

PEEV: Parse Encrypt Execute Verify - A Verifiable FHE Framework

Omar Ahmed, Charles Gouert, and Nektarios Georgios Tsoutsos

University of Delaware
{oaaa, cgouert, tsoutsos}@udel.edu

(Updated September 1, 2026)

Abstract. Cloud computing has been a prominent technology that allows users to store their data and outsource intensive computations. However, users of cloud services are also concerned about protecting the confidentiality of their data against attacks that can leak sensitive information. Although traditional cryptography can be used to protect static data or data being transmitted over a network, it does not support processing of encrypted data. Homomorphic encryption can be used to allow processing directly on encrypted data, but a dishonest cloud provider can alter the computations performed, thus violating the integrity of the results. To overcome these issues, we propose PEEV (Parse, Encrypt, Execute, Verify), a framework that allows a developer with no background in cryptography to write programs operating on encrypted data, outsource computations to a remote server, and verify the correctness of the computations. The proposed framework relies on homomorphic encryption techniques as well as zero-knowledge proofs to achieve verifiable privacy-preserving computation. It supports practical deployments with low performance overheads and allows developers to express their encrypted programs in a high-level language, abstracting away the complexities of encryption and verification.

Keywords: Cloud Computing, Homomorphic Encryption, Private and Verifiable Computation, Zero-Knowledge Proofs.

1 Introduction

Cloud computing has been rapidly growing and adopted by many organizations to outsource heavy computations to high-performance servers that are provided through services maintained and operated by third parties. This removes the burden of creating and maintaining costly computing infrastructure for an organization. Also, it provides people and businesses with increased productivity, speed and efficiency, and cost savings [56, 58, 65]. However, end users keep voicing concerns about their sensitive data, as cloud-level threats can put their privacy at risk. In this case, a cloud user cannot fully trust a cloud provider; for example, since the client’s data are stored and processed on the cloud’s servers, a curious service provider could read the user’s data. This can potentially lead the service provider to learn secret information about individuals and organizations (such as financial information and health records). Likewise, a curious provider can use their clients’ data for online advertising [57]. In addition, cloud computing is susceptible to a variety of cyberattacks including network attacks and account hijacking [20, 35, 50].

While numerous research efforts have been proposed to counter cloud attacks [19, 37, 43], deploying these defenses in practice is limited and doesn’t fully prevent a curious provider from reading the client’s data. A potential solution to mitigate these issues is using modern cryptography: end users can encrypt their data using algorithms like AES and upload it to remote cloud servers. However, this method is only suitable for protecting static data, which limits usability and prevents the server from performing any meaningful computation on the outsourced data. But what if end users need to process their data after being uploaded to the cloud and also preserve their privacy? In this case, traditional cryptography cannot help.

To address this challenge of privacy-preserving computation on the cloud, we need to employ advanced cryptography that allows a cloud provider to perform computations directly over encrypted data without revealing the underlying sensitive plaintexts. A promising solution is Fully Homomorphic Encryption (FHE),

which allows performing meaningful computations over encrypted data without decrypting it; specifically, the decryption of a processed FHE ciphertext equals the output of an equivalent computation over plaintext data. For example, suppose that a user has two plaintext values, x and y , and a function F as in Equation 1:

$$z = F(x, y) = x + y \quad (1)$$

Here, we assume that the values x and y are confidential, and the user does not have the computational resources to compute the function F locally. If the user does not trust a cloud provider with her data in plaintext, FHE offers a viable solution. The client can outsource the computation of F by homomorphically encrypting x and y to x' and y' and introducing an equivalent homomorphic function F' , as shown in Equation 2:

$$z' = F'(x', y') = x' + y' \quad (2)$$

and by decrypting z' , we get $x + y$, as shown in Equation 3.

$$\text{Decrypt}(z') = z = x + y \quad (3)$$

Essentially, as shown in Equation 4, FHE ensures that a computation over a homomorphic ciphertext is equivalent to the same computation over the original plaintext.

$$F(\text{plaintext}) = \text{Decrypt}(F'(\text{ciphertext})) \quad (4)$$

Although FHE offers a paradigm-shift in privacy-preserving computation, it has considerable difficulties that hinder developers from creating scalable and reliable trustworthy cloud services. These difficulties include the correct setup of encryption parameters, translating plaintext data into ciphertext data, and converting a program that operates on the plaintext data into a version that supports ciphertext data. While there are several homomorphic encryption implementations available [5, 40, 61], they are not trivial to use without a thorough understanding of the cryptographic primitives. On top of that, writing and maintaining a consistent program flow is challenging, especially considering these libraries offer different APIs and some of the common programming primitives (e.g., loops) are not directly supported. In addition to the low-level homomorphic libraries, state-of-the-art compilers have emerged that translate a program written in a high-level language into its FHE equivalent [12, 28, 29, 30, 53, 69].

Therefore, FHE has become a powerful new tool for running computations over encrypted data. However, one major challenge still remains: *how can the users be assured that the encrypted computation was performed faithfully?* Indeed, a client cannot be sure that all steps of the outsourced function were correctly followed; for example, when a client sends ciphertexts x' and y' to the cloud and request to compute $F'(x', y')$, an untrusted cloud server can cheat and compute another arbitrary function $G'(x', y')$. In this case, the user receives and decrypts the resulting ciphertext, and instead of getting the sum $x + y$, she may get the difference $x - y$. Verifying that the outsourced computation was performed faithfully is a serious concern for applications that involve critical data, such as medical applications informing decisions on patients' health. Equally important, Machine Learning as a Service (MLaaS), which refers to cloud-based services that run pre-trained machine learning models on demand, has become increasingly popular in the business sector [32, 63]. An untrusted MLaaS provider can violate the integrity of a computation, leading to drastically incorrect results.

To address this challenge, the research community has focused on creating techniques to verify an outsourced computation without leaking any sensitive data. One prominent method is zero-knowledge proofs (ZKPs), which are verification protocols that allow one party (called prover) to convince another party (called verifier) that a mathematical statement is valid without revealing any additional information other than the correctness of the statement [27]. ZKPs have gained much attention and improved over the years due to their importance in verifiable computation [38, 47, 52, 68].

In the cloud computing paradigm, the prover is the cloud server, the verifier is the end user, and the statement to be proved is the computation over the encrypted values. In simple terms, the process works as follows: the user uploads both the encrypted data and the function that needs to be executed directly over the encrypted data. The cloud then performs the computation, generates the computation's proof, and sends

the encrypted result along with the proof back to the user. The user then verifies this proof and proceeds to decrypt the result if the proof is validated. Otherwise, the encrypted result is discarded.

The presented framework combines the power of fully homomorphic encryption and zero-knowledge proofs, creating a trustworthy computing framework, dubbed PEEV, that enables both *private and verifiable computation*. In this context, a user can delegate the execution of a program processing FHE ciphertexts to a remote server, while also verifying the integrity of the computation using ZKPs. Notably, a user does not need extensive knowledge of cryptography in order to write a program that will be executed homomorphically by the cloud; instead, the user writes the program in a high-level language, which makes it easier for developers, instead of using the low-level APIs provided by FHE libraries.

A key component in our approach is the translation of a high-level program into an FHE-compatible arithmetic circuit. Such a circuit is a blueprint that defines the encrypted inputs and the computation to be executed. It is no surprise, however, that creating an arithmetic circuit from high-level code is an intricate process, as it involves multiple steps, from analyzing the program flow and eliminating branches, to unrolling loops, and optimizing the code (e.g., removing variables that do not contribute to the final result). PEEV offers a comprehensive framework that automates key parts of the process: reading and executing an arithmetic circuit in FHE, initializing and setting the encryption parameters, generating and verifying the execution proof, and decrypting the result. Overall, our contributions can be summarized as follows:

1. Introducing PEEV, a verifiable privacy-preserving computation framework that combines the power of zero-knowledge proofs and fully homomorphic encryption for secure outsourcing to the cloud.
2. Design of a novel parser (YAP) that automatically translates high-level code into optimized low-level HE programs.
3. A novel intermediate representation, Operations List (OpL), for FHE, featuring an easy to understand syntax, and compatibility with different HE library targets.

The rest of the paper is organized as follows: Section 2 discusses prior works addressing the problem of verifiable computation, while Section 3 provides a background on homomorphic encryption schemes and libraries, zero-knowledge proofs, and compilers for HE and ZKPs. Section 4 introduces the proposed approach for achieving verifiable privacy-preserving computation on encryption data, while Section 5 describes the implementation details of the framework. Section 6 presents the experimental results over representative benchmarks. Finally, a discussion of practical considerations and the concluding remarks are presented in Sections 8 and 9, respectively.

2 Related Work

In 2012, the authors of [2] introduced a cryptographic primitive called delegatable homomorphic encryption (DHE) that allows one party to delegate the computation of a circuit with encrypted data to an untrusted party. This work was somewhat limited, as it could only handle functions that took one encrypted input. Moreover, the DHE architecture has four parties involved in the process: a sender who wants to delegate a computation; a receiver who publishes public keys for the senders to prepare the encrypted inputs; a trusted authority that assigns computational resources to the evaluator; and the evaluator, who is responsible for executing the computations. Conversely, PEEV can execute circuits with an arbitrary number of user inputs, and a trusted third party is only involved for issuing the ZKP keys for the cloud and the client. Another limitation of DHE is that it does not provide confidentiality on the user’s input/output data, i.e., they are provided as clear text. Moreover, the class of functions that can be delegated are limited. However, PEEV provides confidentiality on the user’s input and the server output with various classes of functions.

Another related cryptographic primitive is a homomorphic encrypted authenticator (HEA) [39] that was proposed in 2014. The HEA enables the construction of verifiable homomorphic encryption that allows confirming an outsourced computation on encrypted data. Nevertheless, homomorphic authenticators still remain not practical because they are computationally expensive [1, 41]. On the contrary, our framework can be practically adopted thanks to the fast performance of the adopted primitives. HEA requires that the programs and the data to be authenticated be labeled, thus it provides no usability for the user. Meanwhile,

PEEV requires the user to only write the program in a high-level language, and the framework handles the remaining steps.

In 2014, Fiore *et al.* proposed an efficient construction for verifiable computation (VC) that enabled authenticating computations on encrypted data [22]. Its efficiency came from a homomorphic hashing technique, which could verify the computations on ciphertext data at the same cost as plaintext data. Nevertheless, the initial generic construction introduced efficiency issues when the FHE ciphertext space does not match the message space supported by the VC scheme. Another downside of this construction is that it is not an outsourcing of a computation, but a function query. This means that a user can have an encrypted input x , send it to a remote server, and receive $F(x)$. However, PEEV allows the user to outsource her own circuit with any number of inputs to a remote server. Furthermore, PEEV uses Zero-Knowledge Protocols to verify the correctness of the computations, but the work presented by Fiore *et al.* used homomorphic hash functions, which are slower than ZKPs.

Similarly, vFHE [45] proposed a methodology for introducing a blind hash to preserve integrity encrypted computation. However, the construction is only applied to matrix multiplication with non-invertible matrices, whereas PEEV is generic and can be applied to arbitrary algorithms. Although the vFHE work addresses the integrity issue on encrypted computations, it offers no usability for the user. Specifically, a user will have to handle the details of the hash functions along with the circuit execution. Conversely, our framework offers a higher usability level that allows users to only write a high-level program without the headache of adjusting the computations and the hashing operations.

The authors of [8] proposed schemes that enabled verifying HE computations of constant multiplicative depth. Their main goal was to allow verifiable and private delegation of computation with three properties: privacy, integrity, and efficiency. In addition, they introduced a protocol based on homomorphic hash functions that allows choosing homomorphic encryption parameters flexibly. Although this model is efficient, it needs a random oracle to become a non-interactive protocol. Meanwhile, the choice of Rinocchio in PEEV offers support for non-interactive proofs. Another difference is that PEEV allows private verifiability, instead of assuming public verifiability. Thus, in PEEV, only the user of the cloud service is able to verify the correctness of the computations. However, in a public verifiability setting, anyone can verify the correctness of the computations. Private verifiability is a more convenient setting in PEEV, since the cloud user solely outsources her own computation to a remote server instead of the cloud which offers a specific computation to the public.

In 2018, Luo *et al.* proposed a methodology for ensuring the decryption correctness for BGV ciphertexts [46]. The authors proposed an interactive ZK protocol to generate proofs. However, one limitation of interactive ZKPs is the additional communication overhead introduced, since it requires an interaction between the prover and the verifier. Conversely, PEEV leverages non-interactive ZKPs, which overcomes this issue; hence, it is more efficient in terms of the amount of data exchanged over the network. Another significant drawback of the work proposed by Luo *et al.* is that a prover can still deceive the verifier; that is, there is no guarantee that the output produced by the prover is the correct result expected by the verifier. However, in PEEV, it is impossible for a prover to deceive the verifier. Even if the prover alters the circuit or changes the proving key, the verification process will detect an invalid proof.

Recent works for providing integrity with homomorphic encryption include verifying FHE computations by utilizing trusted execution environments (TEEs) [18, 54, 66], as well as verifying the integrity of a computation based on MACs [13, 42]. Nevertheless, these approaches rely on different approaches than PEEV, which enables both integrity and confidentiality using FHE and ZKPs.

The concept of privacy-preserving also appears in different applications and contexts. For example, the work proposed in [44] used Elliptic Curve Cryptography and Paillier cryptosystem to introduce a reputation updating scheme vehicular networks. In addition, the authors of [51] proposed a scheme that allows to securely query outsourced encrypted data on location-based services. This scheme efficiently uses the Geohash algorithm to reduce the computations overhead and to speed up the query on large datasets.

Another important aspect that contributes to PEEV's benefits over existing frameworks is its support for system usability. Unlike the aforementioned frameworks, PEEV enables users to express computations in a high-level language, enhancing usability. Consequently, our work bridges the gap between theory and

practice. Although previous works are well-defined and offer concrete solutions to the problem of verifiable computations on encrypted data, their complexity may render them unattractive to end-users. Table 2 shows a comparison between our proposed framework and other existing frameworks.

3 Background

3.1 Homomorphic Encryption Schemes

Homomorphic encryption is akin to traditional cryptography, but with the additional ability to perform computations directly on ciphertexts. Various homomorphic encryption schemes have different computational capabilities. In particular, HE schemes are categorized into three classes: partially homomorphic schemes, leveled homomorphic schemes, and fully homomorphic schemes.

- **Partial HE (PHE)**. These schemes support unlimited evaluations of one type of operation, such as addition or multiplication. Although they are easy to integrate into existing codebases and are generally computationally efficient, their applications are limited, such as for access control [60]. Examples of PHE include the unpadded RSA, ElGamal, and Paillier cryptosystems [49].
- **Leveled HE (LHE)**. More powerful than PHE, LHE supports evaluating circuits with both addition and multiplication but with limited depth. The security of LHE schemes depends on the learning with errors (LWE) [59] and ring learning with errors (RLWE) [48] problems. As a result, performing computations on encrypted data leads to noise growth. If the noise reaches a certain limit, it can result in incorrect decryption of the output for deep circuits, especially those implementing algorithms involving a large set of multiplications. In particular, the noise grows slightly with each addition operation, whereas it grows substantially with each multiplication operation. Likewise, as circuits get deeper, evaluating them becomes more expensive because they require larger parameters to accommodate the noise requirements. This results in more costly additions and multiplications. Examples of LHE schemes are the Brakerski-Gentry-Vaikuntanathan (BGV) [10], the Brakerski/Fan-Vercauteren (BFV) [21], and the Cheon-Kim-Kim-Song (CKKS) [15] cryptosystems.
- **Fully HE (FHE)**. This variant supports evaluating arbitrary circuits by allowing unbounded addition and multiplication and is an extension of LHE with *bootstrapping*. The latter is the mechanism that stands behind the robustness of FHE; it reduces the noise level within a ciphertext, hence allowing more computations to be carried out on the data [25]. Nevertheless, bootstrapping is a very costly technique, being over an order of magnitude slower than other HE operations. Even with proposed optimizations [14, 26, 36], it still adds noticeable computational overhead relative to LHE schemes. Therefore, in the case of circuits with limited depth, an LHE scheme is a better option compared to an FHE scheme. As such, this work focuses on LHE.

3.2 Homomorphic Encryption Libraries

Given the powerful capabilities of homomorphic encryption and the increasing demand of privacy-preserving computing, many open-source HE libraries have been proposed. These libraries implement different schemes, and each one exhibits its own API for executing operations on encrypted data. A few prominent examples are discussed below:

- **Blyss SDK**. Blyss is a private information retrieval library, built on top of an HE backend. It is written mainly in Rust and JavaScript.
- **TFHE**. Written in C++, TFHE provides fast bootstrapping based on the CGGI cryptosystem [16, 17]. It operates on Boolean circuits, where plaintext data are encoded into binary and the ciphertext is generated by encrypting the plaintext bit-by-bit. Another implementation of CGGI is TFHE-rs, which is written in Rust and supports encodings for both integer and Boolean arithmetic [70].
- **FINAL**. A cryptographic implementation written in C++ that provides FHE based on the LWE problem and NTRU cryptosystem. FINAL exhibits optimized bootstrapping which makes it more efficient than the TFHE library [9].

- **HElib**. HElib implements the BGV and CKKS schemes [33]. The developers of the library introduced optimizations for evaluating homomorphic operations. However, the bootstrapping and execution times remain high, which makes it unsuitable for executing arbitrary circuits.
- **Lattigo**. An HE library based on RLWE and written in the Go language, it implements the BFV, CKKS, and BGV schemes. Additionally, Lattigo supports multi-party homomorphic encryption [40].
- **SEAL**. Microsoft released its own HE library called SEAL [61]. It supports the BGV and BFV schemes for performing additions and multiplications on encrypted integers, and the CKKS scheme for performing additions and multiplications on encrypted real numbers. SEAL provides a simple API for writing leveled homomorphic encryption. Although it is not suitable for deep circuits involving a large number of computations, it is optimized for applications that include a finite number of arithmetic operations for several reasons: its simplicity compared to other libraries, the fact that it is written in C++ with no required dependencies (rendering it easy to compile and deploy in different environments), and its fast performance for arithmetic operations. For these reasons, we chose to use SEAL as the HE back-end for this work.

3.3 Zero Knowledge Proofs

ZKPs represent a major innovation in applied cryptography and are used extensively in blockchains and cryptocurrency [62]. They were first introduced in 1985 [27] and enabled conveying a claim without revealing any additional information about that claim other than its correctness or incorrectness. A zero-knowledge protocol has three properties, described below:

- **Completeness**. If the claim is true, and the prover and verifier are honest, the verifier will accept the proof.
- **Soundness**. A dishonest prover cannot trick the verifier into accepting an invalid claim.
- **Zero-knowledge**. The proof leaks nothing about the claim, thus, a verifier learns nothing about the claim beyond its validity or invalidity.

Besides the aforementioned properties, a ZKP has three basic elements:

- **Witness**. This is the secret data that a prover assumes knowledge of.
- **Challenge**. This is a sequence of queries generated by the verifier to confirm the prover’s claim.
- **Response**. This is a sequence of answers generated by the prover as a response to the challenge issued by the verifier.

From these three elements (Witness, Challenge, and Response), it is clear that the prove-verify process is similar to a sequence of questions and answers. In fact, this structure describes the interactive ZKP. In this scenario, the prover and the verifier establish a back-and-forth communication channel with queries from the verifier and answers from the prover. As a result, this interactive nature limits the usage of the ZKPs as they are time-consuming and introduce a significant communication overhead, which makes ZKPs impractical for some applications.

Conversely, a non-interactive ZKP (NIZK) was first proposed by Blum *et al.* in 1988 [6]. In this scheme, the prover has a secret key for generating a proof, and the verifier has another key for verifying the proof. In this way, there is no need for an interactive session between the prover and the verifier, making ZKPs more practical.

There are three common types of NIZK systems: zk-SNARK, zk-STARK, and Bulletproofs. These are discussed below:

- **zk-SNARK**. This stands for Zero-Knowledge Succinct Non-Interactive Argument of Knowledge and was first introduced in [4]. It requires a *trusted setup* to publish a proving key and a verification key. These two keys are public parameters, which are generated only once for each circuit. A zk-SNARK system has the following properties:
 - **Zero-knowledge**. The proof leaks nothing about the witness, so a verifier learns nothing beyond its validity or invalidity.

- **Succinctness.** The proof is small and can be verified quickly and easily.
 - **Non-interactive.** The system does not require multiple rounds of interaction between the prover and the verifier.
 - **Argument of Knowledge.** The prover generates a proof that is sound, and it is impossible for a prover to generate a valid proof for an invalid witness.
- **zk-STARK.** This stands for Zero-Knowledge Scalable Transparent Argument of Knowledge, introduced in Ben-Sasson *et al.* [3]. A zk-STARK is similar to a zk-SNARK but overcomes the problem of trusted setup. Besides the zero-knowledge and the argument of knowledge properties, zk-STARK has the following two properties:
- **Scalable.** This property makes STARKs more favorable over SNARKs, as it is faster at proving large proofs than SNARKs. Given a large witness, proof generation and verification grow slightly with STARK; unlike SNARK, they grow linearly.
 - **Transparent.** This property means STARK does not need a trusted setup; it generates its parameters based on publicly available randomness.
- **Bulletproofs.** This protocol generates short proofs (logarithmic in the witness size) without the need for a trusted setup environment. However, Bulletproofs’ verification process is more time-consuming than SNARK verification. Bulletproofs are efficient for cryptocurrencies; thus, it is very suitable for systems that require secure transactions and distributed and trust-less blockchains [11].

3.4 FHE and ZKP Compilers

Researchers in the cryptography community have been working extensively to create compilers and frameworks that facilitate the creation of FHE systems and other related applications. These compilers aim to translate a high-level language program written over plaintext data into an equivalent encrypted version. This encrypted version could be an implementation using the primitives provided by HE libraries. Likewise, in case of ZKP systems, compilers create an R1CS (Rank-1 constraint satisfiability) system. The R1CS captures a computation and transforms it into a set of matrices and vectors that can be used by proof systems. Such tools have several advantages: by simplifying the code-writing process, a developer does not need an in-depth knowledge of cryptographic primitives, code optimization, and managing key setup, encryption, and decryption. A selection of state-of-the-art compilers are discussed below.

- **Circom** is a compiler with its own language used for defining arithmetic circuits for ZKP applications. It is written in Rust and provides developers with an easy-to-use interface for generating R1CS files. The authors of Circom implemented three zk-SNARK systems: snarkjs, wasmsnark, and rapidSnark.
- **CirC** is a compiler infrastructure, written in Rust, that supports translating a high-level language into circuits [55]. CirC can compile code written in C, ZoKrates, or Circom into circuits for Satisfiability Modulo Theories (SMT), Multi-Party Computation (MPC), or R1CS. One of the powerful features of CirC is that it compiles different high-level languages into an optimized Intermediate Representation (IR). Then, the IR is translated to the target circuit. Although CirC supports different front-end languages, it works best with a modified version of the ZoKrates language, called Z#, supporting different constructs such as loops and arrays.
- **T2** is a cross-compiler and a benchmark suite [30]. The main goal of T2 is to explore and compare different HE libraries, including HELib, Lattigo, PALISADE, SEAL, and TFHE, using an extension of the TERMinator Suite [53]. The authors use their own domain-specific language to write unified code that can be compiled to several different HE libraries, ensuring a fair comparison between benchmarks.
- **HELM** is a privacy-preserving framework for processing data in the encrypted domain with FHE [29]. HELM compiles arbitrary programs written in Verilog into homomorphic circuits. The authors accelerated the execution of circuits by introducing a scheduler that allows the processing of encrypted data in parallel and employing rigorous logic optimization techniques.
- **Concrete** is a CGGI compiler that compiles programs written in Python into their FHE equivalent [69]. Concrete supports a large set of Python operators, in addition to its compatibility with the NumPy library. Despite the extensive work devoted to developing this library, it is not mature yet; it has several limitations, such as not supporting control flow statements (e.g., *if* or *while* loops), not supporting floating point inputs or outputs, and a small bit width of encrypted values.

4 Verifiable Privacy-preserving Computation

The goal of this work is to add an integrity component to privacy-preserving computation, thus enabling verifiable privacy-preserving computation (VPPC), by introducing the PEEV framework. In this regard, a client who is outsourcing a computation to a potentially dishonest server can verify the validity of the computation without revealing any sensitive information.

Figure 1 depicts our proposed approach. To outsource a computation, the client must define the arithmetic circuit to be executed on the server and encrypt the circuit's inputs. The client then sends the arithmetic circuit along with the encrypted input, R1CS and the evaluation key to the server. The server executes the circuit using the evaluation key, generates the proof using the proving key, and sends the proof along with the encrypted result to the client. The client verifies the proof using the verification key, and if it is valid, accepts the computation and decrypts the result. If the proof is invalid, the client discards the result.

Our VPPC model is characterized by a four elements tuple (**Setup**, **ProbForm**, **Eval**, **Conc**):

- **Setup**(P) $\rightarrow$ ($E.Params, V.Params, OpL$) - This is a procedure that takes our program P , written in a high-level language, and outputs the encryption parameters, $E.Params$, the verification parameters $V.Params$, and the compiled program, Operations List (OpL).
- **ProbForm**($OpL, V.Params$) $\rightarrow$ ($C, V.Prov, V.Ver, E.Eval, E.Dec$) - This procedure is fed with the compiled program, OpL , and the verification parameters, $V.Params$. It outputs the encrypted arithmetic circuit C to be executed, the proving key $V.Prov$ for proof generation, the verification key $V.Ver$ for proof verification, and the evaluation key $E.Eval$ for evaluating the circuit.
- **Eval**($C, OpL.Inputs, V.Prov, E.Eval$) $\rightarrow$ (γ, ρ) - The server executes the arithmetic circuit C on the client's specific encrypted inputs, $OpL.Inputs$, using the evaluation key, $E.Eval$. It outputs the resulting ciphertext, γ , and the zero-knowledge proof, ρ .
- **Conc**($V.Ver, E.Dec, OpL.Inputs, \gamma, \rho$) $\rightarrow Res$ - The client enforces atomic verification and decryption gating. The client evaluates the predicate $b \leftarrow \text{Verify}(V.Ver, \rho, OpL.Inputs, \gamma)$, which mathematically authenticates both the public inputs ($OpL.Inputs$) and the public output (γ) against the proof. If $b = 1$, the procedure returns the result, $Res \leftarrow \text{Dec}(E.Dec, \gamma)$. If $b = 0$, it aborts ($\perp$) and purges γ from memory to prevent reaction side-channels.

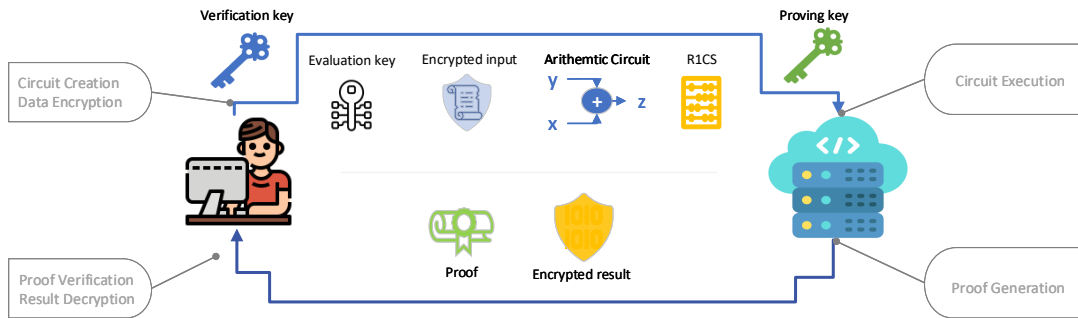


Fig. 1. Client-Server Communication: The client creates the arithmetic circuit, R1CS, and evaluation key, encrypts the input, and sends them to the server. The server runs the computation, generates the proof, and sends the result and the proof back to the client. The client checks if the proof is valid and decrypts the result.

4.1 Client-Side Operations

For a client to delegate a computation to a remote server, two steps have to be completed first: circuit creation and encrypting the circuit’s input. Creating an arithmetic circuit is the process of writing the program that will be executed homomorphically on a remote server. A major challenge is that writing these programs in HE libraries is not trivial, as it requires the user to define each primitive operation explicitly and track each operation’s input and output. The user can end up hard-coding the circuit, leading to potentially thousands of lines of code for even relatively simple algorithms. Moreover, introducing optimizations or updates to the code involves modifying all subsequent lines and other parts of the program since most operations are dependent on each other. Furthermore, incorporating the creation of the R1CS into the program will lead to larger, unoptimized code, so that writing HE programs by hand becomes an infeasible process.

To overcome this issue, PEEV lets the user write her HE program in a high-level language, which is quite easy for developers to optimize, maintain, and define their computations. In this way, our methodology offers several benefits, such as:

- The user will write more readable code that includes common programming structures such as while loops.
- The user can write programs in fewer lines of code compared to a low-level encrypted implementation.
- Optimizing the code or making future modifications can be done rapidly and in a straightforward manner.
- Eliminating the complexity of tracking each operation’s input and output and the creation of the R1CS needed for verifiability.
- The user does not need to manually initialize any HE or ZKP parameters.

Towards that end, we introduce our domain-specific parser, called YAP, that translates a user’s program into an intermediate representation called Operations List (OpL). Specifically, the OpL represents the user’s program in a form that HE libraries can easily parse. The syntax of the OpL contains no complex structures (i.e., functions, classes, and loops), but rather a sequence of operations. The OpL consumes the user’s input and lists all the required operations to compute the final output. Figure 2 illustrates an example of how YAP flattens a simple equation that can be written in almost any high-level language into an OpL.

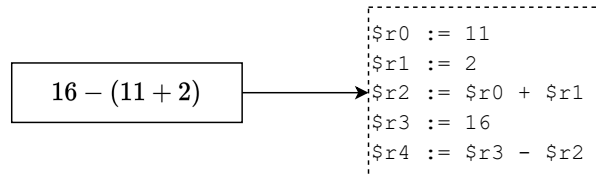


Fig. 2. YAP flattens the equation $16 - (11 + 2)$ into a sequence of operations. The OpL lists every single operation in a single line, and subsequent lines are dependent on previous lines.

The only task an end-user has to perform is to write the desired high-level program, and PEEV handles all necessary steps automatically, including the generation of the OpL, setting of encryption and evaluation parameters, encrypting the input, creating the arithmetic circuit, as well as verifying the proof and decrypting the result after receiving it from the cloud. Initializing the required zk-SNARK keys requires a trusted setup, where the user sends their R1CS to a trusted third party that generates the proving key for the cloud and the verification key for the user.

4.2 Server-Side Operations

After receiving the arithmetic circuit, R1CS, and the encrypted inputs from the user, the server allocates the hardware resources required for the circuit execution. Also, for executing the circuit, the cloud must have the target HE and ZKP libraries. For instance, PEEV employs SEAL and Rinocchio zk-SNARK. After the

homomorphic evaluation of the circuit finishes, the cloud uses the proving key to generate a proof, given the circuit and its R1CS.

Notably, an untrusted cloud will not be able to violate the integrity of the computation. Suppose a cloud provider modified the arithmetic circuit and the R1CS provided by the user (e.g., instead of doing addition between two values, doing multiplication); in this case, the cloud will generate an invalid proof, which will be detected on the client side. This is because the proving and validation keys are functions of the R1CS created by the client. When the user inputs her copy of the R1CS to the verification function, the proof will fail to verify the computations defined by the R1CS.

As shown in Figure 3, the client passes the R1CS definition to the ZKP key generation algorithm, which outputs a verification key for the user to verify the proof and a proving key for the cloud to generate the proof. If the R1CS is changed after key generation, the proving key will not generate a valid proof. Since PEEV uses a zk-SNARK proof system, the key generation algorithm requires a trusted setup environment, which could be acquired through a trusted third party.

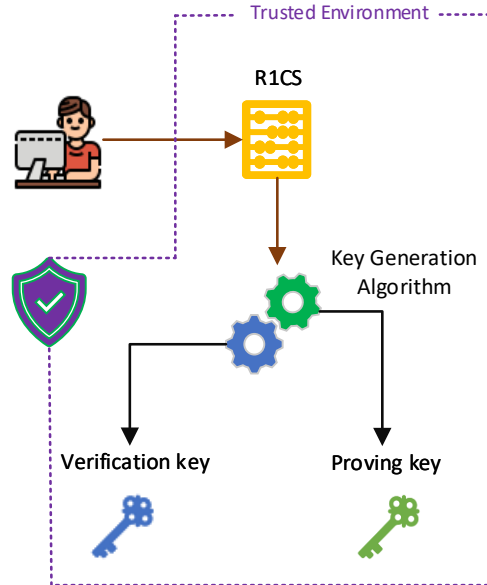


Fig. 3. The cloud user inputs the R1CS to the ZKP key generation algorithm. The key generation process requires a trusted environment, which can be achieved through a third-party who offers a trusted setup. The algorithm outputs two keys: one for the cloud to generate the proof, and the other for the client to verify the proof. Any changes made to the R1CS should reflect new keys.

We note that the user does not share any keys that would allow the cloud to violate the confidentiality of the computation; the user only sends the evaluation key to the server to enable the server to execute the circuit. Similarly, the proving key, which is generated by a trusted third party, doesn't allow the server to violate the integrity of the computation since it is based on the user-provided R1CS. Unlike the ZKP keys, SEAL does not need a trusted third party, thus the encryption keys are generated by SEAL.

4.3 Threat Model

PEEV considers two main entities: the client and the cloud provider. The client has confidential data X along with a function f , and wants to execute f over X such that $f : X \rightarrow Y$. Meanwhile, the cloud provider is assumed to have the computational resources required for executing the function f on the input data, and is the sole computational party for all FHE evaluation operations (excluding encryption and decryption). In the context of zero knowledge proofs, the cloud acts as the prover while the client is the verifier. The threat model assumes a **dishonest** and **curious** cloud provider; meaning that the server can alter the computations (integrity violation) needed by the client and is incentivized to learn information about the user data (confidentiality violation). Integrity guarantees are derived from the Rinocchio SNARK system; the client rejects the result received from the cloud if she fails to verify the proof provided by the cloud. At any point, if the cloud alters the computations defined by the user's function, an invalid proof will be generated and the client will not accept it. Likewise, data confidentiality is assured through the adoption of the BGV FHE scheme, which is based on the Ring Learning With Errors (RLWE) problem. A cloud provider will not be able to learn anything about the data provided by the client except its size.

4.4 Security Considerations

This subsection provides a discussion on the security and reliability of PEEV. Specifically, the security argument of PEEV is attributed to a hard class of mathematical problems, namely ideal lattices. Furthermore, PEEV inherits its reliability in verifying the correctness of computations from the secure encoding and representation of these computations in a structure that is resilient against forgery. Overall, both the BGV encryption scheme and the Rinocchio ZKP encapsulate the security and reliability assumptions of PEEV. Hence, by inheriting the security assumptions of the BGV scheme and Rinocchio, we can argue that PEEV is a secure framework. To simplify this analysis, we have abstracted some details from the original papers of the Rinocchio ZKP [24] and the BGV scheme [10]; for more comprehensive information about both schemes, we refer the reader to the original papers.

BGV Homomorphic Encryption Scheme. Encryption schemes inherit their security from hard mathematical problems that are computationally infeasible to solve. The BGV scheme reduces its security on the Learning With Errors (LWE) and Ring Learning With Errors (RLWE) problems. In particular, LWE is the problem of differentiating between uniform linear equations and random equations that have been garbled with small amount of noise. RLWE is a variant of the LWE, but it is dedicated for polynomial rings over finite fields. The authors of the BGV scheme describe both the LWE and RLWE in a generalized form as the General Learning with Errors (GLWE) problem, which is defined as follows:

Theorem 1. GLWE:

- Let n be an integer dimension, let d be a value of power of 2, let q be a prime integer such that $q \geq 2$, let $R = \mathbb{Z}[x]/x^d + 1$ be a polynomial ring with degree d and $R_q = R/qR$ is a polynomial ring with coefficients in $(-q/2, q/2]$, and $\chi = \chi(\lambda)$ be a noise distribution. All of these parameters are functions of a security parameter λ .
- The GLWE requires distinguishing between these two distributions:
 1. $(\mathbf{a}_i, b_i) \leftarrow R_q^{n+1}$, i.e., sampling a tuple consisting of the vector $\mathbf{a}_i$ and the element b_i uniformly from the ring $R \bmod q$ of size $n + 1$.
 2. $\mathbf{s} \leftarrow R_q^n$, $(\mathbf{a}_i, b_i) \leftarrow R_q^{n+1}$ by uniformly sampling $\mathbf{a}_i \leftarrow R_q^n$, $e_i \leftarrow \chi$, and setting $b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i$.

The GLWE problem assumption is that solving this problem is unfeasible. Typically, if we set $d = 1$, we get the LWE problem, but if we set $n = 1$ we get the RLWE variant.

The BGV scheme is secure under the IND-CPA assumption (i.e., indistinguishability under chosen ciphertext attack), which is defined as a challenge between an oracle (a party that can perform encryption) and an attacker who has a set of plaintext and can query the oracle. The challenge works as follows:

- The attacker sends two plaintexts M_0, M_1 to the oracle to encrypt them.
- The oracle encrypts both M_0 and M_1 into C_0 and C_1 . Then, it randomly selects one of the ciphertexts and sends it to the attacker.
- The attacker wins the challenge if they can determine what plaintext the received ciphertext encrypts with a probability better than 50%.

Based on the above definition of the basis of the BGV scheme, the fundamental encryption is realized by the 4-element tuple (Setup, KeyGen, Enc, Dec):

- **Setup**(1^λ) - A procedure that takes the security parameter λ and generates the parameters set, $\Delta := (R, d, n, q, \chi, N)$ such that R is the ring, d is the degree, n is the dimension, q is the modulus, χ is the noise distribution, and $N = n \times \text{polylog}(q)$.
- **KeyGen**(Δ) - This procedure takes the parameters set Δ and runs two sub-procedures: one for the secret key generation and another for public key generation. The secret key is represented as $\mathbf{s} = (1, \mathbf{t}) \in R_q^{n+1}$, where $\mathbf{t}$ is a vector sampled from χ^n . Meanwhile, the public key is generated by feeding $\mathbf{s}$ and Δ to its sub-procedure to output the public key $\mathbf{A} = (\mathbf{b} || -\mathbf{B}) \in R_q^{N \times (n+1)}$, $\mathbf{B}$ is a matrix sampled uniformly from $R_q^{N \times n}$, $\mathbf{b} = \mathbf{B}\mathbf{t} + 2\mathbf{e}$, and $\mathbf{e}$ is a noise vector sampled from χ^N .
- **Enc**($\Delta, \mathbf{A}, \mathbf{m}$) - This is the encryption procedure that takes the parameters set, Δ , the public key, $\mathbf{A}$, and a message vector $\mathbf{m}$. The ciphertext vector is computed by setting $\mathbf{c} = \mathbf{m} + \mathbf{r}^T \times \mathbf{A} \in R_q^{n+1}$, where $\mathbf{r}$ is a column vector sampled from R_2^N .
- **Dec**($\Delta, \mathbf{s}, \mathbf{c}$) - This is the decryption procedure that takes the parameters set Δ , the secret key $\mathbf{s}$, and the ciphertext vector $\mathbf{c}$ to recover the plaintext $\mathbf{m}$ by computing the dot product between the ciphertext vector and the secret key vector.

Rinocchio ZKP. The structure of Rinocchio relies on securely encoding the inputs of the circuit in a way that allows correctly proving and verifying a computation. Overall, Rinocchio uses two building blocks: Quadratic Ring Programs, which is a system of equations to represent the computations, and a secure encoding scheme over rings.

A verifiable computation model is defined by a tuple of three procedures (Setup, Comp, Verify):

- **Setup**($1^\lambda, F$) - This procedure takes a security parameter λ and a function to execute F , and outputs: a proving key $\mathbb{P}$, a verification key $\mathbb{V}$, and an encoding of the function's input $\mathbb{F}$. In Rinocchio, this procedure is defined by two functions: the first is used to generate a public key and a secret key, and the second for generating an encoding of the function and a verification key.
- **Comp**($\mathbb{P}, \mathbb{F}$) - This procedure takes the proving key $\mathbb{P}$ and allows the remote server to compute the encoded function F . It outputs an encoding of the output $\mathbb{Y}$.
- **Verify**($\mathbb{V}, \mathbb{Y}$) - This is a verification procedure that takes the secret verification key $\mathbb{V}$ and the server's output $\mathbb{Y}$ and returns either 1 indicating the acceptance of the result (a correct computation) or 0 indicating the rejection of the result (an incorrect computation).

The security of this model is represented by a challenge, denoted as H , between an adversary, A (the cloud provider), and a verifier, V (the cloud user). The challenge asserts that given a proving key, $\mathbb{P}$, and a verification key, $\mathbb{V}$, which are generated based on the security parameter λ and the function F , the adversary A cannot alter the computation of F , generate a proof using $\mathbb{P}$, and produce an output $\mathbb{Y}$ such that the verifier accepts the result $\mathbb{Y}$ by verifying it with $\mathbb{V}$. This challenge is expressed as $\Pr(H_A^V[VC, F, \lambda] = 1) = \sigma$, where VC is the computation performed by the server, and σ is a negligible value.

Rinocchio has a structure called Quadratic Ring Programs (QRP) to characterize the satisfiability of arithmetic circuits. In a nutshell, QRP is a building block that allows for the representation of an arithmetic circuit and the definition of the inputs and outputs of the gates. A solution to the QRP is expressed in Equation 5, which states that given a gate with the left input polynomial $V(x)$, the right input polynomial $W(x)$, and the output polynomial $Y(x)$, one must find the target polynomial $t(x)$ that divides $(V(x) \cdot W(x) - Y(x))$.

$$t(x) | (V(x) \cdot W(x) - Y(x)) \quad (5)$$

Constructing a secure SNARK using a QRP requires a secure encoding scheme to represent the polynomials. The encoding scheme works by generating a public key and a secret key based on a security parameter. The secret key allows mapping the polynomials to an encoded form, while the public key is used for evaluating the encoded polynomials of the circuit.

PEEV Security. We formally prove the security of Π_{PEEV} under two core property-based cryptographic definitions: *Computational Verifiability* against a malicious cloud server $\mathcal{S}^*$ (Theorem 2) and *Input Confidentiality* against a curious cloud server $\mathcal{S}^*$ (Theorem 3).

Theorem 2 (Soundness and Verifiability). *Let C_{FHE} be the R1CS-encoded homomorphic arithmetic circuit corresponding to plaintext program C . Assume E.Params bounds the evaluation noise below the BGV threshold $\lfloor q/2t \rfloor$, the Rinocchio SNARK satisfies Knowledge Soundness over polynomial rings R_q , and $E.\text{Eval}$ is hardcoded into C_{FHE} . Then, for any PPT adversary acting as a malicious server $\mathcal{S}^*$, the probability that $\mathcal{S}^*$ produces a verifying proof ρ^* for an incorrect output ciphertext γ^* is negligible in the security parameter λ :*

$$\Pr[\text{Verify}(\text{V.Ver}, \rho^*, X', \gamma^*) = 1 \wedge \text{Dec}(\text{E.Dec}, \gamma^*) \neq C(x)] \leq \text{negl}(\lambda) \quad (6)$$

Proof. We prove Theorem 2 via a reduction to the Knowledge Soundness of Rinocchio:

1. **Game Setup:** The challenger runs $\text{Setup}(1^\lambda, P)$ to obtain parameters and keys $(C_{\text{FHE}}, \text{V.Prov}, \text{V.Ver}, \text{E.Eval}, \text{E.Dec})$, where E.Eval is embedded into C_{FHE} as constant linear combinations. The challenger encrypts $X' \leftarrow \text{Enc}(\text{E.Enc}, x)$ and sends $(C_{\text{FHE}}, X', \text{V.Prov})$ to $\mathcal{S}^*$.

2. **Adversarial Response:** $\mathcal{S}^*$ returns a candidate output ciphertext γ^* and proof ρ^* .

3. **Witness Extraction:** Suppose $\text{Verify}(\text{V.Ver}, \rho^*, X', \gamma^*) = 1$. By the Knowledge Soundness property of Rinocchio, given access to the random coins and auxiliary input of $\mathcal{S}^*$, there exists a PPT extractor $\mathcal{E}_{\text{Rino}}$ that extracts a valid execution witness $\mathbf{w}^* = \{w_i\}_i$ satisfying the R1CS constraints over the public statement vector $\mathbf{u} = (X', \gamma^*)$.

4. **Algebraic Binding:** The R1CS generated by YAP is perfectly constrained, guaranteeing a 1-to-1 deterministic mapping between the witness $\mathbf{w}^*$ and the evaluation. The R1CS enforced by V.Ver includes the explicit binding constraint $w_{\text{internal}} \cdot 1 = \gamma^*$, locking the public output variable γ^* to the internal wire evaluation of C_{FHE} on public input X' .

5. **Correctness Reduction:** Because $E.\text{Eval}$ is hardcoded into C_{FHE} and the noise growth of $C_{\text{FHE}}(X')$ is bounded by E.Params , evaluating C_{FHE} over X' deterministically yields an internal wire w_{internal} such that $\text{Dec}(\text{E.Dec}, w_{\text{internal}}) = C(x)$.

6. **Conclusion:** Since $w_{\text{internal}} = \gamma^*$, any adversary $\mathcal{S}^*$ that causes $\text{Dec}(\text{E.Dec}, \gamma^*) \neq C(x)$ while producing a verifying proof directly breaks the Knowledge Soundness of Rinocchio, which occurs with probability at most $\text{negl}(\lambda)$. $\square$

Theorem 3 (Input Confidentiality). *Assume the BGV homomorphic encryption scheme is IND-CPA secure over the cyclotomic polynomial ring R_q . Then Π_{PEEV} computationally preserves input confidentiality against any PPT cloud server $\mathcal{S}^*$.*

Proof. The proof follows from a standard hybrid argument across games G_0 and G_1 :

- G_0 : Real execution where $\mathcal{S}^*$ receives $X' \leftarrow \text{Enc}(\text{E.Enc}, x)$ alongside public parameters $(C_{\text{FHE}}, \text{V.Prov}, \text{E.Eval})$, where V.Prov is generated independently of x .
- G_1 : Simulated execution where $\mathcal{S}^*$ receives $X'_{\text{dummy}} \leftarrow \text{Enc}(\text{E.Enc}, 0^{|x|})$ alongside public parameters $(C_{\text{FHE}}, \text{V.Prov}, \text{E.Eval})$.

If $\mathcal{S}^*$ distinguishes G_0 from G_1 with non-negligible advantage ϵ , we construct a PPT adversary $\mathcal{B}$ that breaks the IND-CPA security of BGV. $\mathcal{B}$ submits $(x, 0^{|x|})$ to the BGV encryption oracle, receives challenge ciphertext X^* , and passes X^* to $\mathcal{S}^*$. $\mathcal{B}$ outputs whatever decision bit $\mathcal{S}^*$ outputs. Thus, $\text{Adv}_{\mathcal{S}^*}^{\text{PEEV}}(\lambda) = \text{Adv}_{\mathcal{B}}^{\text{BGV, IND-CPA}}(\lambda) \leq \text{negl}(\lambda)$. $\square$

5 Implementation Details

5.1 Verifiable FHE

For enabling private computation, we implement PEEV using SEAL’s implementation of BGV as a back-end. The SEAL library has several advantages that make it a suitable choice for this work. Some of these advantages are:

- It provides a simple, mature API compared to other libraries, which makes integrating it with other frameworks more feasible.
- SEAL is implemented in C++, which can be faster compared to counterparts in languages like Python. Besides, C++ is a versatile language used to develop a wide variety of applications, including database systems, embedded systems, and banking applications.
- It supports different operating systems and environments, including Linux, Android, MacOS, and iOS.
- SEAL enables batching for encoding multiple messages into a single ciphertext, which can increase the HE throughput by several orders of magnitude for certain types of applications.
- Performing arithmetic operations in SEAL is faster than performing the same operations in other libraries such as TFHE, which operates on bits.

For enabling verifiable computation, Rinocchio is used as the back-end ZKP system [24]. Rinocchio is a SNARK that allows verifying ring-based computations. It offers improved performance compared to other systems, and is more FHE-friendly because it supports lattices, which are also the mathematical foundations of FHE schemes [64].

5.2 Translating High-level Languages into Circuits

One of our goals is to allow developers with limited or no in-depth knowledge of cryptography to write programs that can be executed securely on remote servers and verify the computations. In order to do so, we introduce YAP, which is a novel parser that takes a program written in a high-level language and transforms it into OpL. The OpL is then used to create the arithmetic circuit and its R1CS.

Compiling high-level languages is not a trivial process, as it includes comprehending the program flow and transforming complex structures (such as functions and loops) into a simple sequence of primitive operations. Besides that, the compiler should not simply translate every line of code into its corresponding operation; it should optimize the output by removing unusable code blocks and ignoring unused variables or operations that do not contribute to the final result.

We adopt the CirC compiler to take part in this translation process. CirC can compile a modified version of the ZoKrates language called Z# into circuits used for SMT and ZKP. Specifically, we take advantage of the intermediate representation (IR) of CirC. YAP consumes this IR and transforms it into OpL. Processing the IR is a challenging process, as it includes a lot of auxiliary information that does not relate to our application (e.g., metadata). YAP processes the IR as follows:

- Eliminating unwanted blocks such as metadata and prime numbers that are used as moduli.
- Unfolding nested operations into a single operation per line.
- Converting binary values to integers.
- Mapping the index of a value into the variable holding that value.
- Converting array contents into variables, where each variable preserves its value and identity with respect to its source array. Hence, accessing an array index is equivalent to accessing the value of the variable of that index.
- Storing the intermediate results between operations.

After getting the OpL, the next step is creating the HE program that performs the computations defined in the OpL in the encrypted domain using SEAL. To achieve this, the PEEV framework includes two basic components: the first is the *Initializer* class, and the second is the *Circuit* class. The Initializer sets the

parameters required for Rinocchio and SEAL. Furthermore, it creates the required objects for wrapping up the parameters and the encoding objects for enabling batching. The Circuit class handles the creation of the circuits, the creation of R1CS, encrypting the values, and providing the front-end for executing the operations on encrypted data. In addition, the Circuit class provides necessary functions for generating and verifying ZKPs and returning the decrypted result.

Remarkably, a fundamental challenge in mapping FHE computations to ZKP arithmetic circuits is the handling of *relinearization* after ciphertext multiplication. Standard relinearization relies on complex Residue Number System (RNS) base-extension and rounding operations that are mathematically hostile to R1CS constraints, leading to exponential memory and constraint blowup if ignored. To strictly bound ciphertext expansion to two polynomials without introducing RNS complexity into the circuit, PEEV employs a “Prover Advice” paradigm via Base- W digit decomposition. Specifically, the cloud provider natively computes the FHE multiplication and safely decomposes the resulting extended polynomial into Base- W limbs using multi-precision Chinese Remainder Theorem (CRT) arithmetic. These limbs are then injected into the R1CS as private witness variables. The R1CS seamlessly verifies the decomposition via a native linear combination constraint, and subsequently computes the relinearization dot product against an array of custom, hard-coded Base- W evaluation keys. This architecture bypasses the need for in-circuit RNS rounding, drastically shrinking the SNARK constraint system.

Figure 4 summarizes the workflow of our proposed approach. The user writes the program in a high-level language, then CirC compiles it and generates the IR, before YAP parses the IR into OpL, and finally the OpL is used to create the arithmetic circuit and its R1CS definitions. Next, the Rinocchio key generation algorithm generates a proving and a verification key based on the R1CS. After executing the circuit, the cloud uses the proving key to generate the proof. Finally, the cloud user verifies the proof using the verification key and decrypts the result of the circuit.

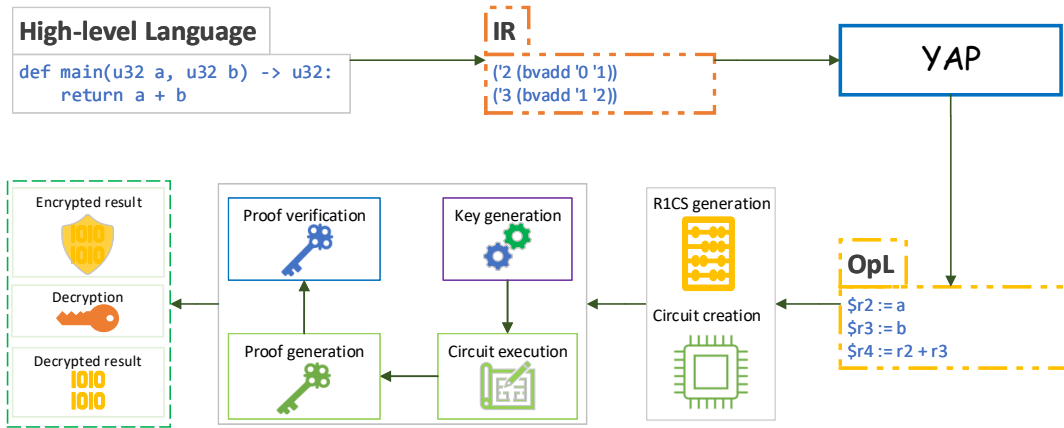


Fig. 4. Workflow of our verifiable privacy-preserving computation model. Starting with ZoKrates code, the program is converted to CirC’s IR, before YAP transforms the IR to OpL. Then, PEEV uses the OpL to generate the arithmetic circuit and the R1CS. After executing the circuit, a proof is generated and verified. Finally, if the proof is valid, PEEV decrypts the result.

PEEV employs the BGV scheme to perform leveled HE operation. This enables executing circuits with limited depth, but at the same time, providing faster running times. This makes our system more practical for applications that involve a finite number of additions and multiplications. To ensure complete mathematical binding between the FHE ciphertexts and the ZKP evaluation, we strictly unify the parameters across both primitives. For the experimental evaluation (next section), all benchmarks (except *Fibonacci v32*) use a

unified polynomial modulus degree of 2^{14} and a plaintext precision of 20 bits for both BGV and Rinocchio, which yields 128 bits of security.

Putting it all together, the three components for PEEV (the BGV scheme, the Rinocchio ZKP, and the CirC language) have been judiciously selected. Specifically, each of component offers several advantages that make them ideal candidates for designing a VPPC framework. On one hand, the Rinocchio ZKP enables compatibility with the BGV scheme, which leads to improved performance and low computational overhead. On the other hand, the CirC high-level language enables end users to effortlessly express their computations. Additionally, the intermediate representation of CirC allows for seamless translation into the corresponding FHE and ZKP backends.

6 Experimental Results

To evaluate PEEV, we employ benchmarks that involve different sets of mathematical operations such as addition, subtraction, and multiplication, including computing the Fibonacci sequence for 8, 16, and 32 iterations, square matrix multiplication for 2×2 and 3×3 matrices, the sum of squares for integers in range 1 to 8, 1 to 16, and 1 to 32, unscaled chi-squared, the summation of 8, 16, 32, and 64 values, vector dot-product of 8 and 16 values, the squared Euclidean distance of 8 and 16 values, and the factorial for $n = 5$ and 8. Additionally, we have implemented the logistic regression inference for inputs of 4, 8, and 10 features.

PEEV was evaluated on a Linux machine with an AMD EPYC 7532 32-Core Processor and 128 GiB of RAM. PEEV is designed for general-purpose use and our implementation is available as open-source software on GitHub.¹ Likewise, users can write and compile their custom programs using the YAParser, which is also available as open-source software on GitHub.² Both repositories include instructions for installing and running the framework.

Table 1 summarizes the execution times of PEEV evaluated across our different benchmarks. The *OpL to Circuit* column presents the time required for parsing the OpL file into SEAL and encrypting the values; meanwhile, the *Circuit&R1CS Generation* column lists the time required for creating the arithmetic circuit and its R1CS definition. The *Rinocchio Keys Generation* column shows the time needed for generating the proving and verification keys given the R1CS of a program, whereas the *Circuit Execution* column shows the time needed for performing the arithmetic operations over encrypted data and getting the final result. The time required for generating the proof, verifying it, and decrypting the result are listed in the *Proving*, *Verifying*, and *Decryption* columns, respectively. Finally, the last two columns show the total running times needed for the client and the server to execute each part of a program. Typically, a client converts the OpL into a circuit (OpL to Circuit), generates the R1CS (Circuit&R1CS Generation), verifies the results (Verifying), and decrypts the result (Decryption). Meanwhile, the server will perform the outsourced computations (Circuit Execution) and generate the proof (Proving). We assume that a third party that maintains a trusted environment handles the ZKP key generation process (Rinocchio Keys Generation).

Figure 5 visualizes the running times (ms) on a logarithmic scale of each benchmark. The most expensive component of the entire process is the proof generation. To strictly bound ciphertext expansion and optimize the SNARK backend, PEEV shifts the computational weight by performing Chinese Remainder Theorem (CRT) decomposition outside the circuit. While this slightly increases the local circuit execution time due to multi-precision arithmetic, it drastically reduces the size of the R1CS. Consequently, this architecture achieves significant speedups in the overall zero-knowledge Proving and Verifying phases.

We remark that the squared Euclidean distance can serve as the basis to perform privacy-preserving facial recognition [7]. Meanwhile, logistic regression is well-suited for binary classification tasks.

Overall, our list of benchmarks, where each benchmark is evaluated at different problem sizes, shows diversity of the applications that can be implemented in PEEV. For instance, the logistic regression model is tested on 4, 8, and 10 data points, with each variant exhibiting distinct execution times. Moreover, the diversity of the experiments is reflected not only in the programs themselves but also in the set of parameters

¹ <https://github.com/TrustworthyComputing/PEEV-verifiableFHE>

² <https://github.com/TrustworthyComputing/YAParser>

Table 1. Execution times of PEEV across different sets of programs including mathematical operations (additions, subtractions, and multiplications). All benchmarks use a plaintext bit size of 20 bits and a unified polynomial modulus degree of 2^{14} for SEAL and Rinocchio, except *Fibonacci v32* which uses a plaintext bit size of 32 bits and a polynomial modulus degree of 2^{15} .

Program	Execution Phase (ms)						Total Time (s)		
	OpL to Circuit	Circuit & R1CS Gen.	Rinocchio Keygen	Circuit Exec.	Proving	Verifying	Decryption	Client	Server
Fibonacci v8	49	31	6526	8	14778	5184	4	5.27	14.79
Fibonacci v16	52	26	11060	19	58580	20048	3	20.13	58.60
Fibonacci v32	242	137	138500	263	617629	219460	14	219.85	617.89
MMul 2x2	231	623	58696	1097	392811	136678	4	137.54	393.91
MMul 3x3	145	1629	256410	2962	2216651	774866	4	776.64	2219.61
Sum Sqrs v8	288	547	34649	871	190522	74977	4	75.82	191.39
Sum Sqrs v16	435	1454	87354	1742	604747	222310	4	224.20	606.49
Sum Sqrs v32	819	2071	255508	4504	2077050	743734	4	746.63	2081.55
Chi Squared	138	290	19626	439	99173	31771	4	32.20	99.61
Sum v8	224	89	7795	11	26292	9979	4	10.30	26.30
Sum v16	258	92	12236	20	69359	25000	7	25.36	69.38
Sum v32	455	160	27382	40	165715	65735	4	66.35	165.76
Sum v64	830	302	65847	78	502964	195628	5	196.77	503.04
Dot Product v8	218	518	35042	1554	192793	68315	3	69.05	194.35
Dot Product v16	446	1015	86442	1761	606852	215640	4	217.11	608.61
Euc. Distance v8	252	546	42363	880	258784	93517	4	94.32	259.66
Euc. Distance v16	447	1119	121000	1806	861994	290253	4	291.82	863.80
Factorial v5	90	333	14755	533	69885	24839	3	25.27	70.42
Factorial v8	129	484	25865	852	136680	46569	4	47.19	137.53
Logistic Reg. v4	219	687	29782	753	157428	56228	3	57.14	158.18
Logistic Reg. v8	408	1121	50785	1202	318962	111057	3	112.59	320.16
Logistic Reg. v10	323	896	68180	1437	416097	149599	3	150.82	417.53

used for encryption. Our experiments are repeatable in different settings and environments using the open-source code of PEEV.

We remark that the judicious selection of PEEV’s components plays an important role in achieving fast execution times; in particular, the compatibility between Rinocchio and the BGV scheme leads to a harmonious computational model. Besides performance optimizations, PEEV not only allows repeatable experiments in different settings but also supports evaluating arbitrary programs, beyond our list of benchmarks.

7 Discussion

Our experimental results provide insights on the time required for each step. Overall, with respect to integrity, the two most expensive steps are the proving step, which is performed on the server side, and the verification step, which is performed on the user side. For example, consider the sum of squares program with 16 values; if PEEV omits the proving step, the server will only execute the circuit, reducing the server’s time to 1742 ms. Similarly, on the user side, omitting the verification step will decrease the user’s total setup and decryption time to 1893 ms. As for SEAL keys generation, the overhead is negligible and can be performed locally at the user side.

Remarkably, PEEV does not incur a large communication overhead. There is no *interactive* communication between the client and server (i.e., the client does need to send and receive messages to and from the

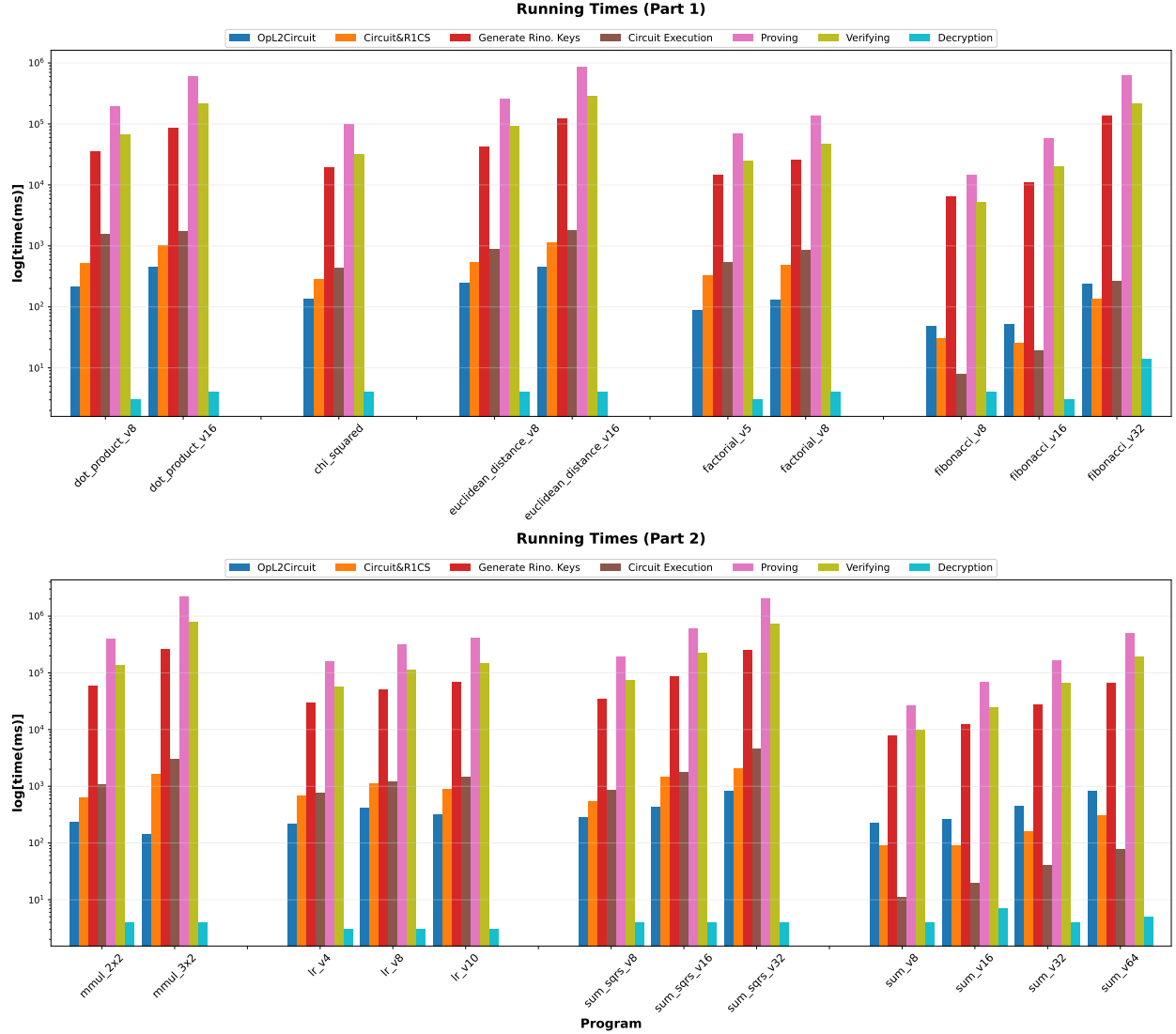


Fig. 5. Execution times of each operation in the benchmark: The vertical axis shows the time in milliseconds, while the horizontal axis corresponds to each benchmark set.

server in real-time). Instead, the client sends a circuit and encrypted data to a server, and later the server responds with the result and the proof.

In our analysis, the only three steps that are not explicitly measured are the compilation time, which is negligible, the SEAL key generation is also negligible, and the time for sending the data from a host to a server, which depends on the network conditions and the size of the data. In practice, given large amounts of data, the transmission from a host to a server happens incrementally, since the data is not transferred all at once.

Baseline computation costs are typically assessed without the additional privacy and integrity measures provided by homomorphic encryption and ZKP protocols. In such cases, evaluations are performed on unencrypted arithmetic circuits, and thus, computations are carried out in the plaintext domain rather than in the space of polynomial rings, leading to faster execution times. Alas, such baseline evaluations do not provide any privacy: since all the computations are exposed to the cloud provider, there is no guarantee that user

data will remain secure. Likewise, without enabling integrity, there is no guarantee that the computations are performed faithfully.

As reported in our experiments, executing the *sum of squares* benchmark with PEEV over 16 values requires 1742 milliseconds (Table 1). Conversely, running the same computation without encryption and integrity protections on the same machine requires a few microseconds (about 18 microseconds). As expected, running the program on plaintext values without protections is faster than its encrypted counterpart. Nevertheless, this corresponds to a trade-off between execution overhead vs. security: disabling the privacy and integrity verification exposes user data to the cloud server, which may not be an option in most scenarios.

Table 2 presents a comparison between PEEV and multiple similar frameworks. As shown in the table, PEEV is a general-purpose framework that employs a leveled HE scheme and supports verifiable computation using the Rinocchio ZKP. Additionally, PEEV features a high-level front-end to enhance usability. In contrast, HELM [29] is a general-purpose framework utilizing a fully HE scheme, and its main limitation is the absence of a verification mechanism and a high-level front-end. Similarly, ArctyrEX [31] is a general-purpose framework that targets a fully HE scheme and offers a high-level front-end, but also lacks a verification mechanism. GALA [71] and REDsec [23] are specialized systems for running encrypted neural networks. Specifically, GALA employs a leveled HE scheme, whereas REDsec utilizes a fully HE scheme; unlike PEEV, neither GALA nor REDsec offers a verification method or a high-level front-end. The work of Zilch [52] concentrates on verifiable computations solely on public data, and it is not employing any HE schemes to enable confidentiality. Zilch incorporates a zk-STARK ZKP system in its backend and provides a high-level front-end. The work in [34] introduces a privacy-preserving learning scheme exclusively for the support vector machine (SVM) model. This SVM training scheme utilizes a partial HE scheme and verification tags to verify computations. Lastly, pvCNN [67] is a specific-purpose scheme for convolution neural network (CNN) models; pvCNN employs a leveled HE scheme (referred to by the authors as L-FHE), and the Groth16 verification protocol, but lacks a high-level front-end.

Table 2. A comparison between our proposed system, PEEV, and seven recent similar works, demonstrating that PEEV is the only general-purpose framework addressing verifiability, homomorphic data encryption, and usability through a high-level front-end.

Work	Purpose	HE	Verification	High-level Front-End
PEEV (Ours)	General-purpose	Leveled – BGV	Rinocchio	Yes
HELM [29]	General-purpose	Fully – CGGI	N/A	No
ArctyrEX [31]	General-purpose	Fully – CGGI	N/A	Yes
GALA [71]	Neural networks	Leveled – BFV	N/A	No
REDsec [23]	Neural networks	Fully – CGGI	N/A	No
Zilch [52]	Verifiable computations	N/A	zk-STARK	Yes
[34]	SVM training	Partial – Paillier	Verification tags	No
pvCNN [67]	CNN	Leveled HE	Groth16	No

8 Practical Considerations

The proposed methodology can readily scale to any problem size; however, thresholds arise from both the SEAL HE backend and Rinocchio. While PEEV successfully introduces custom Base-W relinearization to strictly bound ciphertext sizes and make them compatible with the ZKP backend, deep multiplicative circuits force SEAL to use a massive prime modulus (Q) to maintain the noise budget. A massive Q requires a higher number of decomposition limbs during relinearization. Because each limb is injected as a private ZKP witness,

executing deep circuits significantly increases the number of variables in the R1CS, which can eventually exhaust the available system memory (RAM) during constraint generation.

9 Conclusion and Future Work

This paper introduces the PEEV framework for verifiable privacy-preserving computations. PEEV allows end users to write programs that process encrypted data without having extensive knowledge of cryptography, while also enabling computations performed by a remote server to be verified. We use the BGV scheme to encrypt and process the end user’s data, as well as zk-SNARKs for generating proofs; in particular, PEEV employs Microsoft SEAL as its homomorphic encryption back-end and Rinocchio as its ZKP system. To realize PEEV, we introduce the novel bespoke YAP parser that enables translation from a high-level language into the OpL intermediate representation. The OpL syntax is characterized by its simplicity and readability, which makes it easy to parse in different FHE libraries, as well as extend with new operations. To evaluate the efficiency of our system, we employ a broad range of encrypted programs, and report low performance overheads both for encrypted computation and proof generation.

Acknowledgments

The authors would like to acknowledge the support of the National Science Foundation (Award #2239334). We also thank the anonymous reviewers and Adrian Heath for the feedback.

References

- [1] A. Bampoulidis et al. “Privately connecting mobility to infectious diseases via applied cryptography”. In: *arXiv preprint arXiv:2005.02061* (2020).
- [2] M. Barbosa and P. Farshim. “Delegatable homomorphic encryption with applications to secure outsourcing of computation”. In: *Topics in Cryptology–CT-RSA 2012: The Cryptographers’ Track at the RSA Conference 2012, San Francisco, CA, USA, February 27–March 2, 2012. Proceedings*. Springer. 2012, pp. 296–312.
- [3] E. Ben-Sasson et al. “Scalable zero knowledge with no trusted setup”. In: *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III* 39. Springer. 2019, pp. 701–732.
- [4] E. Ben-Sasson et al. “SNARKs for C: Verifying program executions succinctly and in zero knowledge”. In: *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2013. Proceedings, Part II*. Springer. 2013, pp. 90–108.
- [5] A. Benaissa et al. *TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption*. 2021. arXiv: 2104.03152 [cs.CR].
- [6] M. Blum, P. Feldman, and S. Micali. “Non-interactive zero-knowledge and its applications”. In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. 1988, pp. 103–112.
- [7] V. N. Boddeti. “Secure face matching using fully homomorphic encryption”. In: *2018 IEEE 9th international conference on biometrics theory, applications and systems (BTAS)*. IEEE. 2018, pp. 1–10.
- [8] A. Bois et al. “Flexible and efficient verifiable computation on encrypted data”. In: *IACR International Conference on Public-Key Cryptography*. Springer. 2021, pp. 528–558.
- [9] C. Bonte et al. *FINAL: Faster FHE instantiated with NTRU and LWE*. Cryptology ePrint Archive, Paper 2022/074. <https://eprint.iacr.org/2022/074>. 2022.
- [10] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. “(Leveled) fully homomorphic encryption without bootstrapping”. In: *ACM Transactions on Computation Theory (TOCT)* 6.3 (2014), pp. 1–36.
- [11] B. Bünz et al. “Bulletproofs: Short Proofs for Confidential Transactions and More”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020.
- [12] S. Carpov, P. Dubrulle, and R. Sirdey. “Armadillo: a compilation chain for privacy preserving applications”. In: *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. 2015, pp. 13–19.
- [13] S. Chatel et al. “Verifiable encodings for secure homomorphic analytics”. In: *arXiv preprint arXiv:2207.14071* (2022).

- [14] H. Chen, I. Chillotti, and Y. Song. “Improved bootstrapping for approximate homomorphic encryption”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2019, pp. 34–54.
- [15] J. H. Cheon et al. “Homomorphic encryption for arithmetic of approximate numbers”. In: *Advances in Cryptology—ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3–7, 2017, Proceedings, Part I 23*. Springer. 2017, pp. 409–437.
- [16] I. Chillotti et al. “Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds”. In: *Advances in Cryptology—ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4–8, 2016, Proceedings, Part I 22*. Springer. 2016, pp. 3–33.
- [17] I. Chillotti et al. “TFHE: fast fully homomorphic encryption over the torus”. In: *Journal of Cryptology* 33.1 (2020), pp. 34–91.
- [18] L. Coppolino et al. “Vise: Combining intel sgx and homomorphic encryption for cloud industrial control systems”. In: *IEEE Transactions on Computers* 70.5 (2020), pp. 711–724.
- [19] V. Costan, I. Lebedev, and S. Devadas. “Sanctum: Minimal hardware extensions for strong software isolation”. In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 857–874.
- [20] A. J. Duncan, S. Creese, and M. Goldsmith. “Insider attacks in cloud computing”. In: *2012 IEEE 11th international conference on trust, security and privacy in computing and communications*. IEEE. 2012, pp. 857–862.
- [21] J. Fan and F. Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2012/144. <https://eprint.iacr.org/2012/144>. 2012.
- [22] D. Fiore, R. Gennaro, and V. Pastro. “Efficiently verifiable computation on encrypted data”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 844–855.
- [23] L. Folkerts, C. Gouert, and N. G. Tsoutsos. “REDsec: Running encrypted discretized neural networks in seconds”. In: *Cryptology ePrint Archive* (2021).
- [24] C. Ganesh, A. Nitulescu, and E. Soria-Vazquez. *Rinocchio: SNARKs for ring arithmetic*. 2023.
- [25] C. Gentry. “Fully homomorphic encryption using ideal lattices”. In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 2009, pp. 169–178.
- [26] C. Gentry, S. Halevi, and N. P. Smart. “Better bootstrapping in fully homomorphic encryption”. In: *International Workshop on Public Key Cryptography*. Springer. 2012, pp. 1–16.
- [27] S. Goldwasser, S. Micali, and C. Rackoff. “The knowledge complexity of interactive proof-systems”. In: *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*. 2019, pp. 203–225.
- [28] S. Gorantala et al. *A General Purpose Transpiler for Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2021/811. <https://eprint.iacr.org/2021/811>. 2021.
- [29] C. Gouert, D. Mouris, and N. G. Tsoutsos. *HELM: Navigating Homomorphic Encryption through Gates and Lookup Tables*. Cryptology ePrint Archive, Paper 2023/1382. <https://eprint.iacr.org/2023/1382>. 2023.
- [30] C. Gouert, D. Mouris, and N. G. Tsoutsos. “SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks”. In: *Proceedings on Privacy Enhancing Technologies* 2023.3 (July 2023), pp. 154–172. DOI: 10.56553/popets-2023-0075.
- [31] C. Gouert et al. “ArctyrEX: Accelerated Encrypted Execution of General-Purpose Applications”. In: *arXiv preprint arXiv:2306.11006* (2023).
- [32] I. Grigoriadis et al. “Machine Learning as a Service (MLaaS)—An Enterprise Perspective”. In: *Proceedings of International Conference on Data Science and Applications: ICDSA 2022, Volume 2*. Springer. 2023, pp. 261–273.
- [33] S. Halevi and V. Shoup. “Design and implementation of HELib: a homomorphic encryption library”. In: *Cryptology ePrint Archive* (2020).
- [34] C. Hu et al. “Achieving Privacy-Preserving and Verifiable Support Vector Machine Training in the Cloud”. In: *IEEE Transactions on Information Forensics and Security* (2023).
- [35] R. M. Jabir et al. “Analysis of cloud computing attacks and countermeasures”. In: *2016 18th international conference on advanced communication technology (ICACT)*. IEEE. 2016, pp. 117–123.
- [36] W. Jung et al. “Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), pp. 114–148.

- [37] V. Kiriansky et al. “DAWG: A defense against cache timing attacks in speculative execution processors”. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 974–987.
- [38] A. Konkin and S. Zapechnikov. “Zero knowledge proof and ZK-SNARK for private blockchains”. In: *Journal of Computer Virology and Hacking Techniques* (2023), pp. 1–7.
- [39] J. Lai et al. “Verifiable computation on outsourced encrypted data”. In: *Computer Security-ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I 19*. Springer. 2014, pp. 273–291.
- [40] *Lattigo v4*. Online: <https://github.com/tuneinsight/lattigo>. EPFL-LDS, Tune Insight SA. Aug. 2022.
- [41] S. Li, X. Wang, and R. Xue. “Toward Both Privacy and Efficiency of Homomorphic MACs for Polynomial Functions and Its Applications”. In: *The Computer Journal* 65.4 (2022), pp. 1020–1028.
- [42] S. Li, X. Wang, and R. Zhang. “Privacy-Preserving Homomorphic MACs with Efficient Verification”. In: *Web Services-ICWS 2018: 25th International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings 16*. Springer. 2018, pp. 100–115.
- [43] F. Liu et al. “Catalyst: Defeating last-level cache side channel attacks in cloud computing”. In: *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE. 2016, pp. 406–418.
- [44] Z. Liu et al. “PPRU: A privacy-preserving reputation updating scheme for cloud-assisted vehicular networks”. In: *IEEE Transactions on Vehicular Technology* (2023).
- [45] Q. Lou et al. “vFHE: Verifiable Fully Homomorphic Encryption with Blind Hash”. In: *arXiv:2303.08886* (2023).
- [46] F. Luo and K. Wang. “Verifiable decryption for fully homomorphic encryption”. In: *International Conference on Information Security*. Springer. 2018, pp. 347–365.
- [47] V. Lyubashevsky, N. K. Nguyen, and M. Plançon. “Lattice-based zero-knowledge proofs and applications: shorter, simpler, and more general”. In: *Annual International Cryptology Conference*. Springer. 2022, pp. 71–101.
- [48] V. Lyubashevsky, C. Peikert, and O. Regev. “On ideal lattices and learning with errors over rings”. In: *Advances in Cryptology-EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30-June 3, 2010. Proceedings 29*. Springer. 2010, pp. 1–23.
- [49] G. K. Mahato and S. K. Chakraborty. “A comparative review on homomorphic encryption for cloud security”. In: *IETE Journal of Research* 69.8 (2023), pp. 5124–5133.
- [50] M. Masdari and M. Jalali. “A survey and taxonomy of DoS attacks in cloud computing”. In: *Security and Communication Networks* 9.16 (2016), pp. 3724–3751.
- [51] Y. Miao et al. “Efficient privacy-preserving spatial range query over outsourced encrypted data”. In: *IEEE Transactions on Information Forensics and Security* (2023).
- [52] D. Mouris and N. G. Tsoutsos. “Zilch: A framework for deploying transparent zero-knowledge proofs”. In: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 3269–3284.
- [53] D. Mouris, N. G. Tsoutsos, and M. Maniatakis. “TERMinator Suite: Benchmarking Privacy-Preserving Architectures”. In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 122–125. DOI: 10.1109/LCA.2018.281281.
- [54] D. Natarajan et al. “Chex-mix: Combining homomorphic encryption with trusted execution environments for two-party oblivious inference in the cloud”. In: *Cryptology ePrint Archive* (2021).
- [55] A. Ozdemir, F. Brown, and R. S. Wahby. *CirC: Compiler infrastructure for proof systems, software verification, and more*. Cryptology ePrint Archive, Paper 2020/1586. <https://eprint.iacr.org/2020/1586>. 2020.
- [56] H. Pallathadka et al. “An investigation of various applications and related challenges in cloud computing”. In: *Materials Today: Proceedings* 51 (2022), pp. 2245–2248.
- [57] P. Papadopoulos et al. “If you are not paying for it, you are the product: How much do advertisers pay to reach you?” In: *Proceedings of the 2017 Internet Measurement Conference*. 2017, pp. 142–156.
- [58] F. J. G. Peñalvo et al. “Mobile cloud computing and sustainable development: Opportunities, challenges, and future directions”. In: *International Journal of Cloud Applications and Computing (IJCAC)* 12.1 (2022), pp. 1–20.
- [59] O. Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40.
- [60] U. R. Saxena and T. Alam. “Role-based access using partial homomorphic encryption for securing cloud data”. In: *International Journal of System Assurance Engineering and Management* 14.3 (2023), pp. 950–966.
- [61] *Microsoft SEAL (release 4.1)*. <https://github.com/Microsoft/SEAL>. Microsoft Research, Redmond, WA. Jan. 2023.
- [62] X. Sun et al. “A survey on zero-knowledge proof in blockchain”. In: *IEEE network* 35.4 (2021), pp. 198–205.

- [63] N. Toumi, M. Bagaa, and A. Ksentini. “Machine Learning for Service Migration: A Survey”. In: *IEEE Communications Surveys & Tutorials* (2023).
- [64] A. Viand, C. Knabenhans, and A. Hithnawi. “Verifiable fully homomorphic encryption”. In: *arXiv preprint arXiv:2301.07041* (2023).
- [65] S. Vinoth et al. “Application of cloud computing in banking and e-commerce and related security threats”. In: *Materials Today: Proceedings* 51 (2022), pp. 2172–2175.
- [66] W. Wang et al. “Toward scalable fully homomorphic encryption through light trusted computing assistance”. In: *arXiv preprint arXiv:1905.07766* (2019).
- [67] J. Weng et al. “pvcnn: Privacy-preserving and verifiable convolutional neural network testing”. In: *IEEE Transactions on Information Forensics and Security* 18 (2023), pp. 2218–2233.
- [68] W. Yin. “Zero-Knowledge Proof Intelligent Recommendation System to Protect Students’ Data Privacy in the Digital Age”. In: *Applied Artificial Intelligence* 37.1 (2023), p. 2222495.
- [69] Zama. *Concrete: TFHE Compiler that converts python programs into FHE equivalent*. <https://github.com/zama-ai/concrete>. 2022.
- [70] Zama. *TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data*. <https://github.com/zama-ai/tfhe-rs>. 2022.
- [71] Q. Zhang, C. Xin, and H. Wu. “GALA: Greedy computation for linear algebra in privacy-preserved neural networks”. In: *arXiv preprint arXiv:2105.01827* (2021).