FACT FORTRESS: An On-Chain Zero-Knowledge Proof Framework for Fact Checking

Piergiuseppe Mallozzi
UC Berkeley
Berkeley, USA
mallozzi@berkeley.edu

Guillaume Lethuillier
Blockdaemon
Paris, France
lethuillier@gmail.com

***Abstract*—FACT FORTRESS is a framework that combines blockchain technology and zero-knowledge proofs to enable trustworthy and private fact-checking. By leveraging proofs of data provenance and auditable data access policies, we ensure the trustworthiness of how sensitive data is handled, and provide assurance of the computations that have been performed on it. In addition, our solution democratizes circuit construction and deployment by providing a circuit compiler that supports various data formats and source authentication, and facilitates the deployment of on-chain verifiers.**

FACT FORTRESS provides a powerful mechanism for preserving the privacy of sensitive data while ensuring accountability and transparency in the actions taken on the data. By enabling on-chain verification of computation and data provenance without revealing any information about the data itself, our solution ensures the integrity of the computations on the data while preserving its privacy.

***Index Terms*—Zero-Knowledge Proof, Block-chain Framework**

I. INTRODUCTION

Zero-knowledge proofs (ZKPs) are a powerful tool for enabling privacy-preserving computations and have a wide range of applications, including authentication, identity management, and data privacy. However, the current deployment of ZKP frameworks faces several challenges, such as the lack of trust in how data is handled, difficulty in constructing proofs, and scalability concerns.

In this project, we propose solutions to address these challenges and tackle three critical issues in ZKP-based applications. First, we address the problem of establishing trust in how data is handled by ensuring transparent and verifiable authenticity and provenance of data, implementing access controls based on data provider policies, and maintaining a public log for transparency. We propose a general statement verification process that includes proof of data provenance by design and an architecture of smart contracts to handle access control policies and the on-chain verification of proofs, providing accountability and transparency on the actions taken with the data.

Second, we address the problem of circuit generation, which involves creating an arithmetic circuit to formulate the desired function to be applied to the data. To simplify this process, our solution provides a user-friendly interface for generating circuits at scale. Our tool accepts a variety of data formats and a library of functions that can be chosen to analyze the data. The tool ultimately compiles down to circuits in the Noir language [1], a newly released Zero-Knowledge framework by Aztec. We evaluate the scalability of our approach and address how it handles data of growing size and circuits of growing complexity.

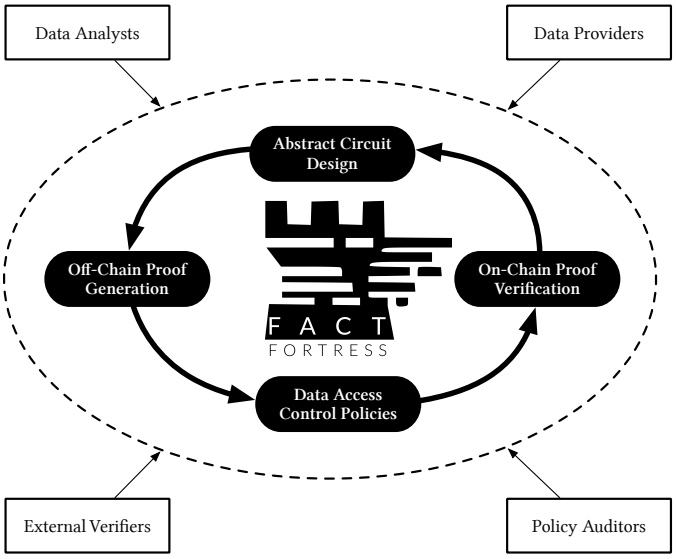

Fig. 1: Overview of the proposed end-to-end solution.

Finally, our project provides a comprehensive solution that covers the entire process from circuit generation to proof generation, while facilitating collaboration among data analysts, data providers, external verifiers, and policy auditors, as illustrated in Figure 1.

We have implemented our approach in a platform named FACT FORTRESS, a blockchain-integrated solution that utilizes ZKPs to enable efficient and trustworthy fact-checking with transparent and verifiable authenticity of data without compromising privacy. The platform is available open-source, and it includes a front-end and a separate tool for circuit compiler (code available[1]).

[1]**Smart-Contracts Dapp**:
https://github.com/pierg/fact-fortress-dapp
Front-end:
https://github.com/pierg/fact-fortress-web
Circuit Compiler Tool:
https://github.com/pierg/fact-fortress-compiler

II. Background

Zero-knowledge proof is a cryptographic technique that allows one party (the prover) to prove to another party (the verifier) that they know a particular piece of information, without revealing that information itself. This can be especially useful in scenarios where sensitive data needs to be kept confidential to ensure the *privacy* and *security* of the data.

The process of zero-knowledge proof involves a prover and a verifier, who must agree on the function to be executed on the data. Let $f(w, x) = y$ be a function on the inputs w and x where w (often called the *witness*) is private and x and y are public. The prover must produce a proof π that convinces the verifier that they know a secret input w such that $f(w, x) = y$.

Figure 2 illustrates the interaction between the prover and verifier to prove a generic claim $f(w, x) = y$ without revealing any information about w. The prover takes as input private data w and public data x and y, and generates a proof π. The verifier takes as input public data x, y, and π, and can either accept or reject the proof based on whether it is valid or not.

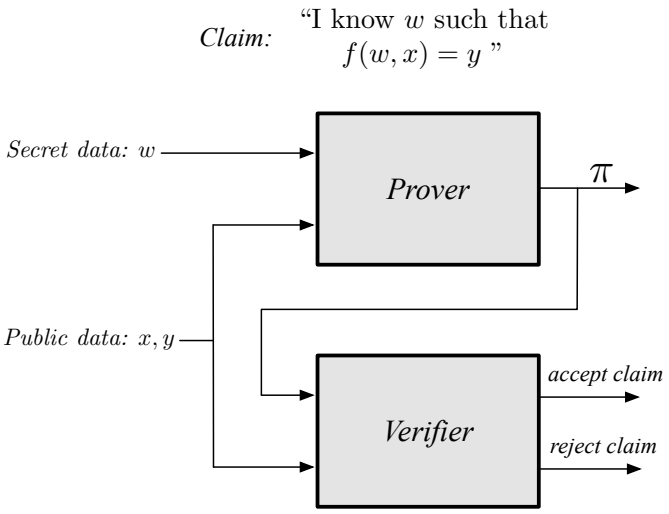

Fig. 2: Generic Prover and Verifier Interaction

One specific implementation of zero-knowledge proof is called zk-SNARKS (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge). It is a type of proof system where the prover produces a succinct and efficient proof of the correctness and the verifier is fast to verify the proof. The main steps to construct a zk-SNARK are:

1) **Arithmetic Circuit**: The function $f(w, x) = y$ needs to be represented as an arithmetic circuit consisting of multiplication and addition gates. This process is done by first converting f into a R1CS (Rank-1 Constrain System), which is then transformed into a series of quadratic arithmetic programs (QAPs) using techniques such as Lagrange Interpolation and Fast Fourier Transform (FFT).
2) **Setup Procedure**: The setup procedure generates the proving and verifying keys S_p and S_v, as well as the public parameters pp. The prover uses S_p to generate proofs, the verifier uses S_v to verify those proofs, and pp defines the mathematical structure used to construct the circuit and is used by all parties. There are three types of setup procedures:
   a) *Trusted setup*, where a trusted party generates (S_p, S_v, pp) for a specific circuit.
   b) *Trusted but updatable setup*, which is similar to the trusted setup but allows the trusted party to update parameters and keys for other circuits.
   c) *Transparent setup*, which generates parameters and keys using a publicly known deterministic algorithm, eliminating the need for a trusted party but may be more computationally expensive.
3) **Prover**: The prover takes as input the private input w, the public data x and y, the proving key S_p, and the parameters pp, and uses them to generate a proof π. This involves constructing a witness polynomial that satisfies the QAP and evaluating it at carefully chosen points, resulting in a proof consisting of two polynomials.
4) **Verifier**: The verifier takes as input the public data x and y, the proof π, the verification key S_v, and the public parameters pp, and uses them to check the validity of the proof. This involves checking that the polynomials in the proof satisfy certain constraints, such as the QAP and the arithmetic relations of the circuit.
5) **Zero-Knowledge Property**: Finally, if the proof is valid, the verifier accepts the proof without learning anything about the private input w or the computation of $f(w, x) = y$, except for the fact that the computation is correct.

In summary, the setup procedure generates the proving and verifying keys and the public parameters, while the prover constructs a proof, and the verifier checks the validity of the proof. If the proof is valid, the zero-knowledge property guarantees that the verifier learns nothing about the private input.

III. Challenges of Deploying Zero-Knowledge Proof Frameworks

Several challenges hinder the adoption and deployment of ZKP frameworks in real-world applications. In this section, we discuss challenges of trust and adoption.

A. Trust

Trust is a critical challenge that must be addressed before zero-knowledge proof (ZKP) frameworks can be widely adopted in the real world. This is particularly important when working with sensitive data, where proving and verifying mechanisms should be transparent and the access to the sensitive data be regulated.

One challenge related to trust is ensuring the authenticity and provenance of the input data used by the prover and the verifier. If the input data used by the prover is compromised or tampered with, the resulting proof may be invalid, compromising the overall security of the system. On the other hand, the verifier needs to be confident that the public data used in the verification process is valid and has not been manipulated to produce a false result.

To address trust-related challenges, one approach is to include the input data as part of the trusted setup procedure. However, this requires trust in the entity that generated the public parameters. This technique can be used to deploy a trusted prover and verifier with fixed trusted data. However, it does not allow for the same prover and verifier to be used on other data, as the setup procedure would need to run again, and a new prover/verifier should be created.

Another approach is multi-party computation (MPC) [2], which allows multiple parties to jointly compute a function without revealing their inputs to each other. We can use MPC to collectively create the input data from several trusted entities. This technique can provide a more robust solution for generating trusted input data and can be used to deploy ZKP systems on a broader range of data without requiring a new trusted setup for each case, however it requires the coordination of multiple trusted entities.

B. Adoption

The adoption of ZKP frameworks has been limited, in part, due to the complexity of expressing functions in terms of arithmetic circuits, which are a crucial component of many ZKP protocols. However, over the recent years, several tools and languages have been developed to facilitate the use of ZKP frameworks.

Hardware Description Languages (HDL) such as Circom [3] provide a way to describe circuits in a low-level format that can be compiled to arithmetic circuits. Libraries such as Arkworks [4] provide modular building blocks that can be combined to create custom ZKP systems.

Finally, there are several programming languages designed to make it easier to implement ZKP frameworks in real-world applications. For example, Zokrates [5] is a popular open-source toolkit that allows developers to write programs in a high-level language and compile them into arithmetic circuits. Noir [1] is a Rust-like language that provides tools for ZKP construction and verification. Leo [6] and Cairo [7] are another programming languages that are designed to allow easy expression of complex circuits in a high-level format.

Even with the availability of Domain-Specific Languages (DSLs), Programming Languages (PLs), and libraries, expressing complex functions in terms of arithmetic circuits remains a challenge for non-domain experts.

IV. Fact Fortress

In the following sections we provides an overview of our proposed solution named Fact Fortress. In Section V we address the problem of trust, providing general circuit design and smart-contract architecture that has data privacy and autheticity at its core. In Section VI we address the problem of adoption, we propose abstraction layers on top of existing frameworks that facilitate the circuit and data specification.

V. Trust by Design

In our approach, the trust of the data is embedded in the design of each circuit as well as in the overall framework.

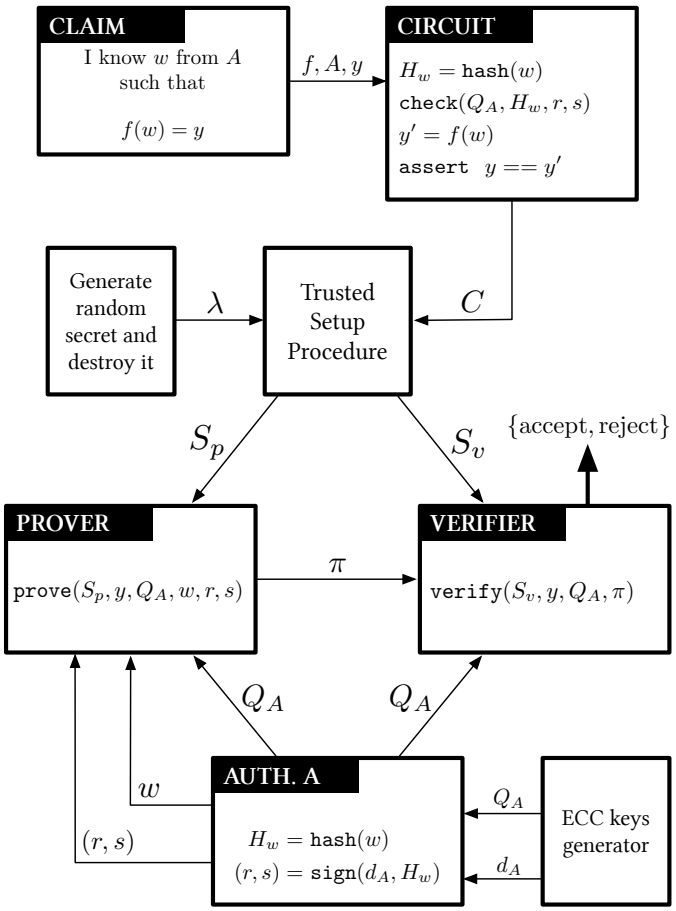

Fig. 3: Statement Verification Process using ZKP

Specifically, each circuit is designed to certify the proof of provenance of the data, ensuring that the input data is coming from authenticated sources. Moreover, the overall architecture framework is designed to facilitate the exchange of data among different parties in a regulated way implementing access control policies. Finally, we ensure accountability through a distributed ledger that stores all interactions among parties in a transparent and auditable manner.

By employing this our approach, the trustworthiness of the ZKP framework is enhanced, making it more suitable for handling sensitive data and real-world applications.

A. Proof of Provenance

Our approach involves embedding a "*proof of provenance*" alongside the proof of statement in each circuit. Figure 3 provides an example of how we prove the truthfulness of a generic statement "*I know w from A such that $f(w) = y$*", without revealing any information about w. The proof is constructed in two steps:

Step 1: Proof of provenance, which proves that:

- w originated from the authority A
- w has not been tampered with or altered in any way

Step 2: Proof of statement, which proves that:

- The result of $f(w)$ is equal to the claimed result y

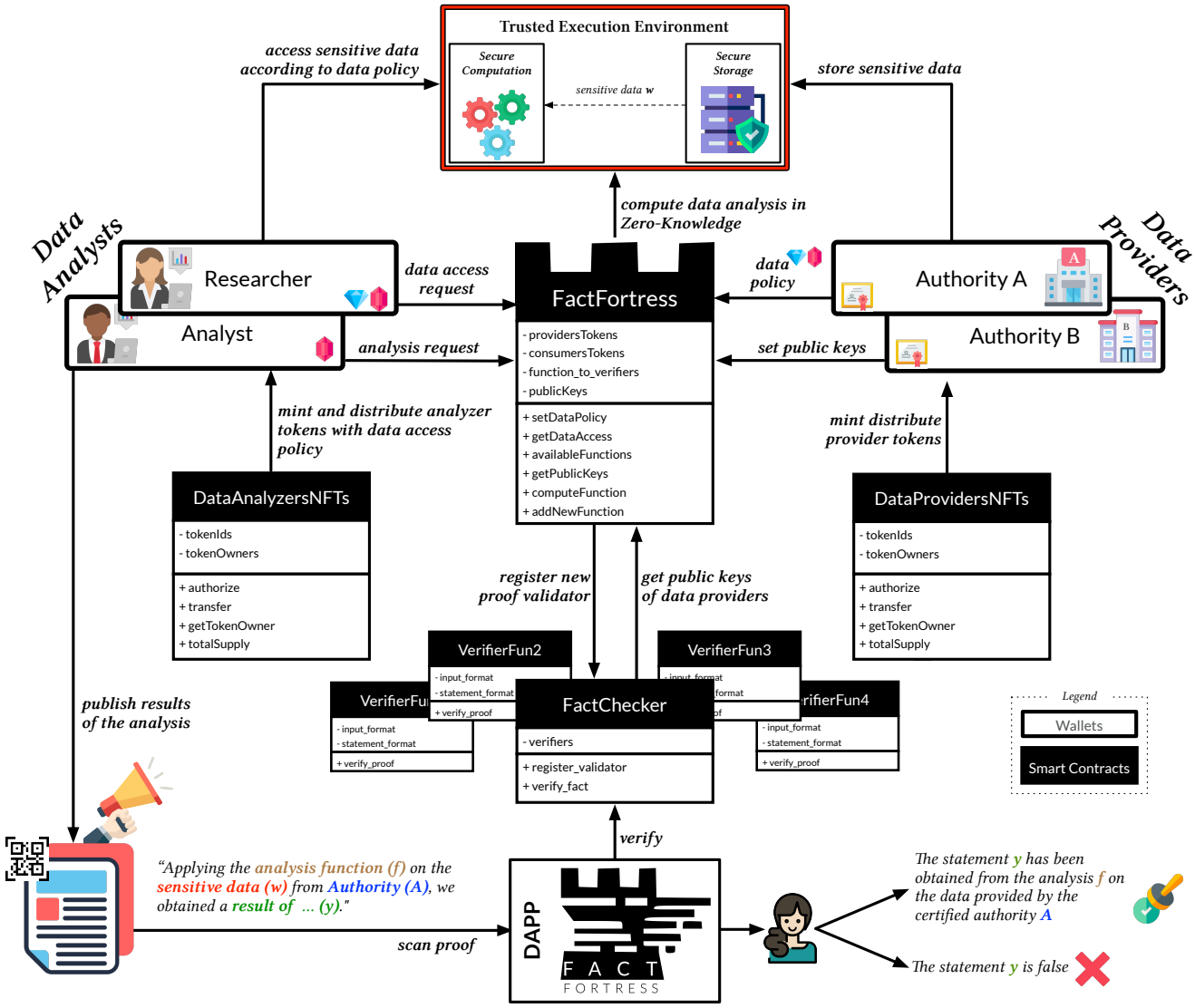

Fig. 4: Smart Contracts Architecture on Blockchain

In addition to the proofs provided in ZK by the circuit, our framework also ensures that '*The function f has been faithfully translated into an arithmetic circuit*' by providing a library of fixed functions that the prover can choose from (Section VI) and that '*The proof π was generated by the function f claimed by the prover*' (Section V-D).

B. Data Access Policies

To provide secure and regulated data access, our framework incorporates data-access policies. Each data provider can define their access policies, which include the type of data that can be accessed, the duration of access, and any restrictions on data usage.

Data analysts are granted granular data access permissions by attributing a Non-Fungible Token (NFT). Then, they can request access to the data at any time by providing proof of compliance with the access policies (*i.e.*, the possession of the token, and the associated data access permissions). They are granted temporary access to the requested data if the proof is valid.

Our access control mechanism ensures that only authorized data analysts can access the data and that they can only use the data according to the specified policies. The NFTs also ensure that data analysts cannot access data that they are not authorized to use or extend their access beyond the specified duration.

By using smart contracts, we can guarantee the integrity and transparency of the data access policies and their enforcement, which is essential in sensitive domains such as government or finance.

C. Accountability

Our framework leverages the use of a transparent and publicly verifiable ledger, such as a blockchain, to ensure that data usage is logged and monitored in a transparent and immutable way. This creates an audit trail that tracks all

data access and usage on the ledger, allowing data providers to monitor how their data is being used and to detect any unauthorized access or usage. By ensuring the integrity and transparency of data access and usage, our framework provides a robust and trustworthy solution for data sharing and analysis.

D. Smart Contract-Based Architecture

To ensure transparency and accountability in data access, our FACT FORTRESS framework incorporates smart contracts into its architecture. The overall architecture of the framework is depicted in Figure 4. By incorporating smart contracts into the architecture, we ensure that all parties involved in the data analysis adhere to the specified policies, and that the public results of the analysis can be trusted.

The framework allows certified *data providers* to securely store their sensitive data and set data access policies on how the data must be handled. Data analysts can request access to the data based on these policies to perform an analysis and compute the ZKP locally. Alternatively, they can delegate the data analysis to FACT FORTRESS, which returns the zero-knowledge proof of the computation and the result directly to them.

The types of analyses that can be performed are defined by a library of functions that can be computed on data of any form. For each function, we deploy a verifier on-chain, which anybody can use to validate a proof. When a proof is submitted for validation, FACT FORTRESS dispatches it to the correct verifier, which checks its validity.

The overall process is as follows:

1) Our circuit compiler generates all the necessary data structures and circuits, ready to be proved in zero-knowledge, and produces a new on-chain verifier for each function in the library.
2) To **verify a proof**, the verifier (i.e. the *Fact Checker* contract) receives the public keys of the authority and uses them as public inputs. The verifier ensures that:
   a) The proof was generated by the function claimed by the data analyst (i.e., the prover).
   b) The data used to generate the proof has not been tampered with by the analyst and comes from a certified data provider.
   c) The claimed result is the correct result of the function applied to the data.
3) To **produce a proof**, the analysts can:
   a) Securely access data via NFT-enabled access policies and produce the computation and proof themselves.
   b) Directly delegate FACT FORTRESS to perform authorized functions in Zero-Knowledge in a trusted execution environment. The smart contract returns the result of the computation together with the proof.
4) Analysts can confidently publish the results together with the proof.
5) Anyone can read the results, download the proof, and publicly verify it on-chain.

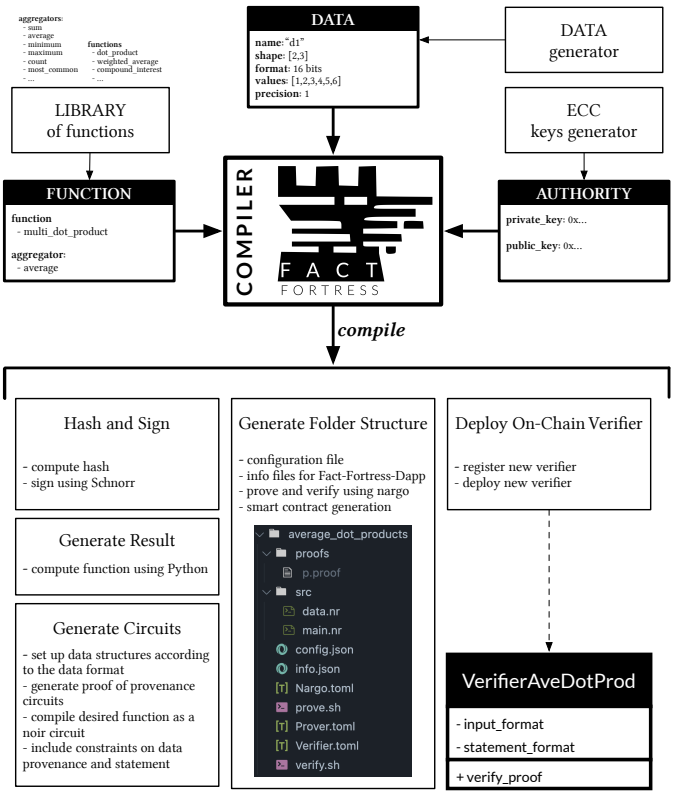

Fig. 5: Circuit Compilation Process

VI. DEMOCRATIZING ZKP CIRCUITS

To increase adoption of zero-knowledge proof (ZKP) frameworks and enable developers with limited experience to implement functions in real-world applications, more user-friendly interfaces and higher-level abstractions are needed to abstract away the low-level details of arithmetic circuits.

One approach to achieve this is through the development of tools for automatic circuit synthesis and by providing easy access for users to utilize them to verify their statements. In the following section, we describe how we have achieved this within our FACT FORTRESS framework.

A. FACT FORTRESS *Circuits Compiler*

We have implemented such a tool by building abstraction layers on top of the Noir framework [1]. Our abstractions provide simple Python APIs that allow developers to express complex computations more easily, without requiring them to understand the underlying arithmetic circuits.

Our tool (code available[2]) enables the automatic generation of arithmetic circuits from high-level abstractions, making it easier to implement zero-knowledge proof (ZKP) protocols in their applications.

Our library provides clear and abstract APIs that allow users to specify the data format, the function to be performed by the circuit, and the authority that provided the data. The library

[2]Configuration file: https://github.com/pierg/fact-fortress-circuits/blob/main/circuits/average_dot_products/config.json

compiles down from Python API to JSON configuration file and ultimately parses the JSON and compiles the fully functioning circuit in Noir as shown in Figure 5.

In order to produce a circuit, our library takes as input:

- *data*: arrays of any size and shape, elements can be integers, string or double. Doubles are quantized by our library with the specified precision.
- *authority*: private keys of the authority providing the data.
- *function*: analysis that must be performed on the data. This function must be chosen by our library, e.g., one can do *average of dot products*, *weighted sums* etc. and compose multiple primitive functions together.

The compiler processes the data and generates all the necessary data structures and circuits, ready to be proved in zero-knowledge. Finally, we can deploy our function-specific on-chain verifier as a smart contract.

Specifically, our library performs the following operations:

1) Compute the hash of data, sign the hash using an authority's private key using Schnorr signature protocol.
2) Perform the chosen function on the data in Python to compare the result with the one executed by the circuit in Zero Knowledge.
3) Generate a new comprehensive configuration file in JSON format to programmatically share and re-create the circuits.
4) Generate the circuit! Given a configuration file, our library will generate a structured folder with all files needed to generate the proof in Zero-Knowledge using Noir. The circuit compiles and generates valid proofs right out of the box without having the user write anything in any domain-specific language. Specifically, our library can generate circuits on any data size and shape and can prove:
   - Proof of Provenance: Compiles circuits that can compute the data hash using SHA256 and checks that the hash is valid and that the data comes from the authority using Schnorr Signature.
   - Proof of Statement: Translates the chosen function into a valid circuit and checks that the function applied to the data results in the expected statement previously computed in Python.
5) Generate smart-contract verifier for the chosen function, ready to be deployed on-chain.

Once the process has completed, the user can navigate to the generated folder and immediately prove and verify the compiled circuit, without any additional modification on the DSL source code.

B. Scalability

Scalability is a critical challenge for any ZKP framework, as the proving and verification times typically increase with the size of the input data and the number of arithmetic gates required by the function.

To address this challenge, we have conducted experiments on one of the functions from our library named `average_dot_products`, which takes as input a two-dimensional matrix and a vector. In this function, the dot-product between each row of the matrix and the vector is computed and then averaged over all rows.

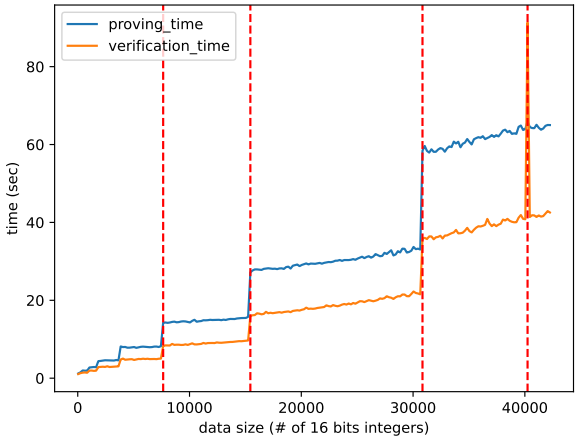

Fig. 6: Scalability of prover and verifier. The x-axis represents the cumulative number of 16-bit integers in all data arrays, which corresponds to the size of the input data. The y-axis represents the proving and verification time in seconds.

We generated data for matrices and vectors of different sizes, where each element is a random integer of 16 bits. Figure 6 shows the proving and verification times for different sizes of data, as the number of operations increases with the data size.

We have observed that the proving and verification times increase with the size of the input data, as expected. Overall, the increase is sub-linear, indicating that our approach can scale well for large datasets. However, in certain regions (indicated by dotted lines), we have noticed a significant increase of the proving and verification times. To address this issue, we plan to conduct more experiments and explore additional optimization techniques to further improve the scalability of our approach in the future.

VII. Implementation on Ethereum

In this section, we provide a detailed description of the technical implementation of Fact Fortress on the Ethereum blockchain. We discuss the various components of the system, including the circuit generator, the proof generator, and the on-chain verifier, and explain how they work together to provide a secure and efficient solution for fact-checking.

Additionally, we highlight the system's current limitations and discuss potential areas for improvement.

A. Off-Chain Layer (Circuits)

Several circuits have been implemented for this project. In that context, we use Noir, a domain-specific language for SNARK proving systems [1] created by the developers of the Aztek Network . Noir compiles circuits into Abstract

Circuit Intermediate Representations (ACIR) compatible with different SNARK-based proving systems. We have used the default one for this project: the PLONK SNARK prover Barretenberg.

1) Circuit 1. Toy Circuit: Validation of Schnorr Signatures: The first circuit we implemented ensures a data provider has signed data. This circuit takes a public key generated on the Grumpkin curve (an elliptic curve based on BN254 [8]) as public input, the hashed private data, and its associated signature as private inputs. It then verifies that the signature corresponds to the public key using the following Rust function .

```
fn verify_signature(
    _public_key_x: Field,
    _public_key_y: Field,
    _signature: [u8; 64],
    _message: [u8]
) -> Field
```

Thanks to this circuit, once proof of provenance is generated, a verifier can ensure that the data provider has signed the hash. This circuit is presently used in our end-to-end solution; we will replace it soon with auto-generated circuits that embeds the proof of provenance (see above).

B. Solidity Smart Contracts

To date, we have implemented three Solidity smart contracts.

The first contract[3] is the entry point of the solution. It notably interacts with two other contracts described below that grant and revoke permissions (to perform on-chain actions and access online data). It also interacts with the validator contracts to verify public inputs and proofs submitted by verifiers.

The second and third contracts[4] implement the ERC-721 Non-Fungible Token standard [9] to manage NFTs. These NFTs are used as *decentralized* authorization tokens. They allow data providers to update their public keys on the blockchain and enable data analyzers to access the data.

We have also improved an existing mechanism from the Noir library to generate smart contracts. Noir offers two ways to generate the verifier smart contract: via command line (`nargo codegen-verifier`) or programmatically, via a TypeScript library (`@noir-lang`). We identified that these approaches generate different smart contracts. After having assessed them, we have determined that generating the verifier smart contract programmatically was more optimal—while imperfect (see Section VII-C)—for our use case, to create the verifier contracts.

The fourth contract, generated in these conditions, verifies the proof of provenance using the Schnorr signature verification circuit. Additional validator contracts—notably verifying the risk score function—are scheduled to be deployed.

[3]https://github.com/pierg/fact-fortress-dapp/blob/main/contracts/factFortress.sol

[4]ERC-721 smart contracts: https://github.com/pierg/fact-fortress-dapp/blob/main/contracts/dataProvidersNFTs.sol, https://github.com/pierg/fact-fortress-dapp/blob/main/contracts/dataAnalystsNFTs.sol

These smart contracts are currently deployed on Ganache, an Ethereum blockchain simulator for development purposes [10].

C. Current Limitations

We have identified and addressed several shortcomings affecting Noir. Issues were expected, as this language is still experimental.

Transient Blocking Issue Affecting the Programmatic Compilation of Circuits

The circuit generator is currently not fully connected to the rest of the application. The issue comes from the fact that our autogenerated contracts, which use recent features offered by Noir (v0.4.0 and above), are not programmatically compilable due to a bug affecting the Noir WASM module, as we explain below. In other words, while the most recent version of the `nargo` command can compile them, our application presently lacks this ability.

Because the circuits are compiling off-chain, we are confident about the technical feasibility of achieving a full integration in the short term.

Overcoming Technical Limitations From Noir

First, `@noir-lang` a WebAssembly (WASM) module, written in Rust, to compile the circuits. Using WASM *modules* allows the Noir developers to maintain the compiler in Rust while offering the possibility of implementing applications (the *hosts*—in WebAssembly terminology) that call this compiler in other languages. The host function provides the path to the circuit definition to the WASM module, which compiles it and returns the compiled circuit to the host. This decoupling between the host and the module has led to difficulties in this project for two main reasons. (a) We could not programmatically compile some contracts, while the CLI command, `nargo`, could; the WASM module was indeed outdated. (b) Compiler errors from the `@noir-lang` WASM modules are hard to analyze, as demonstrated by this example [11], where a runtime error, from the host perspective, is caused by an obscure circuit compilation error, from the module perspective.

Second, we noticed that the default smart contracts autogenerated by Noir to verify Schnorr signatures failed at compile or runtime. More precisely, contracts generated by the command line systematically reverted despite valid inputs. And contracts programmatically generated by (`@noir-lang`) were affected by a `Stack too deep` compiler error, as Noir generates a function referring to a number of variables larger than the EVM stack limit (limited to 16 slots). The issue was fixed by adding block scopes [12], and Fact Fortress now uses this technique to make the verifier contracts compilable.

Third, and more problematically, the verifier smart contract functions generated by Noir only accept the serialized proof as a parameter, which contains public inputs as embedded bytes. In other words, these contracts do not allow to specify the public inputs associated with a proof. To overcome this issue, an extractor and a public input verifier had to be

implemented in the main smart contract[5]. Therefore, the main contract performs a pre-check on the public inputs before calling the relevant verifier smart contract with the actual proof.

The Noir language represents a valuable resource for simplifying the implementation of circuits. Despite its experimental nature and occasional bugs, our experience shows that Noir offers significant advantages for circuit generation, including ease of use and increased efficiency. These benefits can be considerable when time and resource constraints are essential.

VIII. Conclusion

ZKP is mostly associated with blockchain technology, where it enhances transaction privacy and scalability through rollups, addressing the data inherent to the blockchain. Our approach focuses on safeguarding the privacy of data external to the blockchain, with the blockchain serving as publicly auditable infrastructure to verify the validity of ZK proofs and track how data access has been granted without revealing the data itself. This provides a robust mechanism for preserving sensitive information privacy while leveraging blockchain technology's security and transparency.

In addition, our framework Fact Fortress provides an efficient and user-friendly platform for designing and deploying zero-knowledge proofs of general statements. The high-level abstractions we provide enable developers to express complex computations without worrying about the underlying arithmetic circuits. We demonstrated that our approach scales well for large datasets, with a sub-linear increase in proving and verification times with the input data size. In the future, we plan to conduct more experiments and explore optimization techniques to improve the scalability of our approach further.

Our framework is open-source and available on GitHub. We believe that it can help democratize the use of zero-knowledge proofs and enable the development of more privacy-preserving applications.

References

[1] "Noir," https://noir-lang.org.
[2] O. Goldreich, "Secure multi-party computation," *Manuscript. Preliminary version*, vol. 78, no. 110, 1998.
[3] "Circom," https://docs.circom.io.
[4] "Arkworks," https://arkworks.rs.
[5] "Zokrates," https://zokrates.github.io.
[6] "Leo," https://leo-lang.org.
[7] "Cairo," https://cairo-lang.org.
[8] A. Gabizon, Z. Williamson, and T. Walton-Pocock, "Aztec yellow paper." [Online]. Available: https://hackmd.io/@aztec-network/ByzgNxBfd
[9] D. S. William Entriken, "ERC-721: Non-Fungible Token Standard," https://eips.ethereum.org/EIPS/eip-721, 2018.
[10] "Ganache," https://github.com/trufflesuite/ganache.
[11] "@noir-lang library not compiling v0.4.0-compatible circuits · issue #1175 · noir-lang/noir," https://github.com/noir-lang/noir/issues/1175.
[12] "Unable to verify a schnorr signature via a smart contract · issue #1133 · noir-lang/noir," https://github.com/noir-lang/noir/issues/1133#issue-1662133852.

[5]See: https://github.com/pierg/fact-fortress-dapp/blob/main/contracts/factFortress.sol#L233-L287