$. A constraint node contains a meta tag identifying the operator, denoted as $\mathsf{tag}_{\mathsf{operator}}(v)$, which can be addition, subtraction, or multiplication. A variable node contains a meta tag identifying whether the variable is a public one or a secret one, denoted as $\mathsf{tag}_{\mathsf{visibility}}(v)$. It may also contain a meta tag $\mathsf{tag}_{\mathsf{value}}(v) \in \mathbb{F}_p$ for the value when a prover is calculating or proving, but it must not contain a value when the circuit is just constructed at the design stage, except for public variables that are designated as constants. Besides, although not mandatory, intermediate variables $\mathbf{V}_{\mathsf{inter}} = \mathbf{V}_{\mathsf{var}} - \mathbf{V}_{\mathsf{input}} - \mathbf{V}_{\mathsf{output}}$ are all secret ones.

Before a prover proves by following the back-end protocol, he/she needs to fill all variables in R1CS (the circuit) with values by calculation. The prover first fills all input variables with any value he/she selects. Then, all the nodes are traversed in the topological order. When a constraint node $v_{\mathsf{con}} \in \mathbf{V}_{\mathsf{con}}$ is visited, all the previous nodes $\{v_1 | \forall (v_1, v_{\mathsf{con}}) \in \mathbf{E}\}$ are already filled with values. The calculation result is then saved in the constraint node denoted as $\mathsf{tag}_{\mathsf{result}}(v_{\mathsf{con}})$. When a variable node $v_{\mathsf{var}} \in \mathbf{V}_{\mathsf{var}}$ is visited, the value can be copied from its parent constraint nodes $\{v_1 | \forall (v_1, v_{\mathsf{var}}) \in \mathbf{E}\}$. Note that there can be more than one parent constraint node of v_{var}, and in this case, if these constraint nodes have different values, the calculation aborts.[4] After all the nodes are traversed and processed in this way, all the variables have been filled with values that meet the constraints.

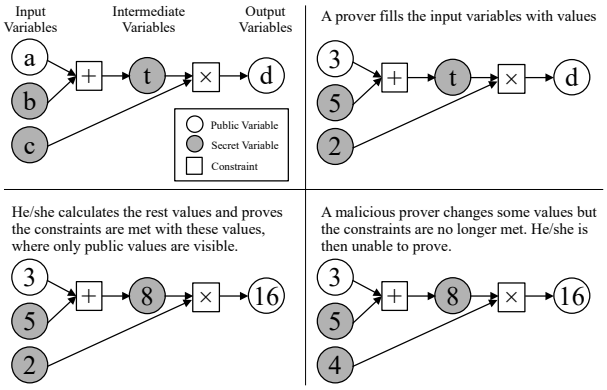


Fig. 2: How zk-SNARKs work in the front-end view

4) How zk-SNARKs Work from the Front-End View: A zk-SNARK system enables a prover to prove that he/she owns a group of variables satisfying the given constraints, where some values of the variables are secret. For instance, considering a constraint $d = (a+b) \cdot c$, where a and d are public. The prover first generates the inputs a, b, and c, and calculates $t \leftarrow a + b$, $d \leftarrow t \cdot c$. He/she then publishes the values of a and d, and runs a zero-knowledge protocol to get a proof. Verifiers are then convinced that the prover actually owns valid b and c. Fig. 2 illustrates the constraints of variables mentioned above as a graph. The prover fills the input values he/she chooses to the graph and calculates the rest. When all the values are calculated, he/she can then prove that he owns this group of values, in which some (input/output) values are visible to everyone, while the rest are secret.

To implement such a zk-SNARK system, we need to consider two problems:

1) Given a standard form of constraints of variables, design a protocol that enables the prover to convince others.
2) Given the constraints of variables that are described in other languages, such as a formula written in a human-readable form or a programming code, convert it to the standard form.

A solution to the first problem is often called the back end of a zk-SNARK system, while the front end corresponds to the latter. Most zk-SNARK systems choose R1CS as the standard form of constraints of variables, which are defined in multiplication of matrices. An equivalent concept of R1CS, a circuit, is a group of constraints of variables in a graph form, where the constraints are limited to addition, subtraction, and multiplication in $\mathbb{F}_p$. The back end of a zk-SNARK system is significant and thus draws the most attention in recent years. Different cryptographic methods have been adopted to construct different zk-SNARK systems such as Groth16 [20] and Aurora [24]. The front end seems to be more like an engineering issue, mainly for the compilers to convert high-level programming languages into R1CS, even though some developers prefer writing the R1CS by hand.

A number of works focus on the back end. However, the front end is as significant considering the security properties and performance of the system. A bad front-end implementation might cause the constraints badly designed such that malicious provers are able to generate a proof with arbitrary values without being noticed, or cause unnecessary variables or constraints that incur more overhead. Since our optimization target is a front-end thing, we focus on the front end, i.e optimizing the R1CS so that less memory is occupied, no matter what specific back end is used.

5) Why zk-SNARKs Take Huge Resources: zk-SNARKs are completely different from a regular computer. In a regular computer, the main objective is to calculate the result, thus a variable is not needed anymore as long as no other variables rely on it, and is wiped out by either the garbage collector or the compiler. Therefore, jumps, conditions, loops, and random accesses are implemented efficiently.

In zk-SNARKs, recall that a prover first calculates every variable in the constraints, then generates a proof that depends on these variables. As a consequence, all the intermediate variables must be preserved. What's worse, as the circuit only supports addition, subtraction, and multiplication in $\mathbb{F}_p$, it does not natively support jumps, conditions, loops, and random accesses. In most zk-SNARK front ends, these features are implemented by unwinding, which is a technique mostly used in regular compilers, originally for optimization. Fig. 3 demonstrates the unwinding technique used by zk-SNARK front

[4]Even if a malicious prover refuses to abort the calculation in this case, the prove procedure would fail by the back-end protocol, as the constraints are not met.

```
   // Loop - Before unwinding.
1  for i ← 1 to 3 do
2  |  a ← a + b;
3  end
   // Loop - After unwinding.
4  a_1 ← a + b;
5  a_2 ← a_1 + b;
6  a_3 ← a_2 + b;
   // Condition - Before unwinding.
7  if a = 1 then
   |  // a is a boolean, i.e. the value is either
   |     0 or 1.
8  |  b = c;
9  else
10 |  b = d;
11 end
   // Condition - After unwinding.
12 t_1 = c − d;
13 t_2 = t_1 × a;
14 b = t_2 + d;
   // Random access - Before unwinding.
15 a ← b[i];
   // Random access - After unwinding. The
      following just contains the first step. The
      If statement needs unwinding, too.
16 if i = 0 then
17 |  a ← b_0;
18 end
19 if i = 1 then
20 |  a ← b_1;
21 end
22 if i = 2 then
23 |  a ← b_2;
24 end
   // Until all possible values of i are
      enumerated.
```

Fig. 3: The unwinding technique used by zk-SNARK front ends

ends.[5] It is obviously extremely inefficient, but is currently the only way to support such a calculation. Undoubtedly, even a single instruction in a regular computer can bring a huge number of intermediate variables, and thus incur significant resource overhead in both CPU time and memory occupation. As a result, a zk-SNARK circuit for a simple purpose might require hundreds of GB memory.

6) Quadratic Arithmetic Programs (QAP): Popular zk-SNARK systems [19] [14] [20] reduce R1CS instances into QAP instances [32]. A QAP instance Φ is a tuple $(k, N, M, \mathbf{A}, \mathbf{B}, \mathbf{C}, D)$, where k and N are defined in the same way as those in R1CS, $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are vectors of polynomials over prime field $\mathbb{F}$ of degree M, and D is a subset of $\mathbb{F}$ of size M. The QAP degree M determines the time cost as well as the memory occupation in both Setup and Prove procedures.

The two reductions from an R1CS instance to a QAP instance are called *QAP instance reduction* and *QAP witness reduction*. In theory, $M_{\text{QAP}} = M_{\text{R1CS}}$ is the minimum suitable QAP degree[6] for an R1CS instance with k_{R1CS} public inputs and M_{R1CS} constraints. Besides, in order to efficiently multiply polynomials, Fast Fourier Transform (FFT) is adopted in real systems, which might result in a larger QAP degree.

[5]Binary exponentiation is used by zk-SNARK front ends to reduce such a crazy loop unwinding whenever possible. But unfortunately, not all loops can be reduced in this way. For example, a loop updating $x \leftarrow a \cdot x + b$ can not take advantage of binary exponentiation.

TABLE I: Notions

Symbol	Description
$S = (F_1, F_2)$	Split
$\mathcal{H}$	Hash function
$F_{\mathcal{H}}$	Hash circuit
$\mathbf{x}_{\mathsf{pub}}$, $\mathbf{y}_{\mathsf{pub}}$	Public inputs/outputs of a circuit
$\mathbf{x}_{\mathsf{priv}}$, $\mathbf{y}_{\mathsf{priv}}$	Private inputs/outputs of a circuit
$\mathbf{m}$	Intermediate variables
$\mathbf{h}$	Hash digest of $\mathbf{m}$
pk, vk	Proving/verification key of a circuit
π	Proof to be verified
$\lvert\mathbf{m}\rvert$, $\lvert F\rvert$	The number of variables in $\mathbf{m}$ and F
$S = (F_1, F_2, ..., F_n)$	n-Split

III. Split Design

In this section, we first define Split that partitions a circuit into two components, and present the zero-knowledge proof procedures of Split, followed by the definition of Good Split. Then, we extend Split to n-Split that partitions a circuit into multiple components. Finally, we justify why the memory usage gets reduced via Split and present examples for illustration purposes. The symbols used in this section are listed in Table I.

A. Split and Good Split

Definition 3.1: A Split $S = (F_1, F_2)$ for a zk-SNARK circuit $\mathbf{y}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{priv}} = F(\mathbf{x}_{\mathsf{pub}}, \mathbf{x}_{\mathsf{priv}})$ is a tuple (F_1, F_2), where F_1 and F_2 are called split circuits, such that:

- Circuit $\mathbf{h}, \mathbf{m} = F_1(\mathbf{x}_{\mathsf{pub}}, \mathbf{x}_{\mathsf{priv}})$, where $\mathbf{h}$ are public variables and $\mathbf{m}$ are secret variables. F_1 ensures that $\mathbf{h} \leftarrow \mathcal{H}(\mathbf{m})$, where $\mathcal{H}$ is a hash function.
- Circuit $\mathbf{y}_{\mathsf{pub}} \cap \mathbf{h}', \mathbf{y}_{\mathsf{priv}} = F_2(\mathbf{x}_{\mathsf{pub}}, \mathbf{m})$, where $\mathbf{h}'$ are public variables. F_2 ensures that $\mathbf{h}' \leftarrow \mathcal{H}(\mathbf{m})$ and $\mathbf{y}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{priv}} = F(\mathbf{x}_{\mathsf{pub}}, \mathbf{x}_{\mathsf{priv}})$.

Fig. 4 demonstrates the split of a circuit and the corresponding original circuit.

We present how zero-knowledge proof procedures Setup, Prove, and Verify are defined with Split.

- $\mathsf{Setup}_{\mathsf{S}}(1^{\mathcal{K}}, F, S) \rightarrow (\mathsf{pk}_{\mathsf{S}}, \mathsf{vk}_{\mathsf{S}})$.
  Given a security parameter $1^{\mathcal{K}}$, a circuit F and a split $S = (F_1, F_2)$, a trusted party runs $(\mathsf{pk}_1, \mathsf{vk}_1) \leftarrow \mathsf{Setup}(1^{\mathcal{K}}, F_1)$ and $(\mathsf{pk}_2, \mathsf{vk}_2) \leftarrow \mathsf{Setup}(1^{\mathcal{K}}, F_2)$, then outputs a proving key $\mathsf{pk}_{\mathsf{S}} = (\mathsf{pk}_1, \mathsf{pk}_2)$ and a verification key $\mathsf{vk}_{\mathsf{S}} = (\mathsf{vk}_1, \mathsf{vk}_2)$.

[6]In the real implementation in libsnark, the minimum suitable QAP degree is $M_{\text{QAP}} = M_{\text{R1CS}} + k_{\text{R1CS}} + 1$ as additional constraints are involved to ensure soundness of input consistency.

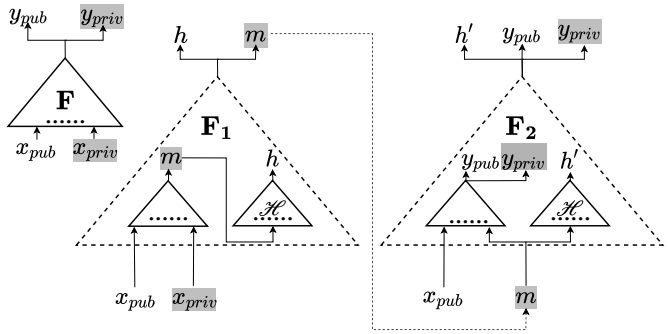


Fig. 4: The split (F_1, F_2) of original circuit F. Variables in shadow are secret ones.

- $\mathsf{Prove}_\mathsf{S}(S, \mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv}, \mathbf{y}_\mathsf{pub}, \mathbf{y}_\mathsf{priv}, \mathsf{pk}_\mathsf{S}) \rightarrow \pi_\mathsf{S}$.
  Given split $S = (F_1, F_2)$, variables $\mathbf{x}_\mathsf{pub}$, $\mathbf{x}_\mathsf{priv}$, $\mathbf{y}_\mathsf{pub}$, $\mathbf{y}_\mathsf{priv}$, and a proving key $\mathsf{pk}_\mathsf{S} = (\mathsf{pk}_1, \mathsf{pk}_2)$, a prover first gets $(\mathbf{h}, \mathbf{m}) \leftarrow F_1(\mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv})$, then runs $\pi_1 \leftarrow \mathsf{Prove}(F_1, \mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv}, \mathbf{h}, \mathbf{m}, \mathsf{pk}_1)$ and $\pi_2 \leftarrow \mathsf{Prove}(F_2, \mathbf{x}_\mathsf{pub}, \mathbf{m}, \mathbf{h}' \cap \mathbf{y}_\mathsf{pub}, \mathbf{y}_\mathsf{priv}, \mathsf{pk}_\mathbf{2})$, finally outputs a proof $\pi_\mathsf{S} = (\pi_1, \pi_2)$.
- $\mathsf{Verify}_\mathsf{S}(\mathbf{x}_\mathsf{pub}, \mathbf{y}_\mathsf{pub}, \mathbf{h}, \pi_\mathsf{S}, \mathsf{vk}_\mathsf{S}) \rightarrow \{0, 1\}$.
  Given public variables $\mathbf{x}_\mathsf{pub}$, $\mathbf{y}_\mathsf{pub}$, $\mathbf{h}$, a proof π_S, and a verification key $\mathsf{vk}_\mathsf{S} = (\mathsf{vk}_1, \mathsf{vk}_2)$, a verifier runs $v_1 \leftarrow \mathsf{Verify}(\mathbf{x}_\mathsf{pub}, \mathbf{h}, \pi_1, \mathsf{vk}_1)$ and $v_2 \leftarrow \mathsf{Verify}(\mathbf{x}_\mathsf{pub}, \mathbf{y}_\mathsf{pub} \cap \mathbf{h}, \pi_2, \mathsf{vk}_2)$, and then outputs $v_1 \cdot v_2$.

One might wonder whether Split can be done for any circuit, and at random positions. Unfortunately, the answer is no, considering performance impacts. Here, we reduce the problem of finding a split that decreases memory usage most effectively with the least overhead into a general graph problem.

Problem 3.1: Let $G = (\mathbf{V}, \mathbf{E})$ be a directed, connected, unweighted, acyclic graph. Let s be a source node, and t be a sink node. Denote by $(\mathbf{S}, \mathbf{T})$ a cut of G, which is a node partition such that $s \in \mathbf{S}$ and $t \in \mathbf{T}$. Let $\mathsf{capacity}(\mathbf{S}, \mathbf{T})$ be the number of edges leaving $\mathbf{S}$ and entering $\mathbf{T}$. Find a cut such that

- $f(\mathbf{S}, \mathbf{T}) = \max\{|\mathbf{S}|, |\mathbf{T}|\} + H \cdot \mathsf{capacity}(\mathbf{S}, \mathbf{T})$ gets minimized, where $H > 0$ is a constant number, and
- $f(\mathbf{S}, \mathbf{T}) < |\mathbf{V}|$

Reduction: Given a circuit $\mathbf{y}_\mathsf{pub}, \mathbf{y}_\mathsf{priv} = F(\mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv})$, and denote all variables as $\mathbf{V}_\mathsf{var}$. Let $\mathbf{V} = \mathbf{V}_\mathsf{var} - \mathbf{x}_\mathsf{pub}$. For any two variables x, y in $\mathbf{V}$, $(x, y) \in \mathbf{E}$ if and only if x is a parent variable of y in the circuit. Solve Problem 3.1 for graph $G = (\mathbf{V}, \mathbf{E})$, and calculate an optimal cut $(\mathbf{S}, \mathbf{T})$. For our purpose we require that $\mathbf{x}_\mathsf{priv} \subset \mathbf{S}$, $\mathbf{y}_\mathsf{pub} \subset \mathbf{T}$, and $\mathbf{y}_\mathsf{priv} \subset \mathbf{T}$. If all these conditions hold, we let $\mathbf{m}$ be the collection of nodes with at least one edge leaving $\mathbf{S}$. Then the circuit F is split into $\mathbf{h}, \mathbf{m} = F_1(\mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv})$ and $\mathbf{y}_\mathsf{pub} \cap \mathbf{h}', \mathbf{y}_\mathsf{priv} = F_2(\mathbf{x}_\mathsf{pub}, \mathbf{m})$, where $\mathbf{h} = \mathbf{h}' = \mathcal{H}(\mathbf{m})$. If any of the above relationships does not hold, one can perform the following transformation: for $\forall v_x \in \mathbf{x}_\mathsf{priv} \not\subset \mathbf{S}$, all the calculations of v_x are in F_2, thus we add a variable m_x into $\mathbf{m}$ such that $m_x \leftarrow v_x \cdot 1$ in F_1, and replace v_x with m_x in F_2. Similarly, for $\forall v_y \in \mathbf{y}_\mathsf{pub} \cap \mathbf{y}_\mathsf{priv} \not\subset \mathbf{T}$, the output value is already calculated in F_1, therefore we add m_y into $\mathbf{m}$ such that $m_y \leftarrow v_y \cdot 1$ in F_1, and $v'_y \leftarrow m_y \cdot 1$ in F_2. After this transformation, one gets a split instance reduced from the original circuit.

However, since Problem 3.1 is hard to solve, we turn to consider Good Split, which reduces memory usage effectively without much overhead. Note that not every circuit accepts a good split, and only those circuits whose minimum cuts are sufficiently small might have good splits.

Definition 3.2: A Good Split is a Split $S = (F_1, F_2)$ such that:

- $|\mathbf{m}| \ll |F|$, where m is defined in Def. 3.1 and $|F|$ is defined in Sec. II-B3.
- $max\{|F_1|, |F_2|\}$ is relatively small.

The performance analysis is given in Sec. IV-B.

B. n-Split

A Good Split reduces the memory usage effectively. Then multiple Good Splits should result in an even better optimization. Here, we extend the concept of Split to n-Split.

Definition 3.3: An n-Split, where $n \geq 3$, for a zk-SNARK circuit $\mathbf{y}_\mathsf{pub}, \mathbf{y}_\mathsf{priv} = F(\mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv})$ is a tuple $(F_{i\in[1,n]})$, where $F_{i\in[1,n]}$ are n circuits, such that:

- Circuit $\mathbf{h_1}, \mathbf{m_1} = F_1(\mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv})$, where $\mathbf{h_1}$ are public variables and $\mathbf{m}$ are secret variables. F_1 ensures that $\mathbf{h_1} \leftarrow \mathcal{H}(\mathbf{m_1})$, with $\mathcal{H}$ being a hash function.
- Circuit $\mathbf{h'_{i-1}} \cap \mathbf{h_i}, \mathbf{m_i} = F_i(\mathbf{x}_\mathsf{pub}, \mathbf{m_{i-1}})$, where $\mathbf{h'_{i-1}}$ and $\mathbf{h_i}$ are public variables and $\mathbf{m_i}$ are secret variables. F_i ensures that $\mathbf{h'_{i-1}} \leftarrow \mathcal{H}(\mathbf{m_{i-1}})$ and $\mathbf{h_i} \leftarrow \mathcal{H}(\mathbf{m_i})$. $(i \in [2, n-1])$
- Circuit $\mathbf{y}_\mathsf{pub} \cap \mathbf{h'_{n-1}}, \mathbf{y}_\mathsf{priv} = F_n(\mathbf{x}_\mathsf{pub}, \mathbf{m_{n-1}})$, where $\mathbf{h'_{n-1}}$ are public variables. F_n ensures that $\mathbf{h'_{n-1}} \leftarrow \mathcal{H}(\mathbf{m_{n-1}})$, and $\mathbf{y}_\mathsf{pub}, \mathbf{y}_\mathsf{priv} = F(\mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv})$.

Similarly, we present the procedures Setup, Prove, and Verify for n-Split here.

- $\mathsf{Setup}_\mathsf{MS}(1^\mathcal{K}, F, S) \rightarrow (\mathsf{pk}_\mathsf{MS}, \mathsf{vk}_\mathsf{MS})$.
  Given a security parameter $1^\mathcal{K}$, a circuit F and split $S = (F_{i\in[1,n]})$, a trusted party runs $(\mathsf{pk}_i, \mathsf{vk}_i) \leftarrow \mathsf{Setup}(1^\mathcal{K}, F_i)$ $(i \in [1, n])$, and then outputs a proving key $\mathsf{pk}_\mathsf{MS} = (\mathsf{pk}_{i\in[1,n]})$ and a verification key $\mathsf{vk}_\mathsf{MS} = (\mathsf{vk}_{i\in[1,n]})$.
- $\mathsf{Prove}_\mathsf{MS}(S, \mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv}, \mathbf{y}_\mathsf{pub}, \mathbf{y}_\mathsf{priv}, \mathsf{pk}_\mathsf{MS}) \rightarrow \pi_\mathsf{MS}$.
  Given a split $S = (F_{i\in[1,n]})$, variables $\mathbf{x}_\mathsf{pub}$, $\mathbf{x}_\mathsf{priv}$, $\mathbf{y}_\mathsf{pub}$, $\mathbf{y}_\mathsf{priv}$, and a proving key $\mathsf{pk}_\mathsf{MS} = (\mathsf{pk}_{i\in[1,n]})$, a prover runs:
  - $(\mathbf{h_1}, \mathbf{m_1}) \leftarrow F_1(\mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv})$
  - $(\mathbf{h'_{i-1}} \cap \mathbf{h_i}, \mathbf{m_i}) \leftarrow F_i(\mathbf{x}_\mathsf{pub}, \mathbf{m_{i-1}})$ $(i \in [2, n-1])$
  - $(\mathbf{y}_\mathsf{pub} \cap \mathbf{h'_{n-1}}, \mathbf{y}_\mathsf{priv}) \leftarrow F_n(\mathbf{x}_\mathsf{pub}, \mathbf{m_{n-1}})$
  - $\pi_1 \leftarrow \mathsf{Prove}(F_1, \mathbf{x}_\mathsf{pub}, \mathbf{x}_\mathsf{priv}, \mathbf{h_1}, \mathbf{m_1}, \mathsf{pk}_1)$
  - $\pi_i \leftarrow \mathsf{Prove}(F_i, \mathbf{x}_\mathsf{pub}, \mathbf{m_{i-1}}, \mathbf{h'_{i-1}} \cap \mathbf{h_i}, \mathbf{m_i}, \mathsf{pk}_i)$ $(i \in [2, n-1])$
  - $\pi_n \leftarrow \mathsf{Prove}(F_n, \mathbf{x}_\mathsf{pub}, \mathbf{m_{n-1}}, \mathbf{y}_\mathsf{pub} \cap \mathbf{h'_{n-1}}, \mathbf{y}_\mathsf{priv}, \mathsf{pk}_n)$

  and then outputs a proof $\pi_\mathsf{MS} = (\pi_{i\in[1,n]})$.
- $\mathsf{Verify}_\mathsf{MS}(\mathbf{x}_\mathsf{pub}, \mathbf{h}_{i\in[1,n]}, \pi_\mathsf{MS}, \mathsf{vk}_\mathsf{MS}) \rightarrow \{0, 1\}$.
  Given public variables $\mathbf{x}_\mathsf{pub}$, $\mathbf{h}_{i\in[1,n]}$, a proof π_MS, and a verification key $\mathsf{vk}_\mathsf{MS} = (\mathsf{vk}_{i\in[1,n]})$, a verifier runs:

- $v_1 \leftarrow \mathsf{Prove}(F_1, \mathbf{x_{pub}}, \mathbf{h_1}, \pi_1, \mathsf{vk}_1)$
- $v_i \leftarrow \mathsf{Prove}(F_i, \mathbf{x_{pub}}, \mathbf{h'_{i-1}} \cap \mathbf{h_i}, \pi_i, \mathsf{vk}_i)$ $(i \in [2, n-1])$
- $v_n \leftarrow \mathsf{Prove}(F_n, \mathbf{x_{pub}}, \mathbf{y_{pub}} \cap \mathbf{h'_{n-1}}, \pi_n, \mathsf{vk}_n)$

and then outputs $\prod_{i\in[1,n]} v_i$.

The definition of Good Split is also extended as below.

Definition 3.4: A Good n-Split is an n-Split $S = (F_1, F_2, \ldots, F_n)$ such that $|\mathbf{m}| \ll |F|$ and $max\{|F_1|, |F_2|, \ldots, |F_n|\}$ is relatively small.

C. How the Memory Usage Get Reduced by Split

The number of variables in a circuit determines the memory usage. Therefore, to reduce the memory usage, the number of variables must be significantly reduced. Recall that one of the main differences between a zk-SNARK system and a regular computer is whether intermediate variables are kept or wiped out. Even if an intermediate variable is no longer needed in a later stage, it has to be kept for the whole lifetime in zk-SNARKs.

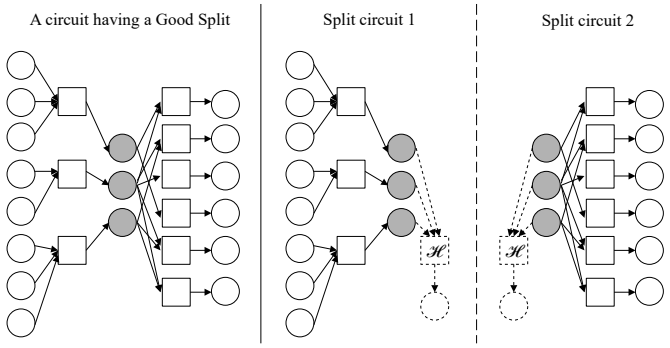


Fig. 5: How Split works in the front-end view

As long as intermediate variables are no longer needed in a later stage, our Split scheme can remove them by partitioning the circuit. Fig. 5 shows an example circuit that is ideal for partition. In this example, the circuit is cut by replicating the shadowed variables to get two components, where some variables only exist in the first component. With this split, the two generated circuits replace the original one by using the common variables as the secret outputs of the first circuit, and also the secret inputs of the second circuit. Provers only need nearly a half memory to generate the proof by applying the zero-knowledge protocol to these two circuits sequentially.

When processing the second circuit, the input values should be the output values of the first circuit. However, remember that a prover is able to choose arbitrary input values, a malicious prover may inject other input values into the second circuit so that the final output changes. This breaks the security properties of zk-SNARKs. To prevent it from happening, a hash algorithm should be involved in both circuits and the digest of the common variables is considered a public output. A malicious prover can not alter the input values of the second circuit anymore, as he/she can't find another group of input values that produce the same digest.

Involving a hash circuit causes overhead, thus the hash algorithm needs to be optimized for zk-SNARKs which requires as few constraints and variables as possible. Additionally, our Split construction is suitable for circuits that can be partitioned with as few common variables as possible. Generally, loops are ideal structures for a split, as a lot of intermediate variables are generated inside an iteration, and these variables become obsolete as long as an iteration ends.

D. Examples

Here, we present two examples to demonstrate the split of a circuit in a straightforward way. Fig. 6 presents a zero-knowledge version of the Fibonacci series algorithm, in which a prover persuades a verifier he/she knows a secret value x (no more than the constant n) that is corresponding to the given Fibonacci number v at index x, without leaking the value x. Recall that statements like loop are unrolled by the front-end compiler, as mentioned in Sec. II-B2. Therefore, the size of the circuit implementing Fig. 6 is almost linear to the constant number n, which causes a high memory occupation. The number of constraints of the circuit is no less than $4n$.[7] There is a Good Split for Fibonacci Series, dividing the circuit into two components, shown in Fig. 7 and Fig. 8, respectively. The split is applied in the middle of the for-loop, saving all the variables that may be used later as intermediate ones $\mathbf{m}$ and its hash value $\mathbf{h}$. A malicious prover can not alter the output value v, as otherwise, the output $\mathbf{h}$ of Fig. 8 would change. Obviously, this is a Good Split, as it follows the definition in Sec. III-A:

- $|\mathbf{m}| = 4 \ll |F| \approx 4n$
- $max\{|F_1|, |F_2|\} \approx 2n + |F_{\mathcal{H}}|$, which is almost the minimum possible value, where $|F_{\mathcal{H}}|$ stands for the additional overhead of the hash algorithm.

and it saves almost 50% of memory space with little extra overhead. Similarly, an n-Split can be applied to save more memory spaces, but with more overhead.

```
constant      : n: The maximum value of the input x.
secret input  : x: An index of Fibonacci series.
public output: v: The Fibonacci number at index x.
a ← 0;
b ← 1;
for i ← 1 to n do
    c ← a + b;
    a ← b;
    b ← c;
    if i = x then
        v ← a;
    end
end
```

Fig. 6: Example Code of Fibonacci Series (Suitable for a Good Split)

[7]For simplification, we count each statement as a constraint, and ignore the unrolling for if-statement that the zk-SNARK compilers actually do. Readers are only required to understand what the loop unrolling is, and may ignore other unrolling details here.

```
constant      : n: The maximum value of the input x.
secret input  : x: An index of Fibonacci series.
secret output : m: Intermediate values that may be
                used later.
public output : h: The hash value of m.
a ← 0;
b ← 1;
for i ← 1 to ⌊n/2⌋ do
    c ← a + b;
    a ← b;
    b ← c;
    if i = x then
        v ← a;
    end
    if i = ⌊n/2⌋ then
        // Unlike others, this if-statement is
           free of charge with loop unrolling.
        m ← (a, b, c) ; // Save variables.
        h ← H(m) ; // Get the hash value.
    end
end
```

Fig. 7: Example Code of Fibonacci Series (Split - Component 1)

```
constant      : n: The maximum value of the input x.
secret input  : m: Intermediate values.
public output : h: The hash value of m.
public output : v: The Fibonacci number at index x.
h ← H(m) ;
(a, b, c) ← m ; // Load saved variables.
for i ← ⌊n/2⌋ + 1 to n do
    c ← a + b;
    a ← b;
    b ← c;
    if i = x then
        v ← a;
    end
end
```

Fig. 8: Example Code of Fibonacci Series (Split - Component 2)

Fig. 9 demonstrates another example in which a prover persuades a verifier he/she knows a secret sequence a that has the longest increasing sub-sequence of length l, without revealing a. Although it seems that all previous variables might be used in the future, the real case is that it also has Good Splits. In each iteration, new variables are generated as long as they have a chance to be updated. So in Fig. 9, although $f_i \leftarrow f_j + 1$ occurs when the condition $a_j \leq a_i \wedge f_i < f_j$ holds, f_i is still updated even if the above condition does not hold. Recall the unrolling details, where the compiler unrolls the if-statement into an assignment, the value of a variable is updated by both the true condition and the false condition, and thus obsolete variables are generated for such an update. Same with Fig. 6, there are Good Splits by partitioning between every two iterations, and therefore the result is omitted due to the space limitation.

```
constant      : Length n of the input sequence.
secret input  : A sequence a of a fixed length n.
public output : The length l of the longest increasing
                sub-sequence.
l ← 0;
for i ← 1 to n do
    f_i ← 1;
    for j ← 1 to i do
        if a_j ≤ a_i and f_i < f_j then
            f_i ← f_j + 1;
        end
    end
    if l < f_i then
        l ← f_i;
    end
end
```

Fig. 9: Example Code of the Longest Increasing Subsequence problem (Suitable for a Good Split)

IV. Analysis

Readers may wonder whether our Split construction breaks the security properties of zk-SNARKs introduced in Sec. II-B1, how exactly our construction saves the memory usage, and whether the performance in CPU times is decreased. In this section, we first carry out a security analysis showing that all three security properties are preserved, then perform a performance analysis indicating how much the memory gets reduced and how much the computational overhead gets involved.

A. Security Analysis

Theorem 4.1: Assume that the underlying zk-SNARK scheme satisfies completeness, soundness, zero-knowledge, and that the hash function satisfies the second preimage security and preimage security, then our Split construction is a zero-knowledge proof system satisfying completeness, soundness, and zero-knowledge.

To prove Theorem 4.1, we first introduce the definition of the three properties of zk-SNARKs [20] as Def. 3.1, and prove Lemma 4.1 under the hash assumption presented in Theorem 4.1.

Definition 4.1: A zk-SNARK scheme has the following properties:

- *Completeness*. For all $\lambda \in \mathbb{N}$ and for all $(F, \mathbf{x}_{\mathsf{pub}}, \mathbf{x}_{\mathsf{priv}}, \mathbf{y}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{priv}}) \in \mathcal{L}$:

$$\Pr\left[\mathsf{Verify}(\mathbf{x}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{pub}}, \pi, \mathsf{vk}) = 1 \;\middle|\; \begin{array}{l} (\mathsf{pk}, \mathsf{vk}, \mathsf{trap}) \leftarrow \mathsf{Setup}(1^{\mathcal{K}}, F) \\ \pi \leftarrow \mathsf{Prove}(F, \mathbf{x}_{\mathsf{pub}}, \mathbf{x}_{\mathsf{priv}}, \mathbf{y}_{\mathsf{pub}}, \mathbf{y}_{\mathsf{priv}}, \mathsf{pk}) \end{array}\right] = 1$$

- *Soundness*. For all PPT adversaries $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}_{\mathcal{A}}$ such that:

$$\Pr\left[\begin{array}{l|l}\mathsf{Verify}(\mathbf{x}_{\mathsf{pub}},\mathbf{y}_{\mathsf{pub}},\pi,\mathsf{vk})=1 & \\ (F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\notin\mathcal{L} & \\ (\mathsf{pk},\mathsf{vk},\mathsf{trap})\leftarrow\mathsf{Setup}(1^{\mathcal{K}},F) & \\ (\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}},\pi)\leftarrow\mathcal{A}||\mathcal{E}_{\mathcal{A}}(\mathsf{pk},\mathsf{vk}) & \end{array}\right]\leq\mathsf{negl}(\lambda)$$

- *Zero-knowledge*. For all $\lambda\in\mathbb{N}$, for all $(F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\in\mathcal{L}$ and for all PPT adversaries $\mathcal{A}$, the following two distributions are statistically close:

$$D_0=\{\pi_0\leftarrow\mathsf{Prove}(F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}},\mathsf{pk}):(\mathsf{pk},\mathsf{vk},\mathsf{trap})\leftarrow\mathsf{Setup}(1^{\mathcal{K}},F)\}$$

$$D_1=\{\pi_1\leftarrow\mathsf{Simulate}(F,\mathbf{x}_{\mathsf{pub}},\mathbf{y}_{\mathsf{pub}},\mathsf{pk},\mathsf{trap}):(\mathsf{pk},\mathsf{vk},\mathsf{trap})\leftarrow\mathsf{Setup}(1^{\mathcal{K}},F)\}$$

Lemma 4.1: Under the assumption in Theorem 4.1, a Split (F_1,F_2) of the original circuit F satisfies

$$\begin{aligned}&(F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\in\mathcal{L}\\&\iff\exists(\mathbf{h},\mathbf{m},\mathbf{h}',\mathbf{m}'),(\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{h},\mathbf{m},\mathsf{pk}_1)\in\mathcal{L}_1\\&\wedge(\mathbf{x}_{\mathsf{pub}},\mathbf{m}',\mathbf{y}_{\mathsf{pub}}\cap\mathbf{h}',\mathbf{y}_{\mathsf{priv}},\mathsf{pk}_2)\in\mathcal{L}_2\\&\wedge\mathbf{h}=\mathbf{h}'\end{aligned}\tag{1}$$

Proof of Lemma 4.1: Because the hash function satisfies the second preimage security, a prover is unable to find a $\mathbf{m}'\neq\mathbf{m}$ such that $\mathbf{h}=\mathcal{H}(\mathbf{m}')$, which means $\Pr[\mathbf{m}=\mathbf{m}'|\mathbf{h}=\mathbf{h}']=1$. Thus, if $\exists(\mathbf{h},\mathbf{m},\mathbf{h}',\mathbf{m}'),(\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{h},\mathbf{m},\mathsf{pk}_1)\in\mathcal{L}_1\wedge(\mathbf{x}_{\mathsf{pub}},\mathbf{m}',\mathbf{y}_{\mathsf{pub}}\cap\mathbf{h}',\mathbf{y}_{\mathsf{priv}},\mathsf{pk}_2)\in\mathcal{L}_2\wedge\mathbf{h}=\mathbf{h}'$ holds, because $\mathbf{h}=\mathbf{h}'$, we have $\mathbf{m}=\mathbf{m}'$, hence $\mathbf{h},\mathbf{m}=F_1(\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}})$ and $\mathbf{y}_{\mathsf{pub}}\cap\mathbf{h},\mathbf{y}_{\mathsf{priv}}=F_2(\mathbf{x}_{\mathsf{pub}},\mathbf{m})$, then by the definition of zero-knowledge procedure of Split, we know $\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}}=F(\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}})$ holds, i.e. $(F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\in\mathcal{L}$. On the other hand, if $(F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\in\mathcal{L}$ holds, i.e. $\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}}=F(\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}})$ holds, let $\mathbf{h}',\mathbf{m}'=\mathbf{h},\mathbf{m}=F_1(\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}})$, we have $\mathbf{y}_{\mathsf{pub}}\cap\mathbf{h},\mathbf{y}_{\mathsf{priv}}=F_2(\mathbf{x}_{\mathsf{pub}},\mathbf{m})$, which is guaranteed by the definition of Split. Therefore, $(\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{h},\mathbf{m},\mathsf{pk}_1)\in\mathcal{L}_1$ and $(\mathbf{x}_{\mathsf{pub}},\mathbf{m}',\mathbf{y}_{\mathsf{pub}}\cap\mathbf{h}',\mathbf{y}_{\mathsf{priv}},\mathsf{pk}_2)\in\mathcal{L}_2$ hold while $\mathbf{h}=\mathbf{h}'$. ■

Proof of Theorem 4.1: A Split (F_1,F_2) of the original circuit F has:

- *Completeness*. For all $\lambda\in\mathbb{N}$ and for all $(F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\in\mathcal{L}$, from Lemma 4.1 we have $(\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{h},\mathbf{m},\mathsf{pk}_1)\in\mathcal{L}_1$ and $(\mathbf{x}_{\mathsf{pub}},\mathbf{m}',\mathbf{y}_{\mathsf{pub}}\cap\mathbf{h}',\mathbf{y}_{\mathsf{priv}},\mathsf{pk}_2)\in\mathcal{L}_2$ holds. Hence,

$$P_1=\Pr\left[\mathsf{Verify}(\mathbf{x}_{\mathsf{pub}},\mathbf{h},\pi_1,\mathsf{vk})=1\;\middle|\;\begin{array}{l}(\mathsf{pk}_1,\mathsf{vk}_1,\mathsf{trap})\leftarrow\mathsf{Setup}(1^{\mathcal{K}},F_1)\\ \pi_1\leftarrow\mathsf{Prove}(F_1,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{h},\mathbf{m},\mathsf{pk}_1)\end{array}\right]=1$$

$$P_2=\Pr\left[\mathsf{Verify}(\mathbf{x}_{\mathsf{pub}},\mathbf{y}_{\mathsf{pub}}\cap\mathbf{h}',\pi_2,\mathsf{vk}_2)=1\;\middle|\;\begin{array}{l}(\mathsf{pk}_2,\mathsf{vk}_2,\mathsf{trap})\leftarrow\mathsf{Setup}(1^{\mathcal{K}},F_2)\\ \pi_2\leftarrow\mathsf{Prove}(F_2,\mathbf{x}_{\mathsf{pub}},\mathbf{m},\mathbf{y}_{\mathsf{pub}}\cap\mathbf{h}',\mathbf{y}_{\mathsf{priv}},\mathsf{pk}_2)\end{array}\right]=1$$

From the zero-knowledge proof procedures of Split we know $\mathsf{Verify}_{\mathsf{S}}(\mathbf{x}_{\mathsf{pub}},\mathbf{y}_{\mathsf{pub}},\mathbf{h},\pi_{\mathsf{S}},\mathsf{vk}_{\mathsf{S}})=\mathsf{Verify}(\mathbf{x}_{\mathsf{pub}},\mathbf{h},\pi_1,\mathsf{vk}_1)\cdot\mathsf{Verify}(\mathbf{x}_{\mathsf{pub}},\mathbf{y}_{\mathsf{pub}}\cap\mathbf{h},\pi_2,\mathsf{vk}_2)$. It follows that

$$\begin{aligned}&\Pr\left[\mathsf{Verify}_{\mathsf{S}}(\mathbf{x}_{\mathsf{pub}},\mathbf{y}_{\mathsf{pub}},\mathbf{h},\pi_{\mathsf{S}},\mathsf{vk}_{\mathsf{S}})=1\;\middle|\;\begin{array}{l}(\mathsf{pk}_{\mathsf{S}},\mathsf{vk}_{\mathsf{S}},\mathsf{trap})\leftarrow\mathsf{Setup}_{\mathsf{S}}(1^{\mathcal{K}},F)\\ \pi\leftarrow\mathsf{Prove}_{\mathsf{S}}(S,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}},\mathsf{pk}_{\mathsf{S}})\end{array}\right]\\&=P_1\cdot P_2\cdot\Pr[\mathbf{m}=\mathbf{m}'|\mathbf{h}=\mathbf{h}']=1\end{aligned}$$

Hence the completeness is satisfied.

- *Soundness*. For all PPT adversaries $\mathcal{A}$, denote $\mathcal{E}_{\mathcal{A}}$ as a PPT extractor. From the soundness of F_1 and F_2 we have

$$\Pr\left[\begin{array}{l|l}\mathsf{Verify}(\mathbf{x}_{\mathsf{pub}},\mathbf{h},\pi_1,\mathsf{vk}_1)=1 & \\ (F_1,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{h},\mathbf{m})\notin\mathcal{L} & \\ (\mathsf{pk}_1,\mathsf{vk}_1,\mathsf{trap})\leftarrow\mathsf{Setup}(1^{\mathcal{K}},F_1) & \\ (\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{h},\mathbf{m},\pi_1)\leftarrow\mathcal{A}||\mathcal{E}_{\mathcal{A}}(\mathsf{pk}_1,\mathsf{vk}_1) & \end{array}\right]\leq\mathsf{negl}(\lambda)$$

$$\Pr\left[\begin{array}{l|l}\mathsf{Verify}(\mathbf{x}_{\mathsf{pub}},\mathbf{h}\cap\mathbf{y}_{\mathsf{pub}},\pi_2,\mathsf{vk}_2)=1 & \\ (F_2,\mathbf{x}_{\mathsf{pub}},\mathbf{m}',\mathbf{h}'\cap\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\notin\mathcal{L} & \\ (\mathsf{pk}_2,\mathsf{vk}_2,\mathsf{trap})\leftarrow\mathsf{Setup}(1^{\mathcal{K}},F_2) & \\ (\mathbf{x}_{\mathsf{pub}},\mathbf{m}',\mathbf{h}'\cap\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}},\pi_2)\leftarrow\mathcal{A}||\mathcal{E}_{\mathcal{A}}(\mathsf{pk}_2,\mathsf{vk}_2) & \end{array}\right]\leq\mathsf{negl}(\lambda)$$

Thus, to get $\mathsf{Verify}_{\mathsf{S}}=1|(F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\notin\mathcal{L}$, adversary $\mathcal{A}$ can only select $(\mathbf{h},\mathbf{m},\mathbf{h}',\mathbf{m}',\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})$ such that:
  - $(F_1,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{h},\mathbf{m})\in\mathcal{L}$
  - $(F_2,\mathbf{x}_{\mathsf{pub}},\mathbf{m}',\mathbf{h}'\cap\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\in\mathcal{L}$
  - $\mathbf{h}=\mathbf{h}'$

From Lemma 4.1 immediately we know the three conditions above hold if and only if $(F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\in\mathcal{L}$. A contradiction is derived. Hence,

$$\Pr\left[\begin{array}{l|l}\mathsf{Verify}_{\mathsf{S}}(\mathbf{x}_{\mathsf{pub}},\mathbf{y}_{\mathsf{pub}},\mathbf{h},\pi_{\mathsf{S}},\mathsf{vk}_{\mathsf{S}})=1 & \\ (F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\notin\mathcal{L} & \\ (\mathsf{pk}_{\mathsf{S}},\mathsf{vk}_{\mathsf{S}},\mathsf{trap})\leftarrow\mathsf{Setup}_{\mathsf{S}}(1^{\mathcal{K}},F) & \\ \pi\leftarrow\mathcal{A}||\mathcal{E}_{\mathcal{A}}(\mathsf{pk}_{\mathsf{S}},\mathsf{vk}_{\mathsf{S}}) & \end{array}\right]\leq\mathsf{negl}(\lambda)$$

Therefore the soundness is satisfied.

- *Zero-knowledge*. Because the hash function satisfies preimage security, the adversary is unable to learn anything of $\mathbf{m}$ from the hash value $\mathbf{h}$. Therefore, for all $\lambda\in\mathbb{N}$, for all $(F,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}})\in\mathcal{L}$ and for all PPT adversaries $\mathcal{A}$, the following two distributions are statistically close:

$$D_0=\{\pi_{\mathsf{S}_0}\leftarrow\mathsf{Prove}_{\mathsf{S}}(S,\mathbf{x}_{\mathsf{pub}},\mathbf{x}_{\mathsf{priv}},\mathbf{y}_{\mathsf{pub}},\mathbf{y}_{\mathsf{priv}},\mathsf{pk}_{\mathsf{S}}):(\mathsf{pk}_{\mathsf{S}},\mathsf{vk}_{\mathsf{S}},\mathsf{trap})\leftarrow\mathsf{Setup}_{\mathsf{S}}(1^{\mathcal{K}},S)\}$$

$$D_1=\{\pi_{\mathsf{S}_1}\leftarrow\mathsf{Simulate}(S,\mathbf{x}_{\mathsf{pub}},\mathbf{y}_{\mathsf{pub}},\mathbf{h},\mathsf{pk}_{\mathsf{S}},\mathsf{trap}):(\mathsf{pk}_{\mathsf{S}},\mathsf{vk}_{\mathsf{S}},\mathsf{trap})\leftarrow\mathsf{Setup}_{\mathsf{S}}(1^{\mathcal{K}},S)\}$$

Hence the zero-knowledge property is satisfied. ■

B. Performance Analysis

1) Computational Overhead:

Claim 4.1: Denote by m_{bits} the bit size of intermediate variables for a Split S, H_{bits} the input bit size of a hash algorithm, and H_{overhead} the overhead of a hash circuit, then the computational overhead can be roughly expressed as $2\cdot\lceil\frac{m_{\mathsf{bits}}}{H_{\mathsf{bits}}}\rceil\cdot H_{\mathsf{overhead}}$ for a Split, and $\sum_{i=1}^{i=n-1}2\cdot\lceil\frac{m_{\mathsf{bits},i}}{H_{\mathsf{bits}}}\rceil\cdot H_{\mathsf{overhead}}$ for an n-Split.

For QAP-based zk-SNARKs, the time complexities of Setup, Prove, and Verify [33] are respectively $O(M)$,

$O(M \log M)$, and $O(1)$, where M stands for the number of R1CS constraints, and the total time cost of running multiple zk-SNARK instances is the sum of them. A Split (F_1, F_2) increases the time cost as a hash algorithm is involved in components. Take SHA256 as an example. The SHA256 procedure adds 34,324 constraints per 128-bit input[8] to the circuit. If $\mathbf{m}$ is more than 512 bits, more hash circuits are introduced, causing more computational overhead. The overhead can be roughly expressed as $\lceil \frac{m_{\mathsf{bits}}}{H_{\mathsf{bits}}} \rceil \cdot H_{\mathsf{overhead}}$ for the calculation of $\mathbf{h} \leftarrow \mathcal{H}(\mathbf{m})$. Since a Split involves two hash procedures for the two split circuits F_1 and F_2, the total overhead can be expressed as $2 \cdot \lceil \frac{m_{\mathsf{bits}}}{H_{\mathsf{bits}}} \rceil \cdot H_{\mathsf{overhead}}$, Therefore, Sec. III-A indicates that a Good Split should have $|\mathbf{m}| \ll |F|$, which means the variable size of $|\mathbf{m}|$ should be much less than the number of constraints of the original circuit so that the computational overhead is acceptable. Similarly, for an n-Split, each collection of intermediate variables $\mathbf{m}_i (i \in [1, n-1])$ involves two hash circuits, resulting in an overhead of $2 \cdot \lceil \frac{m_{\mathsf{bits},i}}{H_{\mathsf{bits}}} \rceil \cdot H_{\mathsf{overhead}}$, thus the total overhead of an n-Split is expressed as $\sum_{i=1}^{i=n-1} 2 \cdot \lceil \frac{m_{\mathsf{bits},i}}{H_{\mathsf{bits}}} \rceil \cdot H_{\mathsf{overhead}}$.

2) Memory Usage:

Claim 4.2: The memory usage can be reduced to $\max\{|F_1|, |F_2|\}$ for a Split, and $\max\{|F_i|\}(i \in [1, n])$ for an n-Split.

The memory usages in the Setup and the Prove procedure are roughly linear to the number of the R1CS constraints, and the memory usage of running multiple zk-SNARK instances is the maximum value among them. Therefore, $\max\{|F_1|, |F_2|\}$ stands for the memory occupation of a Split (F_1, F_2). Thus, the circuit should be split as "evenly" as possible (while keeping $|\mathbf{m}|$ relatively small), so that the memory usage can be reduced to nearly half. Similarly, $\max\{|F_i|\}(i \in [1, n])$ stands for the memory occupation of an n-Split $(F_{i \in [1,n]})$, which can further reduce the memory, but with more overhead. One should carefully consider the trade-off between the computational overhead and the memory occupation.

V. Evaluation

In this section, we set up a series of experiments on a server, illustrating the performance of Split in CPU time and memory usage, to demonstrate how our construction behaves in practice. We analyze the experimental results and explain the effect of the QAP instance reduction on these results.

A. Experiment Setup

To evaluate our scheme and demonstrate the improvement in memory occupation, we design a series of experiments in a server with dual Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz CPU[9] and 128 GB RAM consisting of eight Samsung 16 GB 2133 MT/s DDR4 RDIMM. The operating system is Ubuntu 20.04.2 LTS.

[8]The number of constraints is obtained from the SHA256 circuit provided by xJsnark [29].

[9]In our experiments, only a single core is occupied for running the zk-SNARK toolchain. The use of dual processors is to support more RAM modules.

A QAP-based zk-SNARK system [14] is used in our proof-of-concept demonstration. Considering the fact that loop is the most common structure in zk-SNARKs and also an ideal structure for a split, and an R1CS instance consists of addition, subtraction, and multiplication constraints, we make use of a loop updating $x \leftarrow a \cdot x + b$ with different number of iterations as the original circuit, and select the SHA256 algorithm to implement a hash circuit that ensures the integrity of intermediate variables. We partition each circuit into components by splitting the iterations.

Our program code for the circuit is written in a domain-specific language, xJsnark [29]. The circuit is then compiled into an R1CS instance, which is then reduced to a QAP instance by libsnark. Finally, Setup, Proof, and Verify procedures are executed.

In our experiments, we run the complete procedure of zero-knowledge proof for both the original circuit and the split circuits, where the following metrics are considered:

- The number of constraints of the R1CS instance, which indicates the circuit size.
- The QAP degree of the circuit, which also indicates the circuit size.
- The time cost of Proof.
- The memory usage of Proof.
- The time cost of Verify, which is almost constant.

B. Experiments and Results

1) Experiment I: In Experiment I, we adjust the number of loops and generate several circuits, of which the number of constraints varies between approximately 380,000 to 11,500,000. We split the original circuit into two components, and observe the relationship between circuit size and the memory usage as well as the time cost between the original one and the split ones. Each test case is repeated thirty times and the averages are reported. The results of Proof are illustrated in Fig. 10a and Fig. 10b, and the statistical data is listed in Table II. Note that error bars in the figures are not visible, as the memory usage error is less than 1 MB, while the time cost error is about 0.5 s.

Obviously, our Split construction is more efficient for a large-sized circuit. As the size grows, the reduced memory occupation gets closer to almost 50%. For the time cost, things become complex. In theory, the overhead introduced by the hash algorithm should cause a higher time cost. But in some test cases of the experimental results, the time cost is actually decreased a little. This is due to the implementation of QAP instance reduction, which is explained in Sec. V-C.

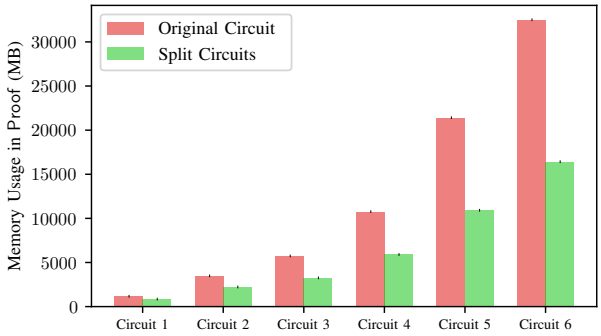
Fig. 11 shows the time cost of Verify. It gets doubled after Split, which is as expected. As the Verify procedure takes in milliseconds, it is acceptable.

2) Experiment II: As demonstrated earlier, a Good Split can reduce the memory usage by up to 50%. An n-Split may further reduce the memory usage. We design Experiment II, where Circuit 4 in Experiment I is split into 2, 3, 4, and 5 components.

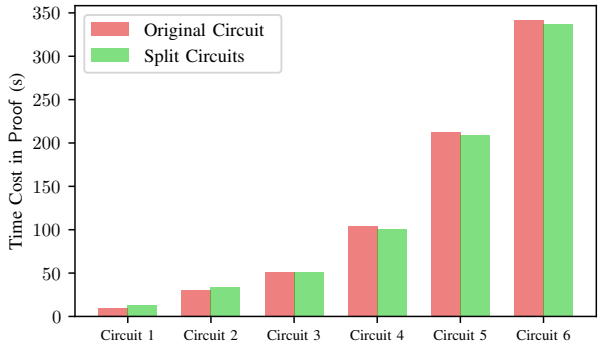
The results are reported in Fig. 12 and Table III. Similar to Experiment I, error bars in Fig. 12 are not visible. The

TABLE II: Performance before and after Split of different circuit sizes

Case	Original Circuit				Split Circuits			
	Constraints	Degree	Time (s)	Memory (MB)	Constraints	Degree	Time (s)	Memory (MB)
Circuit 1	383,381	393,216	10.0336	1,173	501,306	524,288	12.7547	851
Circuit 2	1,150,037	1,179,648	30.8620	3,486	1,267,930	1,310,720	33.7338	2,221
Circuit 3	1,916,725	2,097,152	50.6133	5,742	2,034,618	2,097,152	50.5976	3,258
Circuit 4	3,833,381	4,194,304	104.3493	10,763	3,951,306	4,194,304	101.2589	5,901
Circuit 5	7,666,725	8,388,608	212.2286	21,413	3,892,309	8,388,608	208.8059	10,918
Circuit 6	11,500,037	1,258,2912	341.2828	32,513	11,617,930	12,582,912	336.5552	16,421



(a) The memory usage in Proof before and after Split



(b) The CPU time cost in Proof before and after Split

Fig. 10: The comparison of memory usage and CPU time cost before and after Split

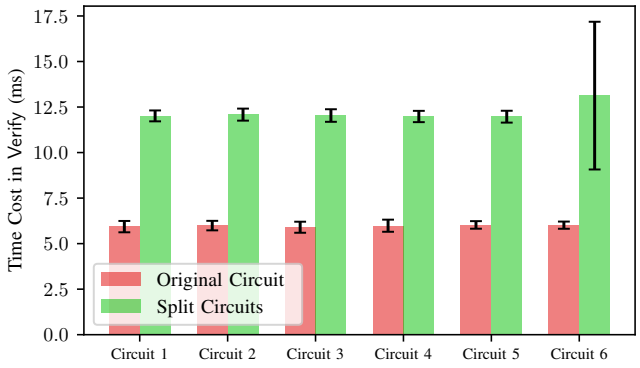


Fig. 11: The time cost in Verify before and after Split

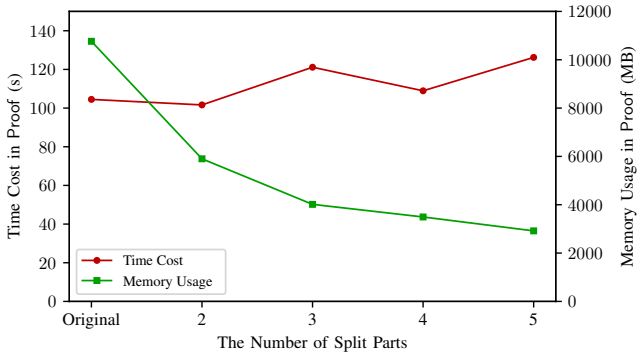


Fig. 12: The memory usage and time cost in Proof of n-Split circuits

TABLE III: Performance of n-Split circuits

n-Split	Constraints	Degree	Time (s)	Memory (MB)
Original	3,383,381	4,194,304	104.4845	10,763
2	3,951,306	4,194,304	101.6369	5,901
3	4,069,199	4,718,592	121.1280	4,016
4	4,187,092	4,259,840	108.9389	3,494
5	4,305,049	5,242,880	126.2172	2,919

data indicates that as the number of split components grows, the memory usage is reduced further, but the reduced rate gets slower. On the other hand, the computational overhead continues to grow, which makes n-Split less efficient, but still acceptable. Also similar to Experiment I, the time cost of 3-Split is abnormal due to the same reason, which is explained in the next subsection.

Fig. 13 illustrates the time cost of Verify. It is linear to the number of split components. Thus, the split number should be decided after careful considerations, to trade-off between the memory occupation and the computational overhead.

From the experimental results, one can conclude that splitting into 5 components optimizes the memory usage to 27%, while having only 20% additional CPU time for the demonstrated circuit.

C. QAP Instance Reduction

As mentioned in Sec. II-B6, an R1CS instance is converted into a QAP instance by QAP instance reduction. As FFT is used to efficiently multiply polynomials, the QAP degree

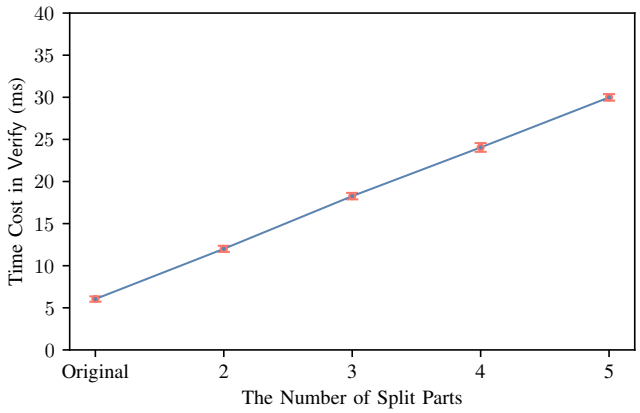


Fig. 13: The time cost in Verify of n-Split circuits

might be larger to meet the need of FFT. A radix-2 FFT implementation has time complexity $O(N\log(N))$, which requires the domain size to be of the form 2^k for a positive integer k. In libsnark, an advanced version of radix-2 FFT named *step radix-2 FFT*[10], relaxes the domain size requirement to be of the form 2^k+2^r, where r is also a positive integer. Therefore, for a QAP instance of degree M, libsnark pads the degree to $2^{\lfloor\log_2 M\rfloor-1}+2^{\lfloor\log_2(M-2^{\lfloor\log_2 M\rfloor-1})\rfloor}$ which is the minimum integer that is no smaller than M and can be represented with the form of 2^k+2^r. As the radix-2 FFT algorithm brings a significant optimization for time cost, the padding scheme causes a necessary overhead.

Back to our Split construction. Denote by M_{R1CS}, $M^{(1)}_{\mathsf{R1CS}}$, $M^{(2)}_{\mathsf{R1CS}}$ the number of R1CS constraints of the original circuit F and the split circuits F_1, F_2, respectively, and denote by M_{QAP}, $M^{(1)}_{\mathsf{QAP}}$, $M^{(2)}_{\mathsf{QAP}}$ similarly for the QAP degrees. As $M^{(1)}_{\mathsf{R1CS}}+M^{(2)}_{\mathsf{R1CS}}=M_{\mathsf{R1CS}}+M^{\mathsf{overhead}}_{\mathsf{R1CS}}$, we have $M^{(1)}_{\mathsf{R1CS}}+M^{(2)}_{\mathsf{R1CS}}>M_{\mathsf{R1CS}}$. In theory, this means that the time cost of the split circuits is greater than the original. But due to the padding above, we have $M^{(1)}_{\mathsf{R1CS}}+M^{(2)}_{\mathsf{R1CS}}>M_{\mathsf{R1CS}} \not\Rightarrow M^{(1)}_{\mathsf{QAP}}+M^{(2)}_{\mathsf{QAP}}>M_{\mathsf{QAP}}$ which brings out the "abnormal" results in time cost. Moreover, it is obvious that, when M_{R1CS} is larger, the chance of having $M^{(1)}_{\mathsf{QAP}}+M^{(2)}_{\mathsf{QAP}}\leq M_{\mathsf{QAP}}$ is also larger.

VI. Conclusion and Future Research

We propose a hash-based memory optimization method for zk-SNARKs that can greatly reduce the memory occupation for resource-constrained provers. The high memory consumption is a burden to make zero-knowledge into practice when users do not have sufficient memory for a large-scale zero-knowledge proof. In our scheme, we split the circuit into components, and ensure the integrity of intermediate variables by introducing the hash circuit. We show that our construction does not compromise the three security properties of zk-SNARKs. The experimental results indicate that Good Split is memory-efficient and the computational overhead is low, making zero-knowledge proofs a bit closer to ordinary users.

[10]See https://github.com/scipr-lab/libfqfft for more details.

In our evaluation, we choose SHA256 as the hash algorithm, as it is widely implemented in popular zk-SNARK toolchains. To take a step forward towards further reducing the overhead in our Split construction, hash algorithms that are optimized for zk-SNARKs can be chosen, as they have smaller circuit sizes. MiMC [34] is a family of block ciphers and hash functions primarily designed for SNARK applications. It explores cryptographic primitives with low multiplicative complexity, motivated by recent advances in practical applications of secure multi-party computation, fully homomorphic encryption, and zero-knowledge proofs. Grassi et al. presented Poseidon [35], a modular framework and concrete instances of cryptographic hash functions, which uses up to 8x fewer constraints per message bit than Pedersen Hash. Once these hash functions get well-implemented in popular toolchains, our Split scheme can be applicable to more algorithms as the overhead can be further reduced.

As long as there is a Good Split for a circuit, the memory occupation can be greatly minimized while the computational overhead can be low. However, splitting the circuit is hard. A Good Split should involve as few intermediate variables as possible, to reduce the number of relatively expensive hash circuits. An approximate algorithm to find a Good Split for arbitrary circuits can enhance the application of Split.

Acknowledgment

This work was supported in part by the National Key Research and Development Program of China under Grant 2020YFB1005900 and National Natural Science Foundation of China (NSFC) under Grant 62232010.

References

[1] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 459–474.

[2] C. Lin, D. He, X. Huang, M. K. Khan, and K.-K. R. Choo, "Dcap: A secure and efficient decentralized conditional anonymous payment system based on blockchain," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2440–2452, 2020.

[3] Z. Guan, Z. Wan, Y. Yang, Y. Zhou, and B. Huang, "Blockmaze: An efficient privacy-preserving account-model blockchain based on zk-snarks," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[4] J. Benet and N. Greco, "Filecoin: A decentralized storage network," *Protoc. Labs*, pp. 1–36, 2018.

[5] T. Miyamae, F. Kozakura, M. Nakamura, S. Zhang, S. Hua, B. Pi, and M. Morinaga, "Zgridbc: Zero-knowledge proof based scalable and private blockchain platform for smart grid," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2021, pp. 1–3.

[6] W. Li, H. Guo, M. Nejad, and C.-C. Shen, "Privacy-preserving traffic management: A blockchain and zero-knowledge proof inspired approach," *IEEE Access*, vol. 8, pp. 181 733–181 743, 2020.

[7] D. Liu, C. Huang, J. Ni, X. Lin, and X. S. Shen, "Blockchain-cloud transparent data marketing: Consortium management and fairness," *IEEE Transactions on Computers*, 2022.

[8] Y. Rahulamathavan, S. Veluru, J. Han, F. Li, M. Rajarajan, and R. Lu, "User collusion avoidance scheme for privacy-preserving decentralized key-policy attribute-based encryption," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2939–2946, 2016.

[9] J. K. Liu, M. H. Au, T. H. Yuen, C. Zuo, J. Wang, A. Sakzad, X. Luo, and L. Li, "Privacy-preserving covid-19 contact tracing app: A zero-knowledge proof approach." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 528, 2020.

[10] C. Liu, H. Guo, M. Xu, S. Wang, D. Yu, J. Yu, and X. Cheng, "Extending on-chain trust to off-chain – trustworthy blockchain data collection using trusted execution environment (tee)," *IEEE Transactions on Computers*, pp. 1–1, 2022.

[11] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "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.

[12] G. Zhang, X. Liu, and Y. Yang, "Time-series pattern based effective noise generation for privacy protection on cloud," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1456–1469, 2015.

[13] W. Zhang, Y. Lin, S. Xiao, J. Wu, and S. Zhou, "Privacy preserving ranked multi-keyword search for multiple data owners in cloud computing," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1566–1577, 2016.

[14] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 781–796.

[15] Z. Ghodsi, T. Gu, and S. Garg, "Safetynets: Verifiable execution of deep neural networks on an untrusted cloud," 2017.

[16] B. Feng, L. Qin, Z. Zhang, Y. Ding, and S. Chu, "Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences," Cryptology ePrint Archive, Report 2021/087, 2021, https://ia.cr/2021/087.

[17] G. Drevon and A. Kampa, "Benchmarking zero-knowledge proofs with isekai," Dec 2019, https://sikoba.com/docs/SKOR_isekai_benchmarking_201912.pdf.

[18] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "DIZK: A distributed zero knowledge proof system," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 675–692. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/wu

[19] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 238–252.

[20] J. Groth, "On the size of pairing-based non-interactive arguments," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2016, pp. 305–326.

[21] M. Xu, C. Liu, Y. Zou, F. Zhao, J. Yu, and X. Cheng, "wchain: A fast fault-tolerant blockchain protocol for multihop wireless networks," *IEEE Transactions on Wireless Communications*, vol. 20, no. 10, pp. 6915–6926, 2021.

[22] E. Ben-Sasson, A. Chiesa, and N. Spooner, "Interactive oracle proofs," in *Theory of Cryptography Conference*. Springer, 2016, pp. 31–60.

[23] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, "Ligero: Lightweight sublinear arguments without a trusted setup," in *Proceedings of the 2017 acm sigsac conference on computer and communications security*, 2017, pp. 2087–2104.

[24] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward, "Aurora: Transparent succinct arguments for r1cs," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2019, pp. 103–128.

[25] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *Annual International Cryptology Conference*. Springer, 2019, pp. 733–764.

[26] L. Ma, Y. Ge, and Y. Zhu, "Tinyzkp: a lightweight authentication scheme based on zero-knowledge proof for wireless body area networks," *Wireless personal communications*, vol. 77, no. 2, pp. 1077–1090, 2014.

[27] N. Khernane, M. Potop-Butucaru, and C. Chaudet, "Banzkp: A secure authentication scheme using zero knowledge proof for wbans," in *2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE, 2016, pp. 307–315.

[28] M. Xu, S. Liu, D. Yu, X. Cheng, S. Guo, and J. Yu, "Cloudchain: A cloud blockchain using shared memory consensus and rdma," *IEEE Transactions on Computers*, pp. 1–1, 2022.

[29] A. Kosba, C. Papamanthou, and E. Shi, "xjsnark: A framework for efficient verifiable computation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 944–961.

[30] J. Eberhardt and S. Tai, "Zokrates-scalable privacy-preserving off-chain computations," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018, pp. 1084–1091.

[31] C. Chin, H. Wu, R. Chu, A. Coglio, E. McCarthy, and E. Smith, "Leo: A programming language for formally verified, zero-knowledge applications," Cryptology ePrint Archive, Report 2021/651, 2021, https://ia.cr/2021/651.

[32] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 626–645.

[33] J. Partala, T. H. Nguyen, and S. Pirttikangas, "Non-interactive zero-knowledge for blockchain: A survey," *IEEE Access*, vol. 8, pp. 227 945–227 961, 2020.

[34] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, "Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity," in *Advances in Cryptology – ASIACRYPT 2016*, J. H. Cheon and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 191–219.

[35] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/grassi