# ZoKrates - Scalable Privacy-Preserving Off-Chain Computations

Jacob Eberhardt, Stefan Tai
*Information Systems Engineering (ISE)*
*TU Berlin*
Berlin, Germany
{je, st}@ise.tu-berlin.de

***Abstract*—Scalability and privacy are two major challenges for today's blockchain systems. Processing transactions at every node in the system limits the system's ability to scale. Further, the requirement to publish all corporate or individual information for processing at every node, essentially making the data public, is – despite of all other advantages – often considered a major obstacle to blockchain adoption. In this paper, we make two main contributions to address these two problems: (i) To increase efficiency, we propose a processing model which employs non-interactive proofs to off-chain computations, thereby reducing on-chain computational efforts to the verification of correctness of execution rather than the execution itself. Due to the verifiable computation scheme's zero-knowledge property, private information used in the off-chain computation does not have to become public to verify correctness. (ii) We introduce ZoKrates, a toolbox to specify, integrate and deploy such off-chain computations. It consists of a domain-specific language, a compiler, and generators for proofs and verification Smart Contracts. ZoKrates hides significant complexity inherent to zero-knowledge proofs, provides a more familiar and higher level of programming abstractions to developers and enables circuit integration, hence fostering adoption.**

***Index Terms*—ZoKrates, zkSNARKs, off-chain, scalability, privacy**

## I. Introduction

While their disruptive potential is widely recognized, today's blockchains are facing many challenges in multiple dimensions, e.g., political, legal and technical. Among the technical challenges, scalability and privacy are particularly prominent problems that need to be overcome to enable broad adoption of this novel class of systems for real world use cases.

Since blockchain transactions must be validated and processed at every node in the network, any blockchain system experiences limitations in scalability. In addition, all data required for validation and processing needs to be available to all nodes. In public blockchain networks like Bitcoin [1] or Ethereum [2], essentially all data on the network therefore becomes publicly accessible, which describes a conflict whenever confidentiality and privacy of information are mandatory.

To address both scalability and privacy, off-chaining has been suggested [3]. The main idea is to reduce computational efforts and data storage on the blockchain by employing blockchain-external resources. However, key properties introduced by using blockchain technology must not be compromised. More specifically, two objectives can be formulated: (a) increase transaction throughput by reducing the cost associated with transaction validation; (b) through off-chaining, remove the requirement to reveal data to all and to become public but allow data to remain private, yet, private data must still have an influence on the transactions processed on-chain.

Transaction throughput is directly linked to the cost implied for clients for block validation. While only mining nodes spend resources on proof of work and are economically incentivized to do so, every full node in the network needs to process all occurring transactions to ensure validity without receiving compensation. To ensure an upper bound for the cost of this validation process, termination is guaranteed. Bitcoin limits transaction complexity by employing a non-turing-complete scripting language and a block size limit. Ethereum, in contrast, uses the concept of gas as a measure for how much burden is put on clients by a transaction (in terms of computation and storage) and defines an upper block gas limit.

In this paper, we make two main contributions:

1) First, we introduce an off-chain processing model based on non-interactive zero-knowledge proofs to improve on transaction throughput and privacy for blockchain systems. In this model, computations are performed off-chain and afterwards the result is written to the blockchain. With the result, a proof attesting correctness of the off-chain execution is supplied. This is then validated on-the blockchain. To realize the benefits mentioned above, the proof system needs to fulfill two important criteria: a) Proof verification needs to be faster than (re-)execution of the original computation. b) The proof needs to be zero-knowledge, that means, it should be possible to proof statements on private data without revealing it. Combining a suitable proof system with the off-chain processing model, we provide means to process blockchain transactions in a more scalable and privacy-preserving way.
2) There are suitable cryptographic verifiable computation schemes, but applying them to blockchains is hard. Computations need to be specified in difficult-to-use low level abstractions. On-chain verification is very complex and requires deep knowledge of the employed schemes. As a solution, we introduce ZoKrates, a set of software tools to support the off-chain processing

model above by employing zkSNARKs as a proof system. ZoKrates defines a domain-specific language which allows developers to conveniently specify off-chain computations at a high level of abstraction. This enables them to specify provable computations without having to understand the proof system's low-level programming abstractions. For that purpose, ZoKrates includes a compiler which translates domain-specific code into provable constraint systems. It furthermore assists in executing off-chain programs and generating proofs attesting their correctness. To conveniently enable on-chain verification, ZoKrates supports the export of verification Smart Contracts, which verify the proofs generated off-chain and therefore confirm the off-chain computation's correctness. Our proof-of-concept implementation targets the Ethereum blockchain, but is by design compatible with other blockchain systems.

## II. Background

In this section, we provide relevant background on Blockchains and the zkSNARK verifiable computation scheme as a foundation for the remainder of this work.

*1) Blockchains and Smart Contracts:* In a nutshell, blockchains are distributed peer-to-peer systems which implement a trustless shared public append-only transaction ledger [4]. Bitcoin, the first implementation of such a system, was proposed in 2008 by Satoshi Nakamoto [1]. The goal was to create a decentralized digital currency which could not be controlled by a central authority and would enable peer-to-peer value transfer without relying on a trusted intermediary. For that, the Bitcoin protocol combines transactions secured by asymmetric cryptography with a consensus algorithm to decide on the transaction order within a network. Peers validate individual transactions by checking cryptographic signatures. In addition to that, however, a global order of the transactions has to be decided on to prevent double spending of digital funds. Since traditional consensus protocols [5] do not scale as required, the proof-of-work consensus protocol was developed as a main innovation of the Bitcoin system. This protocol allocates the right to add transactions to the network in proportion to the computational effort spent by a peer to secure the network. Since directly agreeing on a transaction order is expensive, multiple transactions are grouped into a block for efficiency. These blocks are then ordered by consensus. Each block references its predecessor, which implies a chain data structure – the blockchain.

Extending Bitcoin's idea of peer-to-peer value transfer, Ethereum, a trustless computing platform, was proposed in 2014 [2], [6]. Ethereum adds a turing-complete and stateful programming language to the blockchain idea, which enables the execution of complex code without trusting a server or central party. Trust is replaced by validating each program execution on every peer in the network and agreeing on an outcome. This idea is sometimes referred to as a *world computer*, since it can be seen as one global state machine with a user programable state transition function. The programs that define the system's state transition function and are executed in a trustless and tamper-proof manner in the network are called Smart Contracts. Smart Contracts need to be deterministic as otherwise peers could disagree on the results of valid executions, which would lead to inconsistent state. Therefore, e.g., filesystem and network access are not permitted. Due to the redundant execution of state transitions on every node in the network, operations blocking nodes need to be forbidden to ensure liveness in the network (e.g., infinite loops, long-running transactions). While Bitcoin solves this problem by reducing its instruction set to not allow such operations, Ethereum's virtual machine is turing-complete. Hence, preventing infinite loops reduces to the undecidable halting problem, which is why Ethereum introduced the notion of gas: Every operation is assigned a cost and an initial endowment is specified for the invocation of a Smart Contract function. Since the endowment will be used up at some point, the execution is guaranteed to halt even for infinite loops. A block gas limit specifies an upper bound for the complexity of operations a node has to execute to validate a block, which also defines an upper complexity for operations that can atomically be performed on the blockchain.

*2) zkSNARKs:* Verifiable computation schemes enable computationally weak verifiers to outsource computations to untrusted provers who then perform the computation and return the result including a proof that the result is correct. ZkSNARKs (zero-knowledge Succint Non-interactive ARguments of Knowledge) are a set of verifiable computation schemes with the following key properties:

- Proofs are short and non-interactive, i.e., a prover can convince a verifier with one message.
- A prover can use private information during proof generation and the verifier learns nothing about that information.
- Verification cost is independent of the computational complexity of the input.

ZkSNARKs were introduced in [7] and then further optimized [8], eventually resulting in the scheme employed in this work [9].

The zkSNARK verification scheme is defined on Quadratic Arithmetic Programs (QAPs), an abstraction based on polynomials. There are two slightly higher level abstractions commonly used to specify circuits, Arithmetic Circuits and Rank-1-Constraint-Systems (R1CS). A mapping of Arithmetic Circuits, i.e., circuits composed of additions and multiplications, to QAPs is given in [7]. A similar mapping for Rank-1-Constraint Systems is defined in [9]. R1CS encode a program as a set of conditions over its variables so that correct execution equals finding a satisfying variable assignment. Such an assignment is called witness. Due to the transformability of arithmetic circuits into R1CS and QAPs, programs specified in any of these abstractions are often referred to as circuits.

As an initial step, ZkSNARKs require a trusted one-time setup to be performed in order to create a common reference string (CRS) used during proof creation and verification (often split up into separate proving and verification keys).

## III. Off-chaining Computations

Taking computation and data off the blockchain can have significant benefits for performance, cost and privacy [3]. While anchoring cryptographic hashes to the blockchain showed to be a suitable way of referring to off-chain data [10], [11], moving computations from the blockchain to external nodes without compromising its properties is a more challenging problem.

Delegating computation to arbitrary nodes which publish results to the blockchain can have two main benefits when compared to on-chain execution:

1) A delegate node can use private information during execution of a computation without publicly revealing it. This is not possible for on-chain computations, since they are performed redundantly on all nodes in the network which therefore need to be in possession of all data involved.
2) Ideally, a delegate would only write the result of a computation to the blockchain, which would highly increase efficiency as the only operation performed redundantly by each node would be storing the result. This would largely increase throughput, as many more state transitions could happen in one block.

Hence, off-chain computations could improve privacy and scalability in blockchain networks.

In the above model, however, the delegate node needs to be trusted, which violates the blockchain's key property of trustlessness. To solve this issue, we propose to employ a verifiable computation scheme, so that the delegate node becomes a prover and can publish a proof attesting that the computation was executed correctly along with the result.

For this to be viable in practice, the verifiable computation scheme needs to have the following key properties:

- Short proofs and cheap verification.
- Zero-knowledge, i.e., the verifier learns nothing but the fact that the computation was executed correctly.

Using such a scheme, privacy benefits can immediately be realized by using the zero-knowledge property. This way, a prover can, for example, prove knowledge of a secret without revealing it. As an example, we provide an implementation of Sudoku in section IV-A where a prover can prove that she knows a valid solution for a Sudoku puzzle without showing anything but the fact.

For a result to be accepted in a blockchain network, the accompanying proof has to be verified first. The cost of this verification replaces the cost of on-chain execution of the computation in the first place. Thus, to realize scalability benefits using a VC scheme, on-chain verification needs to be cheaper than on-chain execution of the computation in the first place (for benchmarks see section V-A).

Blockchains, as consensus computers, try to provide a consistent global state. State transitions are currently performed by redundantly executing state transition functions on every node. Employing the mechanism described before, a system where state transitions are not computed, but only validated on chain can be designed as depicted in Fig. 1.

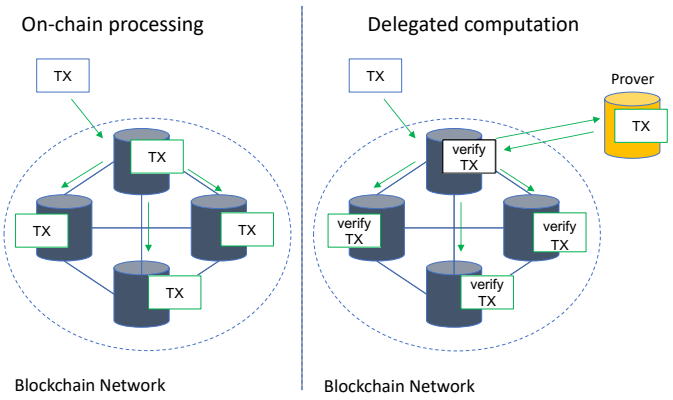


Fig. 1. Off-chaining computation.

Generating a proof is usually the expensive step in VC schemes. Here, the asymmetry in on-chain and off-chain execution environments benefits efficiency: The cheap step is performed redundantly on-chain, while the expensive step can happen in an off-chain environment on powerful hardware without redundancy.

In section IV we introduce ZoKrates, a blockchain-agnostic implementation of these ideas for the zkSNARK verifiable computation scheme. ZkSNARKs fulfil the requirements for our VC scheme and have constant verification cost independent of the off-chain computations' computational complexity. This theoretically allows the on-chain verification of arbitrarily complex computations despite the existence of a block gas limit. The proof-of-concept implementation uses the Ethereum blockchain, but our system can be used with any blockchain supporting the necessary cryptographic primitives (for details, refer to section V).

## IV. ZoKrates Implementation

ZoKrates is designed to support the whole process from specifying a provable program to generating a proof and verifying its correctness on a blockchain. The code is fully open source and available on GitHub[1]. In this proof-of-concept implementation, we use the Ethereum blockchain for proof verification.

As an overview, all steps involved from specifying a computation to verifying its execution on chain are depicted in Fig. 2.

First, a developer specifies a ZoKrates program in a domain-specific language (DSL) further introduced in section IV-A. The program is then compiled into flattened code which is an abstraction that is easily convertible into R1CS and hence compatible with zkSNARK proof systems. Based on the flattened code, a setup phase is performed which results in two public keys, a long proving key and a short verification key.

[1]https://github.com/JacobEberhardt/ZoKrates

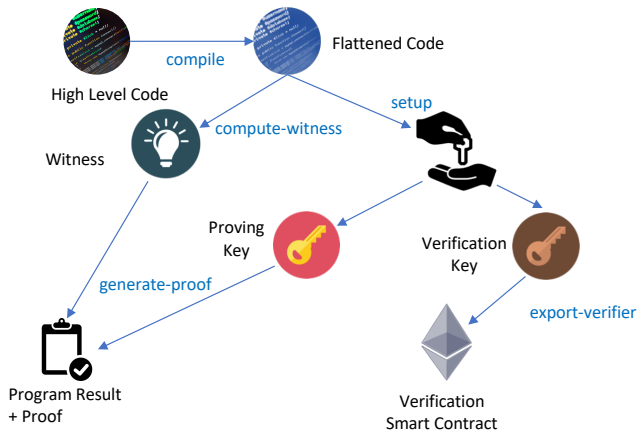


Fig. 2. Full ZoKrates Process.

Based on the verification key, a verification smart contract is generated which can then verify proofs on the blockchain. To generate a proof, a prover first executes the flattened program which is equivalent to finding a solution to the flattened program via the "compute-witness" step. She then uses the proving key and the solution to create a proof and sends it to the verification contract to verify its correctness. Each step will be discussed in more detail in the following sections. An architectural overview of our implementation is depicted in Fig. 3.

### A. The ZoKrates Language

As explained in II, zkSNARK verifiable computation schemes expect programs to be supplied in mathematical equations such as Quadratic Arithmetic Programs (QAPs) [7] or Rank-1-Constraint-Systems (R1CS) [12]. While these abstractions are suitable to build verification schemes, specifying complex computations that way is hard and inconvenient. Therefore, we define a high level domain-specific language which can then be translated into R1CS to enable developers to specify off-chain computations in a way that is more convenient and closer to what they are used to.

The ZoKrates language is a simple imperative language with static scoping designed to be easy to use and at the same time translate into provable constraint systems with very little overhead. Listing 1 a shows an excerpt[2] from a program which proves knowledge of a solution for a Sudoku puzzle without revealing it[3].

The ZoKrates toolchain is also compatible with other circuit generation languages (see section VII) and does not require programs to be specified in its DSL.

*1) Code Structure:* The main entry point of a ZoKrates program is the main function. This function can have public and private inputs as arguments and returns at least one value. While public inputs become public information when a proof attesting the correct execution of the program is sent to the blockchain, the private inputs are not published and will remain the prover's secret. Arbitrary functions can be defined and called. Functions have their own static scope, and a function's definition has to occur before it is called.

```
// Sudoku of format

// | a11 | a12 || b11 | b12 |
// -------------------------
// | a21 | a22 || b21 | b22 |
// =========================
// | c11 | c12 || d11 | d12 |
// -------------------------
// | c21 | c22 || d21 | d22 |

def checkEquality(e11,e12,e21,e22):
counter = if e11 == e12 then 1 else 0 fi
counter = counter + if e11 == e21 then 1 else 0 fi
counter = counter + if e11 == e22 then 1 else 0 fi
counter = counter + if e12 == e21 then 1 else 0 fi
counter = counter + if e12 == e21 then 1 else 0 fi
counter = counter + if e21 == e22 then 1 else 0 fi
return counter

// returns 0 for x in (1..4)
def validateInput(x):
return (x-1)*(x-2)*(x-3)*(x-4)

// variables naming: box'row''column'
def main(a21, b11, b22, c11, c22, d21, private a11,
    private a12, private a22, private b12, private
    b21, private c12, private c21, private d11,
    private d12, private d22):

// validate inputs
0 == validateInput(a11)
[...]

counter = 0 // globally counts duplicate entries in
    boxes, rows and columns

// check box correctness

  // no duplicates
  counter = counter + checkEquality(a11,a12,a21,a22)
  [...]

// check row correctness
  counter = counter + checkEquality(a11,a12,b11,b12)
  [...]

// check column correctness
  counter = counter + checkEquality(a11,a21,c11,c21)
  [...]

// assert counter is 0
counter == 0

return 1
```

Listing 1. Suduko Verification Example

*2) Types:* The basic data type in ZoKrates is a prime field element. This is an positive integer modulo a fixed prime number. In our implementation, a prime number of size 254 bits is used; due to this size the developer can think of a prime field element as a simple unsigned integer type in most cases. To represent binary types, field elements can simply be restricted to values 0 and 1.

[2]Full source code available at https://github.com/JacobEberhardt/ZoKrates/\blob/master/examples/sudokuchecker.code

[3]Ropsten Testnet Verification Contract available at https://ropsten.etherscan.io/address/0x1afcca25176e7d899998e1a57a382ee5061e8efc

*3) Operators:* Common arithmetic operators are supported (e.g., +,−,∗,/) and have positive integer semantics unless there are overflows or divisions with remainders. The usual comparison operators are supported (e.g., ==, <=). The == operator can furthermore be used to define assertions. Boolean operators are not natively defined, but can easily be simulated by arithmetic expressions wrapped in functions, e.g., $AND(a,b) = ab$, and $OR(a,b) = 1 - (1-a)(1-b)$.

*4) Control Flow:* For-Loops are supported, but require an upper bound of iterations to be specified, as they are unrolled into a list of statements during the compilation process (refer to IV-B) to become compatible with the verifiable computation scheme. For the same reason, recursion is not supported. As common in imperative languages, If-Else-Statements enable conditional assignments.

### B. The ZoKrates Toolbox

After a program has been written in the ZoKrates DSL, several steps need to be performed until a proof attesting its correct execution can be sent to and verified on the blockchain. We provide several tools to support this process.

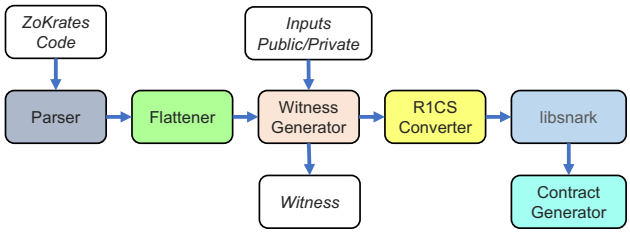


Fig. 3. ZoKrates Architecture.

*1) Compiler:* The compiler, consisting of the parser and the flattener in Fig. 3, translates DSL code into so-called flattened code. Flattened code consists of a a list of variable definitions and assertions of the following form:

```
def (c_1*var_1 + c_2*var_2 + ... + c_n*var_n) {=|==}
      (a_1*var_1 + a_2*var_2 + ... + a_n*var_n) * (
    b_1*var_1 + b_2*var_2 + ... + b_n*var_n)
```

Listing 2. Flattened Code Structure

The left side is linear, the right side comprises a multiplication of linear terms.

This format was chosen since it directly and efficiently translates into abstractions that verifiable computation schemes use, e.g., R1CS and arithmetic circuits. Furthermore, it maintains witness derivability, i.e., allows to calculate all variables in the flattened code from the inputs or previously calculated intermediate results. For witness derivability, the compiler can insert arbitrary DSL statements that instruct the witness generator on how to compute a certain variable's value. These witness-generation directives are prefixed by "#" and only used for witness generation. They are ignored in the transformation of flattened code to R1CS. An exemplary use of directives in the context of equality checks is depicted in 3.

```
# sym_1 = if sym_0 == 0 then 0 else 1 fi
# sym_2 = if sym_0 == 0 then 1 else (1 / sym_0) fi
sym_1 == (sym_0 * sym_2)
```

Listing 3. Witness-Generation Directives

*2) Witness Generator:* Before a proof attesting correct execution can be generated, the program first has to be executed. In this context, that means finding a satisfying variable assignment for a flattened program. Such a variable assignment is referred to as a witness and can then be used during proof generation. A witness for a given program can be found using the witness generator which acts as an interpreter for flattened code. It takes a program's private inputs as parameters, executes the flattened code and stores all variable values, thus finding a witness in the process.

*3) Circuit Import:* In order to be able to leverage hand-optimized circuits for certain computations or integrate with other circuit generation tools (see section VII), R1CS can be imported into semantically equivalent flattened code. However, while fully supported for ZoKrates DSL code, witness generation for imported programs cannot generally be supported and requires users to provide that information.

*4) Setup and Proof Generation:* As explained in section VII, the zkSNARK verifiable computation scheme requires a one-time trusted setup to be performed. It is trusted in the sense that who executes it needs to forget private information used in the process. If that information is not forgotten, the owner could potentially create fake proofs. However, the zero-knowledge property of other proofs would not be impaired. To address this issue, we discuss mechanisms to remove this trust in section VI. The setup phase takes a program and its public arguments as an input and generates a proving and a verification key which can then be used to create and verify proofs [13]. These keys can then be reused an arbitrary number of times. For setup and proof generation, the ZoKrates implementation uses `libsnark`[4], a cryptographic library which implements zkSNARK schemes [14].

*5) Proof Generation:* After a witness has been found and the proving key generated, a proof attesting the correctness of the witness and with that program execution can be created. The resulting proof is very short (see V for details) and can thus be efficiently sent to the blockchain via network.

*6) Contract Generator:* Based on a verification key, we generate a Solidity Smart Contract which can verify proofs on the blockchain. The generated contract has a very simple base structure as shown in Listing 4 and can be customized depending on the concrete use case the on-chain verification is leveraged in. It can be compiled with `solc`[5], the official Solidity compiler, and then be deployed to Ethereum. The `verifyTx` function takes a program's public inputs and results as parameters as well as the previously generated proof attesting correctness of its execution. It returns a boolean value indicating verification success and hence the correctness of

[4]https://github.com/scipr-lab/libsnark
[5]https://github.com/ethereum/solidity

the off-chain execution. If successful, the program's result is confirmed and can be used in further processing on or off the blockchain.

```
contract Verifier {
    // library wrapping precompiles
    using Pairing for *;

    // verification key data structure
    struct VerifyingKey {
      ...
    }

    // proof data structure
    struct Proof {
      ...
    }

    event Verified(string);

    // Accepts proof and public inputs and checks
        proof validity.
    // Proof: a, a_p, b, b_p, c, c_p, h, k.
    function verifyTx(
            uint[2] a,
            uint[2] a_p,
            uint[2][2] b,
            uint[2] b_p,
            uint[2] c,
            uint[2] c_p,
            uint[2] h,
            uint[2] k,
            uint[] inputs_and_res
        ) returns (bool r) {
      ...
    }
```

Listing 4. Solidity Verification Contract Structure

## V. Evaluation

In this section, we give an overview of the performance and cost related to the steps in the ZoKrates process described in section IV.

The on-chain verification was benchmarked on the Ethereum Ropsten Testnet[6]. This benchmark is certainly the most significant in this context, since the on-chain verification cost decides on whether throughput can be gained or costs be saved by off-chaining a computation. This does not account for privacy benefits, which need use case dependent evaluation and cannot be benchmarked. The Ethereum precompiled contracts [15] used for proof verification[7] (Point Addition, Scalar Multiplication, Pairing), are defined for a Barreto–Naehrig elliptic curve [16] with 128 bits of security. Hence, ZoKrates was configured to use this curve too.

The off-chain benchmarks were performed on an Intel Core i5-3210M @2.5 GHz machine with 8 GB RAM and HDD. We divide the off-chain steps into two categories:

1) One-Time steps: Executed only once for a given program.
2) Repeated steps: Executed for every execution of a given program.

In both cases, we show that the performance bottlenecks are the steps which internally use `libsnark`. Thus, [13] gives good performance estimates.

As verifiable computation scheme, we use the zk-SNARK construction for R1CS from [9], as implemented in `libsnark`, which is a slightly modified version of the Pinocchio system [8]. Thus, for this scheme and the Barreto-Naehrig curves, proofs consist of 8 elliptic curve points and have a size of 288 bytes. For the verification of a proof, five pairing checks have to be performed, as well as $\#inputs + 4$ point additions and $\#inputs$ scalar multiplications, where $inputs$ are supplied with the proof.

### A. On-chain Verification

Ethereum limits a block's complexity by assigning a cost denominated in gas to each operation (see II). Currently, the block gas limit is ~8 million[8]. New blocks are generated every 14 seconds on average[9].

For the on-chain verification of a proof, ~1.6 million gas are required. As shown in Fig. 4, the gas cost rises linearly with the $\#inputs$ supplied with the proof. Note, that the verification cost is independent of the complexity of the program the execution of which is verified. The deployment cost of the contract is ~1.5 million gas.

As described in more detail in section VI, the cost could likely be lowered by adopting an optimized verifiable computation scheme.

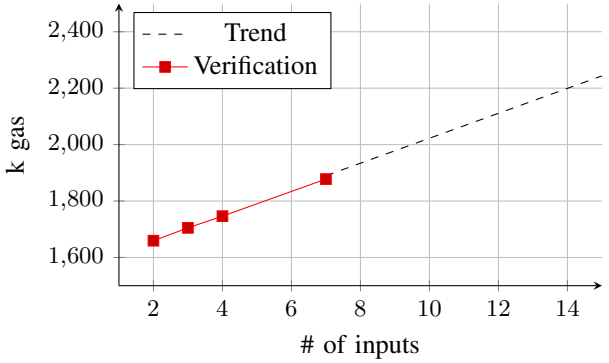


Fig. 4. Verification cost per # of inputs

### B. One-time Steps

Compilation, setup and contract generation are steps that are performed only once for a specific program. Then, the generated artifacts, flattened code as well as proving and verification keys, can be reused. While the compilation and contract generation steps are part of the core ZoKrates implementation, the setup step uses `libsnark` internally. For that reason, the measurements regarding the setup are given primarily to provide some intuition; for detailed performance benchmarks and predictions, we refer to [13]. Fig. 5 shows that compilation is always faster than the setup. Herein, the constraint system

[6]https://github.com/ethereum/ropsten
[7]https://github.com/ethereum/EIPs/pull/212
[8]https://ethstats.net/, accessed 13.03.2018
[9]https://etherscan.io/chart/blocktime, accessed 13.03.2018

with 695 constraints was compiled from the Sudoku example from Listing 1. Refer to the discussion (VI) for how the setup could be distributed to eliminate the need to trust a single party. The contract generation step has a short and constant execution time and was thus not included in the benchmarks.

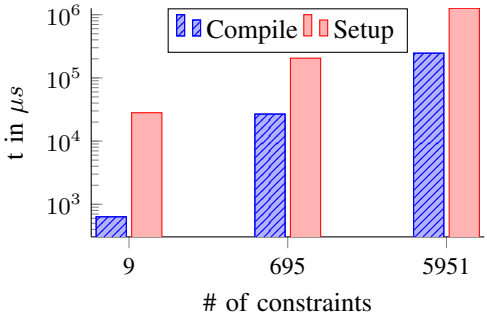


Fig. 5. Average runtime of one-time steps (100 executions).

### C. Repeated Steps

Witness generation happens every time a program is executed and in turn requires a new proof attesting the correctness of that execution to be generated.

As can be seen in Fig. 6, the bottleneck is proof generation. Again, the proof generation time is mainly dependent on `libsnark` and we refer to [13] for detailed benchmarks.

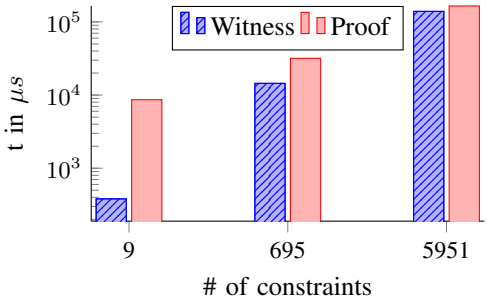


Fig. 6. Average runtime of repeated steps (100 executions).

## VI. Discussion & Outlook

Our ZoKrates prototype shows that on-chain verifications of off-chain program executions is usable in today's blockchain systems to improve users' privacy and increase scalability.

Since the ZoKrates implementation uses zkSNARKs, it inherits the need for a trusted setup phase. To address this issue, a multi-party computation protocol has been proposed [17] which distributes the setup phase over $n$ nodes and is secure as long as there is at least one honest participant. This immensely reduces the trust requirements in the generation of proving and verification key. An adapted version [18] has been used for the distributed setup of the zCash blockchain. In future work, we plan to integrate a blockchain-based version of this protocol with ZoKrates in order to enable convenient decentralized setups with many independent participants. Another more efficient proposal [19] specifies a two-round multi-party setup, where the first round is circuit independent and is then reused during a circuit specific second round.

Using Groth's zkSNARK construction [20], verification cost and proof size of ZoKrates programs could be further reduced. A proof could become as small as 127 bytes for the Barreto-Naehrig curve and require only one pairing check. Switching to a more efficient elliptic curve, e.g, JubJub[10] could further increase performance, but would require the curve to be supported by the verifying blockchain, which is currently not the case in Ethereum.

Another promising direction is the use of verifiable computation schemes that do not require a trusted setup at all. There are two recent proposals realizing such a system. Unlike the constant size zkSNARK proofs, Bulletproofs [21] have a proof size logarithmic in the number of multiplication gates in the circuit verifying a witness. Also, verification is much more expensive. ZoKrates could support them seamlessly as they build on the same circuit abstraction. ZkSTARKs (zero-knowledge Succinct Transparent ARguments of Knowledge) [22] proof sizes and verification costs are even larger and share no circuit abstraction with ZoKrates. Although these schemes are not practical for general purpose on-chain verification yet, they could potentially mature to be a superior alternative to zkSNARKs with further research.

For better usability, language extensions such as more complex types and cryptographic functions to simplify common tasks in circuit construction could be supported.

## VII. Related Work

To our knowledge, this is the first work to consider and support the whole process from program specification over off-chain execution to verification on a blockchain.

The Truebit project [23] shares a similar spirit in that it off-chains computations to third parties and writes results back to the blockchain. Instead of supplying a proof with the results, as ZoKrates does, the result is optimistically accepted and Challengers are economically incentivized to supply fraud proofs in case of invalid results.

Zerocash [24] uses zkSNARKs to realize an anonymous payment scheme on top of blockchains and led to the creation of Zcash[11].

The Hawk project [25] proposes a smart contract system which retains privacy for financial transactions on the blockchain. Therefore, it introduces a set of cryptographic primitives based on zkSNARKs. The scope of the system is much more specific than ZoKrates and to date, no implementation is available.

For circuit specification - in our implementation done in the ZoKrates language - and generation, there are several software tools which perform a similar task, but build on different input abstractions. Pinnocchio [8] proposes to use a subset of C for circuit specification and requires a certain source code layout.

[10] https://z.cash/technology/jubjub.html
[11] https://z.cash/

We chose to design a language optimized for translation into the target abstraction and only offer low cost abstractions to the developer instead of supporting a subset of another language with unexpected limitations. Snarky is a OCaml-based functional circuit generation language[12] which translates into R1CS. The jsnark[13] library allows the specification of arithmetic circuits directly in Java. All these circuit generators can be used together with ZoKrates, as the underlying abstractions are fully compatible.

TinyRAM [12] and vnTinyRAM [9] support the execution of actual C programs in zero-knowledge, but proof generation is too expensive to be practical and viable in our use case.

## VIII. Conclusion

In this paper, we introduced an off-chaining model for computations based on zero-knowledge verifiable computation schemes to address the challenges of scalability and privacy in blockchain systems.

To support this model, we presented ZoKrates, the first comprehensive implementation of a toolbox to specify, integrate and deploy off-chain computations. ZoKrates consists of a domain-specific language, a compiler, and generators for proofs and verification Smart Contracts for zkSNARK verifiable computation schemes. It hides significant complexity inherent to zero-knowledge proofs and provides a more familiar and higher level of programming abstractions to developers, hence fostering adoption. We provide benchmarking results for our implementation, discuss challenges related to the approach and suggest future research directions.

In summary, we showed that on-chain verification of off-chain program executions is usable in today's blockchain systems to improve users' privacy and increase scalability. Furthermore, we provide tools to support the adoption of this paradigm.

## Acknowledgment

We thank Christian Reitwießner for ongoing discussion and Dennis Kuhnert, Thibaut Schaeffer and Steffen Härtlein for their contributions to the implementation.

## References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," *https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-White-Paper*, 2014.

[3] J. Eberhardt and S. Tai, "On or off the blockchain? insights on off-chaining computation and data," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2017, pp. 3–15.

[4] S. Tai, J. Eberhardt, and M. Klems, "Not ACID, not BASE, but SALT - a transaction processing perspective on blockchains," in *Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, INSTICC. ScitePress, 2017, pp. 755–764.

[5] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014.

[12]https://github.com/o1-labs/snarky

[13]https://github.com/akosba/jsnark

[7] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," Cryptology ePrint Archive, Report 2012/215, 2012, https://eprint.iacr.org/2012/215.

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

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

[10] J. Benet, "IPFS - content addressed, versioned, P2P file system," *CoRR*, vol. abs/1407.3561, 2014. [Online]. Available: http://arxiv.org/abs/1407.3561

[11] V. Trón, A. Fischer, D. A. Nagy, Z. Felföldi, and N. Johnson, "Swap, swear and swindle - incentive system for swarm," 2016.

[12] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "Snarks for c: Verifying program executions succinctly and in zero knowledge," in *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 90–108.

[13] scipr-lab, "libsnark performance information for preprocessing zk-SNARK for R1CS relation," https://github.com/scipr-lab/libsnark/tree/master/libsnark/zk_proof_systems/ppzksnark, accessed: 2018-03-11.

[14] E. Ben-Sasson, A. Chiesa, D. Genkin, S. Kfir, E. Tromer, and M. Virza, "libsnark, 2014."

[15] G. Wood and Y. e. a. Hirai, "Ethereum yellow paper, byzantium version 2018-03-12, Appendix E." *Ethereum Project Yellow Paper*, 2018.

[16] P. S. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *International Workshop on Selected Areas in Cryptography*. Springer, 2005, pp. 319–331.

[17] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza, "Secure sampling of public parameters for succinct zero knowledge proofs," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 287–304.

[18] S. Bowe, A. Gabizon, and M. D. Green, "A multi-party protocol for constructing the public parameters of the pinocchio zk-snark," TR 2017/602, IACR, Tech. Rep., 2017.

[19] S. Bowe, A. Gabizon, and I. Miers, "Scalable multi-party computation for zk-snark parameters in the random beacon model," Cryptology ePrint Archive, Report 2017/1050, 2017, https://eprint.iacr.org/2017/1050.

[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] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Efficient range proofs for confidential transactions," Cryptology ePrint Archive, Report 2017/1066, Tech. Rep., 2017, https://eprint.iacr.org/2017/1066.

[22] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," Cryptology ePrint Archive, Report 2018/046, 2018, https://eprint.iacr.org/2018/046.

[23] J. Teutsch and C. Reitwießner, "A scalable verification solution for blockchains," 2017, https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf.

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

[25] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 839–858.