ZAWA: A ZKSNARK WASM Emulator

SINKA GAO, Delphinus Lab., Australia
HONGFEI FU, Shanghai Jiao Tong University, China
HENG ZHANG, Delphinus Lab., Australia
JUNYU ZHANG, Delphinus Lab., Australia
GUOQIANG LI*, Shanghai Jiao Tong University, China

WebAssembly, or WASM for short, is a binary code format for a stack-based virtual machine, first published in 2018 and now becomes a main-steam technology for providing distributed serverless functions. Recently, the demand for privacy and trustless serverless functions has started to grow in cloud, edge, and grid computation, which poses a question for those serverless function providers: what feature they need to add to make WASM runtime more secure so that the application run on top it become trustless to their users. To address this, we leverage the technology ZKSNARK (zero-knowledge Succinct Non-interactive Argument of Knowledge), a powerful proof system that allows efficient verification of the evaluation problem of statements, to give WASM runtime the ability to provide trustless computation service. More precisely, we present ZAWA, a ZKSNARK backed virtual machine that emulates the execution of WASM bytecode and generates zero-knowledge-proofs for the emulation result. The proof generated by the ZAWA virtual machine can then be used to convince an entity, with no leakage of confidential information, that the result of the emulation enforces the semantic specification of WASM.

ACM Reference Format:
Sinka Gao, Hongfei Fu, Heng Zhang, Junyu Zhang, and Guoqiang Li. 2022. ZAWA: A ZKSNARK WASM Emulator. *Proc. ACM Program. Lang.* 1, 1 (November 2022), 22 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

WASM (or WebAssembly) is an open standard binary code format close to assembly. Its initial objective is to provide an alternative to java-script with better performance in the current web ecosystems. Benefiting from its platform independence, front-end flexibility (can be compiled from the majority of languages including C, C++, assembly script, rust, etc.), good isolated runtime and speed that is close to native binary, its usage starts to arise in the distributed cloud and edge computing. Recently it has become a popular binary format for users to run customized functions on AWS Lambda, Open Yurt, AZURE, etc.

As with the technology of WASM runtime for cloud and edge computing shifts, security and privacy [20, 37] issues emerge in scenarios that demand trustless computation [11, 33, 43] and privacy computing [40, 46]. For instance, suppose that there is a voting hub hosted in the cloud to

*Corresponding Author.

Authors' addresses: Sinka Gao, xgao@zoyoe.com, Delphinus Lab., P.O. Box 1212, Sydney, NSW, Australia, 43017-6221; Hongfei Fu, jt002845@sjtu.edu.cn, Shanghai Jiao Tong University, 800 Dongchuan Rd., Shanghai, China, 200240; Heng Zhang, shindar90@gmail.com, Delphinus Lab., P.O. Box 1212, Sydney, NSW, Australia, 43017-6221; Junyu Zhang, junyu92@gmail.com, Delphinus Lab., P.O. Box 1212, Sydney, NSW, Australia, 43017-6221; Guoqiang Li, li.g@sjtu.edu.cn, Shanghai Jiao Tong University, 800 Dongchuan Rd., Shanghai, China, 200240.

2475-1421/2022/11-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

collect votes for proposals. The role of this service is to report the voting results to users while not leaking any information about any voters' choices. In this scenario, we would like a service that not only provides the voting results but also provides proof to convince users that the provided results are calculated with predefined protocols (voting protocols). However, since the service cannot leak voters' personal choices, it should not reveal any information about the voting tickets that are signed by voters, which makes it tricky to provide evidence of the result.

Traditional ways to achieve trustlessness and privacy usually involve invasive modification [1, 15, 17, 28] to the source code of the service running on the cloud and those changes are usually applied in a case-by-case manner. In this work, instead of changing the code itself, we propose a novel approach by implementing ZAWA, which is a WASM virtual machine that not only runs the WASM bytecode but also provides a zero-knowledge proof to convince a verifier that the execution result is trustworthy.

The idea of ZAWA is derived from ZKSNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) [3, 22, 35], which is a combination of SNARG (Succinct non-interactive arguments) and zero-knowledge proof. In general, the adoption of ZKSNARK usually requires implementing a program in arithmetic circuits or circuit-friendly languages (Pinocchio [27, 34], TinyRAM [4], Buffet/Pequin [42], Geppetto [16], xJsnark framework [30], ZoKrates [18]) that forms a barrier for existing programs to leverage the power of it. An alternative way is that instead of applying ZKSNARK on the source code, we apply it on the bytecode level of a virtual machine and implement a ZKSNARK-backed virtual machine (Similar ideas can be found in Risc0 [39] and ZKEVM [2, 41] while the underlying bytecode they support has less flexibility and portability for cloud application). In this work, we take the approach of writing the whole WASM virtual machine in ZKSNARK circuits so that existing WASM applications can benefit from ZKSNARK by simply running on the ZAWA without any modification. Therefore, the cloud service provider can prove to any user that the computation result is computed honestly and no private information is leaked.

The Problem. To implement a ZKSNARK-backed WASM virtual machine, we need to connect the implementation of WASM runtime with the proof system of ZKSNARK. In general, a ZKSNARK system is represented in arithmetic circuits with polynomial constraints. Therefore we need to abstract the full imperative logic of a WASM virtual machine systematically and rewrite it into arithmetic circuits with constraints. Given two outputs, one is generated by emulating the WASM bytecode in WASM runtime that enforces the semantics of WASM specification, and the other satisfies the constraints imposed on the arithmetic circuits. If the circuits we write preserve the semantics, these two outputs must be the same. Hence the proof of the ZKSNARK derived from the circuits also shows that the output is valid as a result of emulating the bytecode in WASM runtime.

Our Contribution. In this paper, we systematically abstract the WASM runtime implementation and rewrite it into arithmetic circuits with constraints. By doing so, we have proposed and implemented the first ZKSNARK WASM virtual machine that supports WASM specification and produce succinct zero-knowledge correctness proofs of the execution result. Moreover, by providing ZAWA, an existing program compiled to WASM can then satisfy (without any modification) the privacy and trustless requirements that have recently emerged in the cloud and edge computing.

Organization of the Paper. After a brief introduction to the basic ideas about how to connect a stateful virtual machine with ZKSNARK in Section 2, we describe the basic building block and ingredients used to construct ZAWA circuits in Section 3 and then present the circuits architecture in Section 4. After the architecture is settled, we discuss the circuits of every category of WASM instructions in Section 5. In addition, in Section 5.4 we discuss foreign instruction expansion which provides a way to extend the virtual machine for better performance and integration. In Section 6, we present the partition and proof batching technique to solve the long execution trace problem,

and then discuss the performance benchmark in Section 7. In the end, we draw our conclusion and pursue the future work in Section 8.

2 OVERVIEW

Throughout the paper, we use the notation $a : A$ to specify a variable of type A, $\mathbf{F}$ to specify a number field, and $\mathbf{F}^n$ to specify a multi-dimensional vector with dimension n. We denote by $A \to B$ the function type from A to B and use $\circ$ for function composition. Moreover, we use $\mathbf{G}[i][j]$ to specify the value of the cell of matrix $\mathbf{G}$ at the i-th row and j-th column.

2.1 WASM Run-Time as a State Machine

We consider the WASM virtual machine as a gigantic program, with the input as a tuple $(\mathbf{I}(\mathbf{C}, \mathbf{H}), \mathbf{E}, \mathbf{IO})$, where $\mathbf{I}$ is a WASM executable image that contains a code image $\mathbf{C}$ and an initial memory $\mathbf{H}$, $\mathbf{E}$ is its entry point, and $\mathbf{IO}$ represents the (**stdin**, **stdout**) firmware. In the serverless setup, the WASM run-time starts with an initial state based on the loaded image $\mathbf{I}$, then jumps to the entry point $\mathbf{E}$ and starts executing the bytecode based on the WASM specification.

Internally the WASM run-time maintains a state $\mathcal{S}$ denoted by a tuple $(iaddr, \mathcal{F}, \mathcal{M}, \mathcal{G}, \mathcal{SP}, \mathbf{I}, \mathbf{IO})$ where $iaddr$ is the current instruction address, $\mathcal{F}$ is the calling frame with a $depth$ field, $\mathcal{M}$ is the memory state, $\mathcal{SP}$ is the stack and $\mathcal{G}$ is the set of global variables. The run-time simulates the semantic of each instruction start at $\mathbf{E}$ until it reaches the exit. The instructions it simulates form an execution trace $[t_0, t_1, t_2, t_3, \cdots]$ and each transition t_i is a function between states that takes an input $s : \mathcal{S}$ and outputs a new state $s' : \mathcal{S}$. For simplicity, we will use the notation of record field to specify a field in state $s : \mathcal{S}$. For example, $s.iaddr$ denotes the current instruction address of state s, $s.\mathbf{IO}.\mathbf{stdin}$ denotes the input of state s, etc. We also use $op(iaddr)$ to denote the opcode (operation code that specifies the operation to be performed) at address $iaddr$ in the code section C of image $\mathbf{I}$.

Based on the above definition, we define the criteria for a list of state transitions to be valid under $(\mathbf{I}(\mathbf{C}, \mathbf{H}), \mathbf{E}, \mathbf{IO})$, as follows.

Definition 2.1 (Valid Execution Trace). Given a WASM machine with input $(\mathbf{I}(\mathbf{C}, \mathbf{H}), \mathbf{E}, \mathbf{IO})$, and s_0 is the initial state with $s_0.iaddr = \mathbf{E}$. A valid execution trace is a list of transition functions t_i such that the following holds:

(1) t_0 matches the semantic of the instruction op at the entry $iaddr_0 = \mathbf{E}$.
(2) For all k, $s_k = t_{k-1} \circ \cdots t_1 \circ t_0(s_0)$, t_k enforces the semantics of $op(s_k.iaddr)$.
(3) If s_e is the last state, then the depth of the calling frame is zero: $s_e.\mathcal{F}.depth = 0$.

We take the output of the final state $s_e.\mathbf{IO}.output$ to be the result of the WASM run-time with input $(\mathbf{I}(\mathbf{C}, \mathbf{H}), \mathbf{E}, \mathbf{IO})$. The $output$ is a valid result if and only if there exists an valid execution sequence $[t_0, t_1, \cdots]$ such that s_e is the last state of t_i under $(\mathbf{I}(\mathbf{C}, \mathbf{H}), \mathbf{E}, \mathbf{IO})$.

2.2 Succinct Proof of a Program

Compared with a standard WASM run-time, ZAWA aims to provide a proof to prove that the output is valid so that it can be used in scenarios which require trustless and privacy computation. Moreover, the verifying algorithm needs to be simple in the sense of complexity to be useful in practical. Before we dive into how to construct such a proving and verifying scheme for the complex WASM run-time, we go through a few basics about how to construct such a scheme for functions.

Suppose that we have a pure function f, a list of parameters $params$ for f, an entity A that calculates $r = f(params)$ and an entity B that would like to know r but does not willing to do the exact computation. A scheme that enables A to prove to B about the correctness of r is of great interests in cryptography design if the complexity for B to verify the proof is negligible comparing

to the complexity to do the actual calculation. If such a scheme is not interactive and the verifying complexity is negligible comparing to the initial complexity of f, then we say it is a SNARK (succinct non-interactive argument of knowledge). If the SNARK proof also leaks no information then it is a ZKSNARK (zero-knowledge succinct non-interactive argument of Knowledge).

Topics about constructing ZKSNARK has been well studied in the literature [8–10, 14, 21–23, 23, 32] where functions f are defined by a computational program $\mathbf{P}$. A common approach for constructing such ZKSNARK is to turn the program $\mathbf{P} : \mathbf{F}^m \longrightarrow \mathbf{F}^r$ into a special form of constraint systems $C_i(x) = 0$ (where $C : \mathbf{F}^n \longrightarrow \mathbf{F}$), such that for any parameters $param : \mathbf{F}^m$ of $\mathbf{P}$, there exists a unique vector of witness $w : \mathbf{F}^{n-m-r}$ and a unique vector of result $r : \mathbf{F}^r$ that satisfy

$$C_i(param_0, param_1, \cdots, w_0, w_1, \cdots, r_0, r1, \cdots) = 0.$$

We call such constraint system arithmetic circuits (see 2.3 for the precise definition). Once the arithmetic circuits are constructed based on $\mathbf{P}$, the problem of proving $\mathbf{P}(params) = r$ becomes the problems of finding witness vector w and prove that the vector $v = (params; w; r)$ satisfies $C(v) = 0$.

Once the problem of constructing ZKSNARK for a program $\mathbf{P}$ is turned into the problem of constructing ZKSNARK for the correspondent constraint system $\mathbf{C}$, various approach can be applied based on the shapes of $\mathbf{C}$. The basic idea to construct ZKSNARK for $\mathbf{C}$ is to turn the proof for the constraint system $\mathbf{C}$ into proofs of polynomial evaluations, that is deriving a list of polynomials p and a list of evaluation pairs (x_i, v_i) such that $\forall i, p_i(x_i) = v_i$ implies $\mathbf{C}$. The technical and implementation details of such transform is not the focus of this paper. We omit the details and give an example to motivate the basic technique behind it. For example, suppose that a constraint system $\mathbf{C}_i(x) = 0$ can be rewritten into the matrix form $\sum_j c_{ij} x_j = 0$ (linear constraints are used here for simplicity). By Lagrange interpolating on each column vector of $\mathbf{C}$ and vector x we get a list of polynomials $\bar{c}_i(X)$ and $\bar{p}(X)$ such that $\bar{c}_i(j) = c_{ij}$ and $\bar{p}(j) = x_j$. Therefore, $\mathbf{C}_i(x) = 0$ is equivalent to the polynomial equation $\sum \bar{c}_i(X)\bar{p}(X) = 0$ when $X = 1, 2, 3 \cdots$. It follows that the proof of $\mathbf{C}$ can be turned into the proof of the polynomial evaluation problem by proving the evaluation of $\sum \bar{c}_i(X)\bar{p}(X) = 0$ at $X = 0, 1, 2 \cdots$.

A powerful tool for constructing ZKSNARK schemes for the statement of polynomial evaluation is PCS (Polynomial Commitment Schemes [6, 7, 29]). In this paper, without specification, we use KZG (Kate, Zaverucha and Goldberg [29]) as our polynomial commitment scheme. Below we will put more efforts on explaining the specific arithmetic circuits we use to describe the semantics of our target program $\mathbf{P}$ which is a WASM virtual machine.

2.3 Arithmetic Circuits

Arithmetic circuits are a crucial building block in the ZKSNARK of a program. Among various arithmetic circuit systems investigated recently [19, 26, 36], we use the Halo2's [25] circuit system for its flexibility in customization and better integration with polynomial lookup which we needed for table lookup and range check.

Based on the arithmetic circuits provided by ZAWA, the Halo2's zero-knowledge proof system generating execution proofs in a ZKSNARK way. The feature of zero-knowledge makes ZAWA extremely useful in scenarios where the prover would like to prove that certain output is calculated from the execution of a particular program image but does not want to leak the data used.

Due to the complicated structure of a full WASM virtual machine, we need to pick a constraint system $\mathbf{C}$ that is rich in expressiveness and fast in proof producing.

A circuit in Halo2 is defined by a tuple $(\mathbf{G}, \mathbf{C})$ where $\mathbf{G}$ is a n column matrix with rows to be filled later and $\mathbf{C}$ is a set of constraints on a row basis. More precisely, suppose that each cell in G

is indexed (relative to a row l) as $\mathbf{G}_{l,c,r} = \mathbf{G}[l+r][c]$, then each constraint $\mathbf{C}_i$ in $\mathbf{C}$ is one of the following form

$$\mathbf{C}_i(l) = \begin{cases} \mathbf{P}\left(\mathbf{G}_{l,c_0,r_0}, \mathbf{G}_{l,c_1,r_1}, \cdots, \mathbf{G}_{l,c_k,r_k}\right) = 0 & \textbf{or} \\ \left(\mathbf{G}_{l,c_0,r_0}, \mathbf{G}_{l,c_1,r_1}, \cdots, \mathbf{G}_{l,c_k,r_k}\right) \in \mathbf{T} \end{cases} \tag{1}$$

where c_k, r_k are constants, $\mathbf{P}$ is a fixed multi-linear polynomial and $\mathbf{T}$ is a table.

Remark 1. *There are two ways to define constraint in Halo2's constraint system* $\mathbf{C}$. *One way is using polynomial equations of cells and the other is using polynomial lookup. Polynomial lookup is a special constraint that can enforce that an expression expr of cells belongs to an existing table* $\mathbf{T}$. *In the rest of the paper, we use the expression* $plookup(\mathbf{T}, expr) = 0$ *to indicate* $expr \in \mathbf{T}$.

In the rest of this paper, we use *cur* as the current row that $\mathbf{C}_i$ is apply on and use the notation $r_i.(cur + r)$ to denote $\mathbf{G}_{cur,i,r}$ to emphasize the column r_i. With this notation, we see that the constraint system $\mathbf{C}$ provides a flexible way for us to define constraint of cells of a row and their siblings. For example, given the following summarize function *sum*

```
function sum(v) {
    for (suc=0,i=0;i<v.length;i++) {
        suc +=v[i];
    }
    return suc;
}
```

The circuit for it can be constructed as in Table 1 and the constraint system enforced on each row is defined in Equation 2.

Table 1. Circuit matrix of sum

s	acc	operand
1	$sum_0 = 0$	v_0
1	$sum_1 = sum_0 + v_0$	v_1
1	$\cdots$	$\cdots$
0	sum_k	nil

$$\mathbf{C}(cur) = \begin{cases} s.(cur) \times (acc.(cur) + operand.(cur) - acc.(cur+1)) = 0 \\ s.(cur) \times (1 - s.(cur)) = 0 \end{cases} \tag{2}$$

Remark 2. *Notice that the first constraint ensures that addition is applied to each row except for the last row, and the second constraint enforces that* s *is either* 1 *or* 0*).*

Motivated by the above example, we present the formal definition of arithmetic circuits as follows.

Definition 2.2 (Arithmetic Circuit). An arithmetic circuit is a matrix with m columns equipped with a constraint system $\mathbf{C}$ that each constraint $\mathbf{C}_i$ of $\mathbf{C}$ is defined as in Equation 1 and for each row *cur* in the matrix $\mathbf{C}(cur)$ holds.

2.4 Connecting ZAWA Virtual Machine with Arithmetic Circuits

Now we are ready to make one step further. Instead of constructing a ZKSNARK scheme for simple programs, we would like to construct a ZKSNARK, for a WASM virtual machine. ZAWA needs to emulate the execution of **I** start with **E** under **IO**.**stdin** to generate **IO**.**stdout** and provide a ZKSNARK proof which proves that **IO**.**stdout** is valid. Just like what needs to be done for simple programs to produce a ZKSNARK, we need to construct a huge arithmetic circuits with carefully designed constraints **C** such that the following two are equivalent:

(1) **IO**.**stdout** is the unique valid output if the execution of **I** start with **E** under **IO**.**stdin** satisfies the WASM specification.
(2) There exists a list of witness s_i such that $\mathbf{C}(\mathbf{I}, \mathbf{E}, \mathbf{IO}, s_0, s_1, \cdots, s_e) = 0$.

We noticed that a valid execution trace will always produce a valid output respecting the WASM specification. So to construct ZKSNARK for WASM virtual machine, it is sufficient to construct an arithmetic circuit **C** of two states before and after an instruction so that the following two are equivalent.

(1) Given $(\mathbf{I}, \mathbf{E}, \mathbf{IO})$ and s_0 is the initial state, $[t_0, t_1, t_2, \cdots]$ is a valid execution trace satisfy Definition 2.1.
(2) Given an execution trace $[t_0, t_1, \cdots]$ of $(\mathbf{I}, \mathbf{E}, \mathbf{IO}, s_0)$ and $s_k = t_{k-1} \circ \cdots t_1 \circ t_0(s_0)$. $\mathbf{C}(\mathbf{I}, \mathbf{E}, \mathbf{IO}, s_k, s_{k+1}) = 0$ implies t_k enforces the semantics of $op(s_k.iaddr)$.

We do not construct such circuit **C** from scratch, we construct it from small building blocks in Section 3 then create the architecture of **C** in Section 4 and present all the details in Section 5.

3 BASIC BUILDING BLOCKS OF ZAWA CIRCUITS

As described in Section 2.4, the arithmetic circuit of execution trace is crucial in constructing SNARKS of WASM virtual machine. In this section we will give a brief of some basic techniques and elementary circuits used to construct our final arithmetic circuits in ZAWA.

3.1 Representing Basic Types in Halo2 Constraint System

Recall that to prove an arithmetic circuit matrix with constraint **C** holds, the Plonkish proof system interpolates each column c_i into polynomials $c_i(x)$ such that $c_i(j) = c_{ij}$ and then uses KCG commitment scheme to prove $C(c_i(x)) = 0$ holds for all $x = 1, 2, 3, \cdots$.

However, to use KZG commitment scheme on polynomial c_i, we require $c_i(x) \in \mathbf{F}$ where **F** is the scalar field of some elliptic curve **C**. Therefore, each $c_i(j)$ is in the scalar field **F** of the elliptic curve **C** in Halo2's arithmetic circuit system. Since the basic types in WASM are i64 and i32 which do not match the number field **F** in Halo2, we need to add a constraint $x < 2^{32}$ (or $x < 2^{64}$) to represent a variable x with type $i32$ or $i64$. In ZAWA, we use $\mathbf{T_N}$ to denote a table containing elements from 0 to $2^N - 1$ and then we use a polynomial lookup to prove that values of some column c_i are less than $2^N - 1$ by $plookup(\mathbf{T_N}, c_i(j)) = 0$. When N is large (e.g. 64) and $\mathbf{T_N}$ becomes too big, we will decompose an i64 into several parts and prove that each part is less then $2^8 - 1$. Below we use the notation $x \in \mathbf{T_N}$ to denote $x < 2^N$ and omit the details of decomposing x into small pieces when necessary.

3.2 Representing Map Using Polynomial Lookup of Tables

Other than specifying a range for cells, another usage of polynomial lookup is that we can encode the state of key-value map into tables and use polynomial lookup to specify the semantics of getting a value of a certain key in a map.

Here is an example. Recall that we represent the state of ZAWA by $(iaddr, \mathcal{F}, \mathcal{M}, \mathcal{G}, \mathcal{SP}, \mathbf{I}, \mathbf{IO})$ where $\mathcal{C}$ and $\mathcal{H}$ are fixed by the WASM image. We encode state $\mathcal{C}$ and $\mathcal{H}$ in tables $\mathbf{T}_{\mathcal{C}} : Address \times$

Opcode and $\mathbf{T}_{\mathcal{H}} : Address \times U64$. By doing this, we can use the polynomial lookup to specify the semantic of getting opcode op at address $addr$ in C by $(iaddr, op) \in \mathbf{T}_C$ and specify the semantic of WASM of getting the initial byte data v at address $addr$ in $\mathcal{H}$ by $(d, addr) \in \mathbf{T}_{\mathcal{H}}$.

3.3 Representing Math Semantic as Arithmetic Circuits

According to the WASM specification, the semantics of opcodes are usually defined as mathematical equations and state transformation. Thus we need to construct arithmetic circuits to enforce the semantics of the opcodes. For example, suppose that the opcode div_u (a division of unsigned int) has the following semantics:

$$div_u(a, b) = (a - a \bmod b) \div b$$

It follows that to write the above mathematical definition into polynomial constraints we need to introduce intermediate witness r such that the above semantics can be rewritten as follows:

$$\begin{cases} a = div_u(a, b) * b + r \\ r < b \end{cases} \tag{3}$$

However, since r and b are in $\mathbf{F}$, it needs more work to represent $r < b$ into polynomial constraints. Fortunately, in ZAWA, we use range check to constraint r and b within 64 bits. The above constraints can be further rewritten into the following polynomial constraints with one more extra witness k:

$$\begin{cases} a = div_u(a, b) * b + r \\ b = r + k \\ a, r, b, k \in T_{64}, \end{cases} \tag{4}$$

When dealing with opcode that has more complicated mathematical semantics, we need a way to formally prove that the derived constraints represent the same semantic. In ZAWA, we use Z3 to formally check that the mathematical definition is correctly refined to the arithmetic circuits.

3.4 Enforcing Valid Dynamic State Accessing using Polynomial Lookup Tables

Given a sequence of state transition function $\{t_i\}$ such that each transition might read or write finitely many key-value pairs (e.g. access memory $\mathcal{M}$, stack $\mathcal{SP}$ or global $\mathcal{G}$) in the state $\mathcal{S}$. We label each read or write of $\{t_i\}$ in a sub sequence $\{t_i^k\}$ and use the tuple $(tid = i, mid = k, accessType, address, value)$ to denote the access log of $\{t_i^k\}$ such that each access log has the following semantic:

- Init memory: $(tid, mid, init, addr, v) := \{s.addr = v;\}$
- Write value v to memory: $(tid, mid, write, addr, v) := \{s.addr = v;\}$
- Read from memory: $(tid, mid, read, addr, v) := \{assert(s.addr \equiv v);\}$

As the address in t_i^k can be randomly distributed which makes it hard to reason about the fact that a read from a address $addr$ should get the value v that related to the latest write or init of that $addr$. To solve this, we do the following. First, we create a lookup table $\mathbf{T}$ by rearranging the log by their access address and order them by (tid, mid) within each address block (see Table 2).
Second, we enforce the semantic of init, read and write by equip Table 2 with constraints of each row using Equation 5.

$$\mathbf{C}_{\mathbf{T}}(cur) = \begin{cases} r(cur).address \equiv r(next).address \rightarrow r(cur).id \leq r(next).id \\ r(next).accessType \equiv read \rightarrow r(next).value \equiv r(cur).value \\ r(cur).address \neq r(prev).address \leftrightarrow r(cur).accessType \equiv init \\ r(cur).address \neq r(prev).address \rightarrow r(cur).address > r.(next).address \end{cases} \tag{5}$$

Table 2. Memory access table

address	id = (tid, mid)	accessType	value
$addr_1$	tid_1	acc_1	v_1
$addr_1$	tid_2	acc_2	v_2
$addr_1$	tid_3	acc_3	v_3
$addr_2$	tid_4	acc_3	v_4
$addr_2$	tid_5	acc_4	v_5
$\cdots$	tid_k	acc_k	v_k

Remark 3. *Rearranging means the map from access log to* **T** *is a one to one map. The first constraint enforces that for access logs visiting the same address, they are sorted by their accessing order. The second constraint enforces that the read access get the correct value and the third constraint enforces that init happens once and only once at the beginning of each address block.*

Theorem 3.1. *Give an memory access log L_i, the log L_i is valid if there exists a table* **T** *such that* **T** *satisfies the above constraints and each L_i is in* **T** *and vice versa.*

Proof. First, for init access log, the only constraint we need to enforce is that each address can only be init once. Suppose there are two init access logs L_a and L_b init the same address in the access log L_i, then they must both exists in **T** in the same address block, which contradicts the third constraint of Equation 5. Second, for read access log $L_r = (tid, mid, read, addr, v)$, we need to prove that the latest write or init access log L_{latest} to $addr$ has put value v into $addr$. Consider the second constraint of Equation 5, it is sufficient to prove that the L_{latest} is the closest entry to L_r in **T**. Suppose that L_{latest} is not the closest rewrite (init) entry to L_r in **T**, then there exists another write access log L_o between L_{latest} and L_r such that $L_o.id > L_{latest}.id$ (by the first constraint of Equation 5) which means that L_o is the latest update of $addr$ and contradicts the assumption. In the end, all the write access are valid because all the parameters are explicit. □

In the end, as a consequence of Theorem 3.1, given any read access at address $addr$ with fixed tid, mid and $accessType$, we can check the validity of the return value v by checking $(t_i, mid, accessType, addr, v) \in T$.

4 ZAWA ARCHITECTURE CIRCUITS

As we have prepared our circuit building blocks in Section 3, we start constructing the main circuits involved in ZAWA. We will first describe the workflow of ZAWA by splitting it into four stages to give a big picture of how different circuits (see Figure 1) interplay with each other and then we will present the details of each circuit.

Step 1: Image Setup. Defined by the WASM specification, a WASM image **I** is divided into sections. Among them, there are sections that do not affect the execution of WASM (custom section, type section, export section, data count section) and sections that decide the execution semantics (initial memory section, code section, global data section). At the image setup stage, we encode the code section into the lookup table $\mathbf{T_I}$ and the data section into the lookup table $\mathbf{T_H}$. These two tables will be used to enforce that each instruction in the execution trace is a valid instruction and that all the initialization of the memory access log table complies with the initial data section of image I.

Step 2: Execution Trace Generation. Recall that a valid execution trace is a sequence of transition functions $[t_0, t_1, \cdots]$ such that each t_i is related to the ith instruction during the execution of

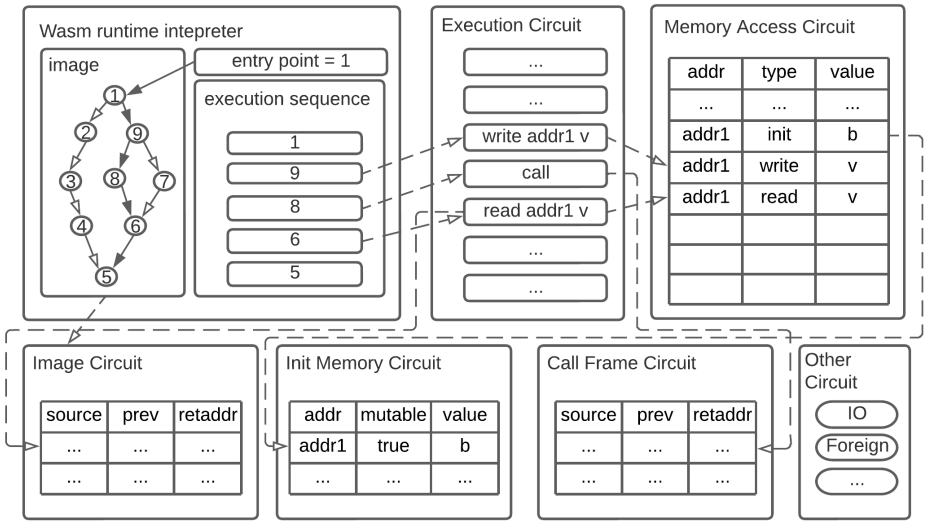

Fig. 1. Architecture circuits

(**I**, **E**, **IO**). We uses the standard WASM run-time interpreter to generate t_i that is valid as defined in Definition 2.1.

Remark 4. *We do not require the WASM run-time interpreter to be a trust component since if it generates an invalid sequence, the constraints of the Execution Circuit fail because our Execution Circuit enforces the semantics of each instruction.*

Step 3: Synthesis Circuits. Once a valid execution trace is generated, it can be used to fill our main execution circuit $\mathbf{T}_{\mathcal{E}}$, together with other lookup tables $\mathbf{T}_{\mathcal{F}}$ (calling frame table), $\mathbf{T}_{\mathcal{M}}$ (memory access log table), $\mathbf{T}_{\mathcal{G}}$ (global access log table) and $\mathbf{T}_{\mathcal{SP}}$ (stack access log table).

Step 4: Proof Generation. After all the circuits are synthesised, we can generate a ZKSNARK proof via Halo2's proof system. The proof can be used to prove that the execution trace and its output are valid.

4.1 Setup Circuits

Setup circuits are filled by the ZAWA compiler component and its purpose is to provide lookup tables $\mathbf{T}_{\mathcal{C}}$, $\mathbf{T}_{\mathcal{H}}$, $\mathbf{T}_{\mathcal{G}}$ that encode code section, initial memory section and global data section.

Code Section. The elementary items in the code section are *opcode*s of instructions that are grouped in a tree-like hierarchy. Each instruction can be indexed by *moid* (modular id), *mmid* (memory block instance id), fid (function id) and *iid* (offset of the instruction in a particular function). We denote *iaddr* to be the tuple of $(moid, mmid, fid, iid)$ and represent the code section as a map from *iaddr* to *opcode*. Using the technique in Section 3.2, it is equivalent to encoding the code section into $\mathbf{T}_{\mathcal{C}}$ (see Table 3). Code table $\mathbf{T}_{\mathcal{C}}$ is later used to constrain entries in execution table $\mathbf{T}_{\mathcal{E}}$ (see Section 4.2) such that if $e \in \mathbf{T}_{\mathcal{E}}$ then $(e.iaddr, e.opcode)$ must also in $\mathbf{T}_{\mathcal{C}}$.

Table 3. Code table

moid	mmid	fid	iid	opcode
$0x00$	$0x01$	$0x01$	$0x00$	add
$0x00$	$0x01$	$0x01$	$0x01$	sub
$\cdots$	$\cdots$	$\cdots$	$\cdots$	sub

Initial Memory & Global Data. The element items in the memory section of WASM image are unsigned 64 bit words (u64). The address of each $u64$ word can be indexed by $mmid$ and $offset$. Besides the value, memory can have types that are either mutable or immutable. Thus the memory section can be represented as a map from $(mmid, offset)$ to $(value, isMutable)$. Similarly, using the technique in Section 3.2, we can encode the initial memory section into $\mathbf{T}_{\mathcal{H}}$. Similar to the init memory section, the global data section contains variable instances that can be shared between different modules which can also be represented as a map from $(mmid, offset)$ to $(value, isMutable)$. Thus we merge two tables into one and use $ltype = Memory \mid Global$ to distinguish them (see Table 4 for an example of $\mathbf{T}_{\mathcal{H}}$).

Table 4. Initial memory table

$ltype$	$mmid$	$offset$	$value$	$isMutable$
$Heap$	$mmid_0$	1	$0x01$	$true$
$Heap$	$mmid_1$	1	$0x01$	$true$
$Global$	$mmid_2$	1	$0x01$	$true$
$Global$	$mmid_3$	1	$0x01$	$false$

We use $\mathbf{T}_{\mathcal{H}}$ to constrains entries in the memory access log table $\mathbf{T}_{\mathcal{M}}$ (see Table 2) so that $\forall e, e \in \mathbf{T}_{\mathcal{M}} \wedge e.accessType = Init \rightarrow (e.iaddr, value) \in \mathbf{T}_{\mathcal{H}}$. The meaning of this constraint is that for each init access log in $\mathbf{T}_{\mathcal{M}}$ it must be defined in the initial memory section or global data section.

4.2 Execution Trace Circuits

Execution Trace Circuits are used to constraint the execution trace $[t_0, t_1, t_2, \cdots]$ (see Section 2.1) emulated from WASMI (WASM interpreter). Each trace element is related to an instruction in the code table $\mathbf{T}_C$ and has a predefined semantic based on the opcode. The semantics of a WASM opcode is defined based on its parameters derived from the stack and the micro operations. First, since WASM is a stack machine, we define the *operands* of an opcode op to be

$$operands(op) = p_0, p_1, p_2 \cdots, p_k$$

where p_i are values on the stack and $p_i = stack[sp + i]$. Second, we define the semantics of op by a sequence of microoperations

$$mop_i = \begin{cases} w_i = load(ltype, addr) \text{ where } addr \in \{p_1, p_2, \cdots, p_k, w_0, w_1, \cdots, w_{i-1}\} \\ write(ltype, addr, v) \text{ where } addr, v \in \{p_1, p_2, \cdots, p_k, w_0, w_1, \cdots, w_{i-1}\} \\ w_i = arith(p_1, p_2, \cdots, p_k, w_0, w_1, \cdots, w_{i-1}); \\ FALLTHOUGH; \\ GOTO(iaddr); \\ if\ b\ then\ \{mop_{i+1}, \cdots mop_j\}\ else\ \{mop_{j+1}, \cdots\}. \end{cases}$$

When filling execution trace into the execution circuit, we arrange the instruction into small blocks (see Table 5) of the execution circuit such that each block represents an instruction. Within each block, we use the *start* column to indicate whether this row is the start of a new instruction block and put *op* and *mop* in the opcode column. In the address column, we push all used addresses and the first row is the instruction address of this instruction in $\mathbf{T}_C$ and in the *sp* column we record all the changes of stack pointer.

Table 5. Execution table

start	opcode	bit cell	state	aux	$address \in T_I$	*sp*	u64 cell
true	op	b_0	tid_0	aux	$iaddr_0$	sp	w_0
0	mop_0	b_1	$frame$	aux_0	$addr_0$	$\cdots$	w_1
0	mop_1	b_2	..	aux_1	$addr_1$	$\cdots$	w_2
0	mop_2	b_3	s_3	aux_2	$addr_2$	$\cdots$	w_3
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
true	op_1	b	tid_1	aux	$iaddr_1$	sp'	w
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Although different opcodes might have different semantics thus different mop_k, $addr_i$, etc. There are some common constraints that we need to enforce in the execution circuit. First. we need to enforce that each instruction exists in the code section, thus $(iaddr, opcode) \in \mathbf{T}_C$. Second, suppose that *operand* p_i is got from stack pointer *sp* as a result of $mop_k(sp)$, then $(sp, read, iaddr, k, p_i) \in \mathbf{T}_{\mathcal{M}}$, which means the result p_i is enforced from a valid memory access log table. Similarly, suppose that witness w_i is got from memory access of $addr_j$ with access type $ltype$ as a result of mop_k, then $(mem, addr_j, ltype, k, w_i) \in \mathbf{T}_{\mathcal{M}}$. Third, we enforce that all the cells in bit column are either zero or one and all the cells in u64 witness column and operand column are in $\mathbf{T}_{64}$ (less than 2^{64}).

4.3 Frame Circuit

Frame Circuit is a table (see Figure 2) that helps us to find out the next *iaddr* of the return instruction (see Section 5.2). Each entry of $\mathbf{T}_{\mathcal{F}}$ is a tuple of $(prevFrame, currentFrame, iaddr)$ where $currentFrame$ is the tid of the call instruction that starts this call frame, $prevFrame$ is the tid of the call instruction of previous call frame and *iaddr* is the call instruction address of the current call frame. Suppose that t_i is a return instruction at state s_i with $(currentFrame, prevFrame)$ and the state $s_{i+1} = t_i \circ t_{i-1} \circ \cdots t_0(s_0)$, then we constrain that

$$plookup(\mathbf{T}_{\mathcal{F}}, (prevFrame, currentFrame, s_{i+1}.(iaddr - 1))) = 0$$

to make sure the return address is correct (see Figure 2).

4.4 Access Log Circuit

Recall that the access log circuit is a unique table corresponding to a valid memory access log sequence and satisfies Equation 5. In WASM specification, an access log is used for three different types that are memory access, stack access and global access. Each access log has a type field that is either *Init*, *Read* or *Write* and all logs are sorted by $(address, (tid, tmid))$ where *address* is indexed by $(mmid, offset)$, tid is the transition index of the execution log that contains the access and $tmid$ is the index of the access micro-op in that instruction (see Table 2).

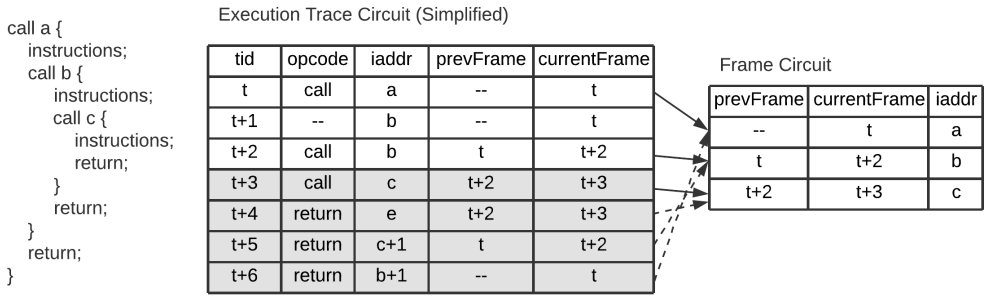

Fig. 2. Frame circuit

4.5 IO Circuits that Support Zero-knowledge

Zero-knowledge of inputs is not supported in WASM specification. Thus to support the private inputs which we do not want to leak, we need to add special instructions in the ZAWA to distinguish between private and public inputs. We represent public inputs in a separate column and use the polynomial lookup to link input values with the result of *get_public_input(inputCursor)* (See Figure 3). Similarly, we use a separate column to hold output data and use a polynomial lookup to enforce

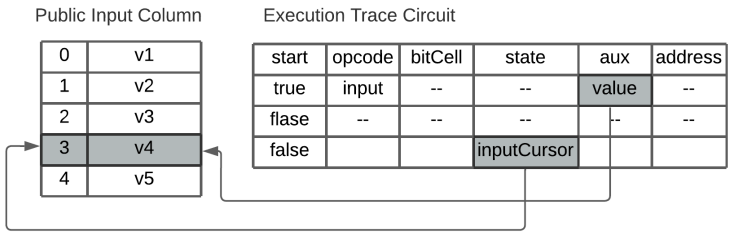

Fig. 3. Public input circuit

that the value we output in the execution circuit *aux* cell lies in the output column. When dealing with private inputs from *get_private_input(inputCursor)*, we put them into the related cell with no constraints as the proof system will hide the value for us.

5 INSTRUCTION CIRCUITS

Once the architecture circuits are all prepared in Section 4, the remaining things are constructing the circuits C_{op} of various opcode *op* for instructions supported by WASM specification. Since the constraint defined on the execution trace circuit will be applied on a row basis, and the cells of the constraints of each *op* will span over multiple rows, we use the notation $c.(curr + k)$ to denote the kth cell in column c followed by the current row.

For example, suppose that we want to define the constraints of add instruction (see Figure 4) using the circuit layout in Table 6 where w_1, w_2 are got from the stack and w_0 is equal to the result of the add instruction which is pushed back to the stack. First, we know that by definition of *add*, $w_0 = (w_1 + w_2) \bmod 2^{64}$. Thus by introducing a new witness $overflow$ we encode the mod

Table 6. Add circuit within execution trace circuit

start	opcode	bit cell	state	aux	$address \in T_I$	$sp \in T_{\mathcal{F}}$	u64 cell	extra
true	add	$overflow$	tid	nil	$iaddr_0$	sp	w_0	nil
0	$readStack$	nil	nil	nil	nil	–	w_1	nil
0	$readStack$	nil	nil	nil	nil	–	w_2	nil
0	$writeStack$	nil	nil	nil	nil	–	w_3	nil
true	$otherop$	–	$tid+1$	nil	$iaddr_1$	sp'	w'_0	nil

```
def add :=
    w1 = read(stack sp);
    w2 = read(stack sp-1);
    sp' = sp-1;
    w0 = (w1 + w0) mod 2^64;
    write(stack, sp-1, w0);
    FALLTHROUGH
```

Fig. 4. Add instruction definition

semantic into arithmetic constraint as $w_0 + overflow \times 2^{64} = w_1 + w_2$. Second, we enforce the stack operation are valid, that is $(stack, read, sp-1, tid, 0, w0) \in T_{\mathcal{M}}$, $(stack, read, sp, tid, 1, w1) \in T_{\mathcal{M}}$ and $(stack, write, sp-1, tid, 2, w2) \in T_{\mathcal{M}}$. Third, we need to constrain that the next instruction must follow $iaddr_0$ in address, therefore $iadd_1 = iaddr_0 + 1$. In the end, we constrain the sp column by $sp' + 1 = sp$. Put it all together, and replace variables using the notation of $columnName.(curr + k)$, we have Equation 6.

$$\mathbf{C_{add}} = \begin{cases} w.(curr) + bit.(curr) \times 2^{64} - w.(curr+1) + (w.curr+2) = 0 \\ Plookup(T_{\mathcal{M}}, (stack, read, sp.(curr), tid, 0, w1)) = 0 \\ Plookup(T_{\mathcal{M}}, (stack, read, sp.(curr)-1, tid, 1, w2)) = 0 \\ Plookup(T_{\mathcal{M}}, (stack, write, sp.(curr)-1, tid, 2, w0) = 0 \\ iaddr.curr + 1 - iaddr.(curr+4) = 0 \\ sp.(curr+4) + 1 - sp.(curr) = 0 \end{cases} \tag{6}$$

Since constraints are applied on a row basis of a circuit, we need to make sure that $\mathbf{C}_{add}$ does not apply on rows that are not a starting row of an instruction block or a block with other opcodes. So a natural way to apply C_{add} only on necessary rows is to multiply $\mathbf{C}_{add}(curr)$ with $curr.start \times (curr.opcode == op)$ and the final constraint related to opcode add is $\overline{\mathbf{C}}_{add}(curr) := curr.start \times (curr.opcode == op) \times \mathbf{C}_{add}(curr) = 0$.

Remark 5. *For better readability, from this point we will simply use the name of the cell instead of the notation $columnName.(curr + k)$ if no confusion is introduced by doing so.*

The content of the rest of this section is arranged in subsections to describe circuits of instructions in different categories. After we have constructed all the constraints C_{op_i} for all opcodes op_i, we simply sum them up and get the final constraint $C_{op}(curr) := \sum_i cur.start \times (cur.opcode == op) \times C_{op_i}(curr) = 0$ for $\mathbf{T}_{\mathcal{E}}$.

5.1 Numeric Instructions

Numeric Instructions are the majority of instructions in WASM. In general, the semantics of numeric instructions contain file parts, parameters preparation, arithmetic calculation, writing result back to stack, update stack pointer and FALLTHROUGH as in Figure 5.

```
def arithop :=
    param1 = read(stack sp); \\ parameters preparation
    param2 = read(stack (sp-1)); \\ parameters preparation
    ...
    paramN = read(stack (sp-N+1)); \\ parameters preparation
    result = arith(param1, param2, param3, ..., paramN); \\ calculation
    write(stack, (sp-N+1), result); \\ result write back
    sp = sp-N+1;
    FALLTHROUGH;
```

Fig. 5. Arithmetic instruction

Based on the arithmetic definition, we assign the cells in the execution trace circuit $\mathbf{T}_{\mathcal{E}}$ in Table 7 and the constraints of arithmetic opcode are defined in Equation 7.

Table 7. Add circuit in execution trace circuit

start	opcode	bit cell	state	aux	$address \in T_I$	$sp \in T_{\mathcal{F}}$	u64 cell	extra
true	$arithOp$	$nill$	tid	nil	$iaddr_0$	sp	$param_0$	nil
0	nil	nil	nil	nil	nil	nil	$\cdots$	nil
0	nil	nil	nil	nil	nil	nil	$param_N$	nil
0	nil	nil	nil	nil	nil	nil	$result$	nil
true	$otherop$	–	$tid+1$	nil	$iaddr_1$	sp'	w'_0	nil

$$\mathbf{C}_{arith} = \begin{cases} arith(param_0, param_1, ..., param_N) - result = 0 \\ Plookup(\mathbf{T}_{\mathcal{M}}, (stack, read, sp-k, tid, k, param_k)) = 0 \\ Plookup(\mathbf{T}_{\mathcal{M}}, (stack, write, sp'-1, tid, N, result)) = 0 \\ iaddr_0 + 1 - iaddr_1 = 0 \\ sp - sp' - N + 1 = 0 \end{cases} \tag{7}$$

5.2 Control Flow Instructions

In WASM specification, there are three different categories of control flow: *FALLTHROUGH*, *branch*, and *call* (*return*). Implementation of the *FALLTHROUGH* is already covered in Section 5.1. Thus it is sufficient to implement call (return) and branch.

Call (Return) Circuit. Call instruction will first add a new frame table entry $(tid, preFrameId, iaddr_0)$ into the frame circuits $\mathbf{T}_{\mathcal{F}}$ and then load calling parameters onto the stack and go to the $iaddr_1$ for next instruction (see Table 8 for the circuit layout of *call*). The circuit constraint for call instruction is Equation 8.

Table 8. Circuit layout of call

start	opcode	bit cell	state	aux	$address \in T_I$	$sp \in T_{\mathcal{F}}$	u64 cell	extra
true	$call(targetIaddr)$	$nill$	tid	nil	$iaddr_0$	sp	$param_0$	nil
0	nil	nil	$preFrameId$	nil	nil	nil	$\cdots$	nil
0	nil	nil	nil	nil	nil	nil	$param_N$	nil
true	$otherop$	–	$tid + 1$	nil	$iaddr_1$	sp'	nil	nil
0	nil	nil	tid	nil	nil	nil	nil	nil

$$C_{call} = \begin{cases} Plookup(\mathbf{T}_{\mathcal{M}}, (stack, write, sp + i, tid, i, param_i)) = 0 \\ Plookup(\mathbf{T}_{\mathcal{F}}, (tid, pFrameId, iaddr_0)) = 0 \\ iaddr_1 - targetIaddr = 0 \\ sp' - sp - N = 0 \\ nFrameId - tid = 0. \end{cases} \tag{8}$$

To define the constraint for *Return* instruction, we need to find the correct return address of the current frame and set the frame state to the previous frame. Recall that, as described in Section 4.3, the entries in $\mathbf{T}_{\mathcal{F}}$ are used to help the return instruction to find the correct calling frame and previous frame. Thus we can define the semantics of *return* by finding the correct return *iaddr* from $T_{\mathcal{F}}$ and then enforce that the next instruction address is equal to *iaddr* and update the frame state accordingly (see Table 9 for the circuit layout and Equation 9 for the circuit constraint).

Table 9. Circuit layout of return

start	opcode	bit cell	state	aux	$address \in \mathbf{T}_I$	$sp \in \mathbf{T}_{\mathcal{F}}$	u64 cell	extra
true	$return$	$nill$	tid	nil	$iaddr_0$	sp	nil	nil
0	nil	nil	$prevFrameId$	nil	nil	nil	nil	nil
true	$otherop$	–	$tid + 1$	nil	$iaddr_1$	sp'	nil	nil
0	nil	nil	$nFrameId$	nil	nil	nil	nil	nil

$$C_{return} = \begin{cases} Plookup(\mathbf{T}_{\mathcal{F}}, (pFrameId, nFrameId, iaddr_1 - 1)) = 0 \\ sp' - sp = 0 \end{cases} \tag{9}$$

Branch Circuit. Branch instructions in WASM include *br*, *br_if*, *if * then * else **, etc. The semantics of branch instructions can be uniformly abstracted as three steps (see Figure 6). The circuit layout

```
def branchop :=
    param1 = read(stack sp); \\ parameters preparation
    param2 = read(stack (sp-1)); \\ parameters preparation
    ...
    paramN = read(stack (sp-N+1)); \\ parameters preparation
    iaddr1 = select(param1, param2, ..., paramN); \\ calculate branch address
    GOTO iaddr2; \\branch to target address
```

Fig. 6. Semantic of branch instruction

of the branch instruction is sketched in Table 10 and its circuit constraint is defined in Equation 10.

Table 10. Circuit layout of call

start	opcode	bit cell	state	aux	$address \in T_I$	$sp \in T_{\mathcal{F}}$	u64 cell	extra
true	$branchop$	$nill$	tid	nil	$iaddr_0$	sp	$param_0$	nil
0	nil	nil	$frameId$	nil	nil	nil	$\cdots$	nil
0	nil	nil	nil	nil	nil	nil	$param_N$	nil
true	$otherop$	–	$tid + 1$	nil	$iaddr_1$	sp'	nil	nil
0	nil	nil	$frameId$	nil	nil	nil	nil	nil

$$C_{branch} = \begin{cases} Plookup(T_{\mathcal{M}}, (stack, write, sp + i, tid, i, param_i)) = 0 \\ iaddr_1 - select(param_0, param_1, \cdots) = 0 \\ nFrameId - pFrameId = 0. \end{cases} \tag{10}$$

5.3 Memory (Stack, Global) Instructions

Memory, Stack and Global instructions can be abstracted as a tuple of ($category$, $type$, $address$, $size$ = 8|16|32|64, $value$) where $category$ can be Memory, Stack or Global and $type$ can be Init, Read or Write. The layout of the circuit is defined in Table 11. By using the access log circuit defined in Chapter 4.4, the constraint for the memory circuit is simply $(category, ltype, tid, address, value') \in T_{\mathcal{M}} \wedge trunc(value', size) = value$.

Remark 6. *For read, this constraint ensures the result read from address is valid. For write, $T_{\mathcal{M}}$ ensures that the next read of address will return the previously written value correctly.*

Table 11. Memory access circuit within execution trace circuit

start	opcode	bit cell	state	aux	$address \in T_I$	$sp \in T_{\mathcal{F}}$	u64 cell	extra
true	$op(type, size)$	$nill$	tid	nil	$iaddr_0$	sp	$address$	nil
0	nil	nil	$frameId$	nil	nil	nil	$value$	nil
true	$otherop$	–	$tid + 1$	nil	$iaddr_1$	sp'	w'_0	nil
0	nil	nil	$frameId = tid$	nil	nil	nil	w_3	nil

5.4 Customized Instruction Extension

Given a fixed image, an entry function and an array of input arguments, the execution trace is then decided which means the number of instructions is fixed. As described in Section 4.2, each instruction occupies n (a fixed number of) rows in the execution circuit $\mathbf{T}_{\mathcal{E}}$. Thus the total rows of $\mathbf{T}_{\mathcal{E}}$ are fixed. When doing proof in Halo2 using KZG commitment, each column is interpolated into polynomials using FFT (fast fourier transform). Because FFT is an algorithm of $NlogN$ complexity, the total rows of $T_{\mathcal{E}}$ affect the overall performance in a nonlinear way. Thus to reduce the number of columns, a good way is to compress multiple instructions into one.

ZAWA supports two ways of customizing foreign instructions for the purpose of compressing. One is implementing customized inline opcodes and the other is using external proofs of specific foreign functions.

Compress using customized inline instruction. When the semantics of instructions we would like to compress is simple and can fit into one instruction block, we can use the inline extension. For

example, suppose that we want to sum the lowest 4 bits $x : u64$ by function $sumLowest(x)$. If we use a standard loop to implement the algorithm, it will take 4 instructions to extract 4 bits and another 2 to do the sum. However, if we inline this function into a customized inline instruction, then we can encode the arithmetic constraint within one instruction block. As a case study, we compare the SHA256 execution trace with and without inline instructions in Table 12.

Table 12. Row reduce by using customized instructions

original	original rows	customized	optimized rows
$\lambda x, y, (x\&y)\|(complete(x)\&z)$	4	$ch(x, y)$	1
$\lambda x, y, z, z\|(x\&(y\|z))$	2	$maj(x, y, z)$	1
$\lambda x, rotr_{32}(x, 2)\|rotr_{32}(x, 13)\|rotr_{32}(x, 22)$	5	$lsigma0(x)$	1
$\lambda x, rotr_{32}(x, 6)\|rotr_{32}(x, 11)\|rotr_{32}(x, 25)$	5	$lsigma1(x)$	1

Compress using customized foreign functions. When the semantics of instructions we would like to compress is too complex to fit into one instruction block and the semantic of these instructions can be abstracted into a pure function then we can using foreign functions to compressing the execution trace. A foreign function f in ZAWA is a special purpose circuit that can be used to constraint that the input and output of f is valid. Although foreign calls saves the size of it will introduce extra costs when more circuits are added.

6 PROGRAM PARTITION AND PROOF BATCHING

As discussed in Section 5.4, when encoding execution trace in $T_{\mathcal{E}}$, each instruction will take a constant number of rows. In Halo2 proof system, there is a limit to the total number of rows of arithmetic circuits [25]. Therefore, for large WASM images, we probably can not fit the whole execution trace into $T_{\mathcal{E}}$. To solve the problem of long execution trace, ZAWA use the technique of program partition and proof batching. The idea is that we split the execution trace $[t_0, t_1, \cdots]$ into a group of sub sequences, generate execution proof for each group and batch all the proofs in the end. Here we first give a bold sketch of the overview of the technique and then presents the extra constraints we need to provide when batching sub proofs.

Given an execution sequence t_i, we split it into small execution chunks $t_{[a,b]} = t_a, t_{a+1}, \cdots, t_{b-1}$ and denote the $\mathcal{M}_{[a,b]}, \mathcal{SP}_{[a,b]}, \mathcal{G}_{[a,b]}$ to be the memory, stack and global access log related to $t_{[a,b]}$. We notice that given an execution trace $t_{[a,b]}$, by using the arithmetic circuits constructed in Section 4, we can prove $t_{[a,b]}$ is valid under the context $(\mathcal{F}, \mathcal{M}_{[a,b]}, \mathcal{SP}_{[a,b]}, \mathbf{I}(\mathcal{C}, \mathcal{H}), \mathbf{IO})$. We denote $\mathcal{P}_{[a,b]}$ the proof of the valid execution of $t_{[a,b]}$ and $\mathcal{P}$ is the proof of the valid execution of t_i under the full access log $(iaddr, \mathcal{F}, \mathcal{M}, \mathcal{G}, \mathcal{SP}, \mathbf{I}, \mathbf{IO})$. Now it remains to find out what conditions we need to enforce so that

$$(\mathcal{P}_{[0,k-1]} \wedge \mathcal{P}_{[k,2k-1]} \wedge \cdots \wedge \mathcal{P}_{[0,end]}) \rightarrow \mathcal{P}.$$

Thus it is sufficient to make sure that for each $t_i \in t_{[a,b]}$ the constraints applied on it in $\mathcal{P}_{[a,b]}$ is equivalent to the constraints applied on it in $\mathcal{P}$. As we have presented in Section 4, constraints applied on each instruction block in $T_{\mathcal{E}}$ contains two parts, that are polynomial constraints about cells of the current and next instruction block and constraints of polynomial lookup of state (memory, stack, global) access logs.

Equivalent of Polynomial Constraints. Regarding the polynomial constraints of cells, it is easy to check that if $t_i \in t_{[a,b]}$ and $i < b$ then all polynomial constraints of cells of the instruction block of

t_i in $\mathcal{P}_{[a,b]}$ are equal to the those in $\mathcal{P}$. So it remains to constrain that the last instruction of $t_{[a,b]}$ has the same polynomial constraints both in $\mathcal{P}_{[a,b]}$ and $\mathcal{P}$. However, it is not true in general since if we split the execution sequence into blocks that are disjoint, then the connection between the two sequences is lost. Therefore, to solve this problem, we need to pad a glue instruction at the end of each sub sequence and enforce the address of the glue instruction equal to the address of the first instruction of the next block (see Figure 7). By doing so we can check that the polynomial constraints of each t_i in $\mathcal{P}_{[a,b]}$ is equivalent as it is in $\mathcal{P}$.

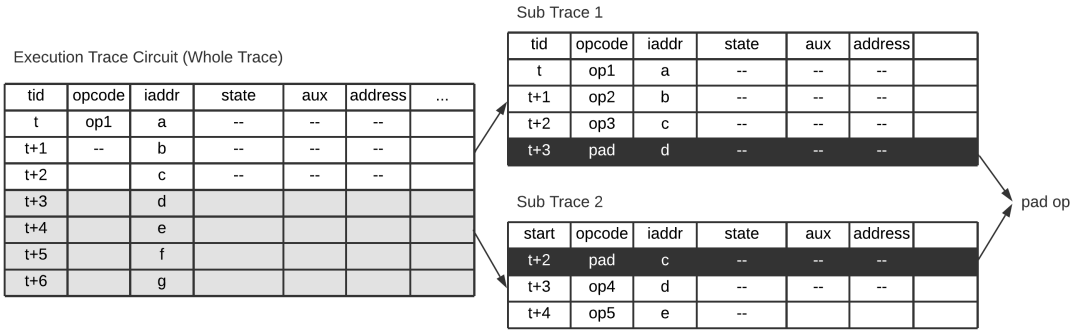

Fig. 7. Split execution sequence into sub sequence

Equivalent of Polynomial Lookup. Given a constraint of polynomial lookup for a cell in t_i, we need to show that $c \in \mathbf{T}_{\mathcal{M}}$ if and only if $c \in \cup \mathbf{T}_{\mathcal{M}_k}$. By the definition of Equation 5 we know that the property hold if and only if the concatenate of $\mathbf{T}_{\mathcal{M}_k}$ satisfies Equation 5. Notice that Equation 5 only constraints adjacent rows, we extract a glue table $\mathbf{TG}_{\mathcal{M}}$ for $\mathbf{T}_{\mathcal{M}_k}$ $(k = 1, 2, \cdots)$ as in (Figure 8) and then it follows that $\mathbf{T}_{\mathcal{M}} = \cup \mathbf{T}_{\mathcal{M}_k}$ if $\mathbf{T}_{\mathcal{M}}$, $\mathbf{T}_{\mathcal{M}_k}$ and $\mathbf{TG}_{\mathcal{M}}$ all satisfy Equation 5.

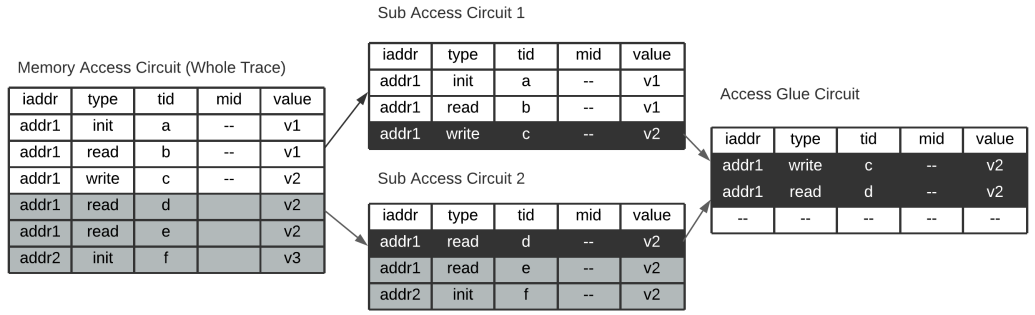

Fig. 8. Split memory access log into sub log

As a conclusion, to solve the long execution trace problem, we split the execution trace into $t_{[a_0,b_0]}, t_{[a_1,b_1]}, t_{[a_2,b_2]}, \cdots$ and construct $\mathbf{T}_{\mathcal{M}_k}$ $\mathbf{T}_{\mathcal{SP}_k}$ $\mathbf{T}_{\mathcal{G}_k}$ for execution block $t_{[a_k,b_k]}$. Suppose that $\mathcal{P}_k$ proves the valid execution of $t_{[a_k,b_k]}$ and $\mathbf{TG}_{\mathcal{M}}$ is the gluing map constructed as in Figure 8,

then we claim that a proof $\mathcal{P}_{batch}$ can prove that the execution sequence t_i is a valid execution if and only if $\mathcal{P}_{batch}$ is the batched proof of all the constraints in Equation 11.

$$\begin{cases} \mathcal{P}_k \text{ proves } t_{[a_k,b_k]} \text{ is a valid execution under } (\mathcal{F}, \mathcal{M}_{[a_k,b_k]}, \mathcal{SP}_{[a_k,b_k]}, \mathcal{I}(C,\mathcal{H})). \\ t_{b_{k+1}}.iaddr = t_{a_k}.iadr \text{ when } k > 0\ . \\ t_{b_k} \text{ is a glue instruction when } k > 0 \text{ and } t_{[a_k,b_k]} \text{ is not the last exectuion block.} \\ TG_{\mathcal{M}} \text{ satisfies Constraint 5.} \end{cases} \tag{11}$$

In ZAWA, we write the verifying algorithm of $\mathcal{P}_k$ into arithmetic circuits $\mathcal{V}_k$ and the total batch circuit of Equation 11 is construct by putting the verifying circuits together with the circuits that do the other simple checks.

Remark 7. *Proof batching is an active research topic. Instead of writing verify function into arithmetic circuits, there are other methods [5, 13, 24, 31] that are worth trying as well. Since we focus more on the consistency of program partition and memory access log in this paper, we leave the analysis of trying different batching methods as future work.*

7 PERFORMANCE BENCHMARK

All the benchmark test suites are run on a machine with AMD Ryzen 7 5800X3D 8-Core Processor, one GeForce RTX 3090 graphic card and 32G * 4 DDR4 2133 ram.

7.1 Performance without Program Partition and Proof Batching

Among programs whose valid execution trace fits into the max sequence size of ZAWA, we measure the performance of ZAWA when dealing with two special programs: the Fibonacci function which has a deep call stack and the binary search function which has frequent memory access. As shown in Table 13 and Table 14, circuit size *CS* denotes the total number of rows of circuits in ZAWA as a power of 2 and trace size denotes the total instructions included in the execution trace. Proof time is the time ZAWA used to create the proof for the valid execution and the verify time is the time for a verifier to check the proof. The column *memory swap* indicates whether the overall memory consumption of ZAWAis large than 128G.

Table 13. Benchmark for fibonacci in ZAWA

circuit size	trace size	call depth	proof time	verify time	memory swap
18	9037	13	44s	22 ms	false
19	23677	15	88s	24 ms	false
20	38317	16	178s	22 ms	false
21	100333	18	358s	22ms	false
22	162349	19	828s	29ms	true

From Table 13 and Table 14 we see that the verifying time is O(1) and proof time grows linearly when the size of the circuit grows. When the memory consumption of the simulated image (Fibonacci) is small, the instruction volume grows linearly as the circuit size grows (see Table 13). When the memory consumption of the simulated image (binary search) grows linearly as the circuit size grows, the instruction volume grows slowly as shown in Table 14.

7.2 Performance for Large Program with Proof Batching

For a program with a large execution sequence, we split the execution trace into partitions and generate a sub-proof for each partition. To batch these sub proofs, we use the batching circuit

Table 14. Benchmark for binary search in ZAWA

circuit size	trace size	search buf size (64k per page)	proof time	verify time	memory swap
18	585	26	44.200s	22 ms	false
19	616	63	87.200s	24 ms	false
20	647	124	173.200s	22 ms	false
21	678	246	342.200s	24ms	false
22	809	490	803.200s	25ms	true

(see Section 6) to generate a batched proof. Therefore the time of creating a final proof of a large program is the sum of sub-proof generation time plus the proof batching time. It can be seen in

Table 15. Benchmark for proof batching in ZAWA

partition CS	partition proof	total pieces	batching CS	batching proof	verify time	memory swap
18	2	1	22	168s	4.7 ms	false
18	2	2	22	326s	4.63 ms	false
20	2	1	22	168s	4.77 ms	false
20	2	2	22	324s	5.05ms	false
22	2	1	22	168s	5.07ms	false
22	2	2	22	325s	4.93ms	true

Table 15 that the proving time for batching sub-proofs with different partition size is constant and the only factor that affects the batching proof time is the number of pieces of sub-proof.

In conclusion, if we have a large program for ZAWA and we want to optimize the total time of generating the final proof, then the factors we need to consider are the following: First, What is the partition size we use to split the whole transaction sequence. Second, how many pieces we should put together in one proof batch. Third, how we generate the proofs in parallel.

8 CONCLUSION & FURTHER WORK

We presented ZAWA, a WASM virtual machine that leverages the technology of ZKSNARK. As far as know, we are the first to present a novel way to implement the semantics of a WASM virtual machine in arithmetic circuits. Using ZAWA, we are able to deploy serverless service in an untrusted cloud since ZAWA does not only emulates the execution but also gives a correctness proof of the execution result. Hence any vulnerability on the cloud will not affect the correctness of the verifiable result (a faulty result cannot have correct ZKSNARK proof).

For large applications that provide large execution traces, we successfully applied the execution partition and proof batching technique so the ZAWA scales when the program size grows. Regarding the performance, various optimization can be applied in the future including improving the commitment scheme [12], adopting a better parallel computing strategy [44], using SNARK specific hardware [38, 45], etc.

REFERENCES

[1] Farag Azzedin and Muthucumaru Maheswaran. Evolving and managing trust in grid computing systems. In *IEEE CCECE2002. Canadian Conference on Electrical and Computer Engineering. Conference Proceedings (Cat. No. 02CH37373)*, volume 3, pages 1424–1429. IEEE, 2002.

[2] Olivier Bégassat, Alexandre Belling, Théodore Chapuis-Chkaiban, and Nicolas Liochon. A specification for a zk-evm, 2021.

[3] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference*, pages 90–108. Springer, 2013.

[4] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference*, page 90–108. Springer, 2013.

[5] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79(4):1102–1160, 2017.

[6] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Efficient polynomial commitment schemes for multiple points and polynomials. *Cryptology ePrint Archive*, 2020.

[7] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Recursive zk-snarks from any additive polynomial commitment scheme. *Cryptology ePrint Archive*, 2020.

[8] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, page 327–357. Springer, 2016.

[9] Benedikt B"unz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*, page 315–334. IEEE, 2018.

[10] Benedikt B"unz, Ben Fisch, and Alan Szepieniec. Transparent snarks from dark compilers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, page 677–706. Springer, 2020.

[11] Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning. Trustless grid computing in concert. In *International Workshop on Grid Computing*, pages 112–125. Springer, 2002.

[12] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. *Cryptology ePrint Archive*, 2022.

[13] Alessandro Chiesa, Lynn Chua, and Matthew Weidner. On cycles of pairing-friendly elliptic curves. *SIAM Journal on Applied Algebra and Geometry*, 3(2):175–192, 2019.

[14] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: preprocessing zksnarks with universal and updatable srs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, page 738–768. Springer, 2020.

[15] Stephen Chong, Eran Tromer, and Jeffrey A Vaughan. Enforcing language semantics using proof-carrying data. *Cryptology ePrint Archive*, 2013.

[16] Craig Costello, C'edric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, page 253–270. IEEE, 2015.

[17] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *International Conference on Automated Deduction*, pages 106–120. Springer, 2003.

[18] Jacob Eberhardt and Stefan 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)*, page 1084–1091. IEEE, 2018.

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

[20] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, pages 265–279, 2020.

[21] Jens Groth. Efficient zero-knowledge arguments from two-tiered homomorphic commitments. In *International Conference on the Theory and Application of Cryptology and Information Security*, page 431–448. Springer, 2011.

[22] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.

[23] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *Annual International Cryptology Conference*, page 581–612. Springer, 2017.

[24] Ulrich Haböck, Alberto Garoffolo, and Daniele Di Benedetto. Darlin: Recursive proofs using marlin. *arXiv preprint arXiv:2107.04315*, 2021.

[25] Halo2. The halo2 book, 2020. Accessed on 2022-9-3.

[26] Max Hoffmann, Michael Klooß, and Andy Rupp. Efficient zero-knowledge arguments in the discrete log setting, revisited. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2093–2110, 2019.

[27] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *2013 ACM SIGSAC Conference on Computer and Communications Security, (CCS'13)*, pages 955–966. ACM, 2013.

[28] Yier Jin and Yiorgos Makris. Proof carrying-based information flow tracking for data secrecy protection and hardware trust. In *2012 IEEE 30th VLSI Test Symposium (VTS)*, pages 252–257. IEEE, 2012.
[29] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Polynomial commitments. *Tech. Rep*, 2010.
[30] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, page 944–961. IEEE, 2018.
[31] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*, pages 359–388. Springer, 2022.
[32] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, page 2111–2128, 2019.
[33] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. Security chasms of wasm. *NCC Group Whitepaper*, 2018.
[34] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Communications of the ACM*, 59(2):103–112, 2016.
[35] Juha Partala, Tri Hong Nguyen, and Susanna Pirttikangas. Non-interactive zero-knowledge for blockchain: A survey. *IEEE Access*, 8:227945–227961, 2020.
[36] Luke Pearson, Joshua Fitzgerald, Héctor Masip, Marta Bellés-Muñoz, and Jose Luis Muñoz-Tapia. Plonkup: Reconciling plonk with plookup. *Cryptology ePrint Archive*, 2022.
[37] Siani Pearson. Taking account of privacy when designing cloud computing services. In *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 44–52. IEEE, 2009.
[38] BO Peng, Yongxin Zhu, Naifeng Jing, Xiaoying Zheng, and Yueying Zhou. Design of a hardware accelerator for zero-knowledge proof in blockchains. In *International Conference on Smart Computing and Communication*, pages 136–145. Springer, 2020.
[39] risc zero. risc zero: overview of the zkvm, 2022. Accessed on 2022-9-3.
[40] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, 8(6):24–31, 2010.
[41] Vitalik. The different types of zk-evms, 2022. Accessed on 2022-9-3.
[42] Riad S Wahby, Srinath Setty, Max Howald, Zuocheng Ren, Andrew J Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. *Cryptology ePrint Archive*, 2014.
[43] Gavin Wood and Jutta Steiner. Trustless computing—the what not the how. In *Banking Beyond Banks and Money*, pages 133–144. Springer, 2016.
[44] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, 2018.
[45] Charles F Xavier. Pipemsm: Hardware acceleration for multi-scalar multiplication. *Cryptology ePrint Archive*, 2022.
[46] Zhifeng Xiao and Yang Xiao. Security and privacy in cloud computing. *IEEE communications surveys & tutorials*, 15(2):843–859, 2012.

Received 10 November 2022