IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 12 June 2024, accepted 29 June 2024, date of publication 8 July 2024, date of current version 18 July 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3424420

RESEARCH ARTICLE

PEEV: Parse Encrypt Execute Verify–A Verifiable FHE Framework

OMAR AHMED, CHARLES GOUERT, (Student Member, IEEE), AND NEKTARIOS GEORGIOS TSOUTSOS, (Member, IEEE)
Electrical and Computer Engineering Department, University of Delaware, Newark, DE 19716, USA
Corresponding author: Nektarios Georgios Tsoutsos (tsoutsos@udel.edu)

This work was supported by the National Science Foundation under Grant 2239334.

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.

INDEX TERMS Cloud computing, homomorphic encryption, private and verifiable computation, zero-knowledge proofs.

I. 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 [57], [59], [67]. 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 [58]. In addition, cloud computing is susceptible to a variety of cyberattacks including network attacks and account hijacking [21], [36], [51].

While numerous research efforts have been proposed to counter cloud attacks [19], [38], [44], 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.

The associate editor coordinating the review of this manuscript and approving it for publication was Jiafeng Xie.

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:

$$z = F(x, y) = x + y \tag{1}$$

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:

$$z' = F'(x', y') = x' + y' \tag{2}$$

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

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

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.

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

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], [41], [62], 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], [29], [30], [31], [54], [71].

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 [33], [64]. 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 [28]. ZKPs have gained much attention and improved over the years due to their importance in verifiable computation [39], [48], [53], [70].

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 II discusses prior works addressing the problem of verifiable computation, while Section III provides a background on homomorphic encryption schemes and libraries, zero-knowledge proofs, and compilers for HE and ZKPs. Section IV introduces the proposed approach for achieving verifiable privacy-preserving computation on encryption data, while Section V describes the implementation details of the framework. Section VI presents the experimental results over representative benchmarks. Finally, a discussion of practical considerations and the concluding remarks are presented in Sections VIII and IX, respectively.

II. 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 confidentially on the user's input/output data, i.e., they are provided as clear text. Moreover, the class of functions that can 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) [40] 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], [42]. 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 [23]. 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 [46] 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 [47]. 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], [55], [68], as well as verifying the integrity of a computation based on MACs [13], [43]. 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 [45] used Elliptic Curve Cryptography and Paillier cryptosystem to introduce a reputation updating scheme vehicular networks. In addition, the authors of [52] 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.

III. BACKGROUND

A. 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 [61]. Examples of PHE include the unpadded RSA, ElGamal, and Paillier cryptosystems [50].
- **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) [60] and ring learning with errors (RLWE) [49] 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 accomodate 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) [22], 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 [26]. Nevertheless, bootstrapping is a very costly technique, being over an order of magnitude slower than other HE operations. Even with proposed optimizations [14], [27], [37], 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.

B. 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 [72].
- **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 [34]. 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 [41].
- **SEAL**. Microsoft released its own HE library called SEAL [62]. 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.

C. ZERO KNOWLEDGE PROOFS

ZKPs represent a major innovation in applied cryptography and are used extensively in blockchains and cryptocurrency [63]. They were first introduced in 1985 [28] 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].

D. 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 [56]. 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 [31]. 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 [54]. 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 [30]. 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 [71]. 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.

IV. 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.

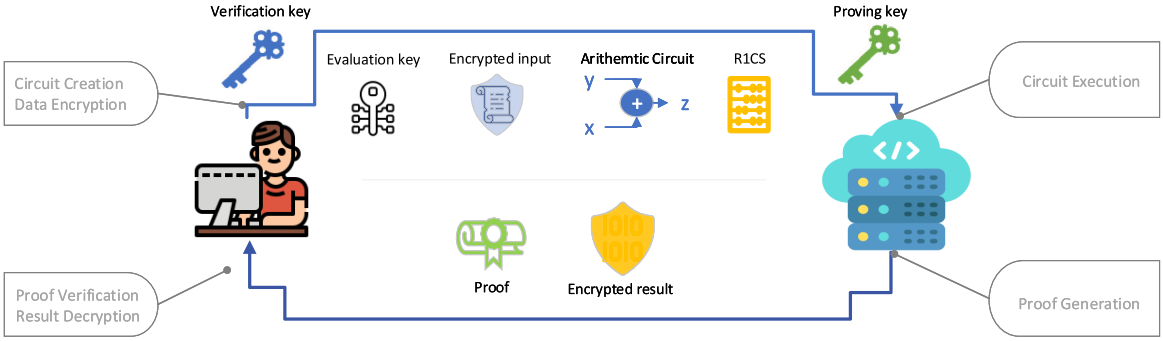


FIGURE 1. Client-Server Communication: The client creates the arithmetic circuit, R1CS, and evaluation key, encrypts the input, and sends them to the server. The server runs the computation, generates the proof, and sends the result and the proof back to the client. The client checks if the proof is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

- Eval(C, V.Prov, V.Ver, E.Eval) $\rightarrow$ (γ, ρ) - This is the procedure that will be performed by the server and outputs the encrypted result, γ, and the proof, ρ.
- Conc(V.Ver, E.Dec, γ, ρ) $\rightarrow$ *Res* - The user runs this procedure to verify the proof, ρ, using the verification key, V.Ver. If the proof is valid, the procedure returns the result, Res, decrypted using the decryption key, E.Dec. Otherwise, Res is returned as *None*.

A. 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.

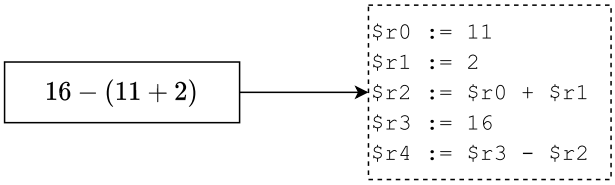


FIGURE 2. YAP flattens the equation 16 − (11 + 2) into a sequence of operations. The OpL lists every single operation in a single line, and subsequent lines are dependent on previous lines.

- The 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.

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.

B. 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

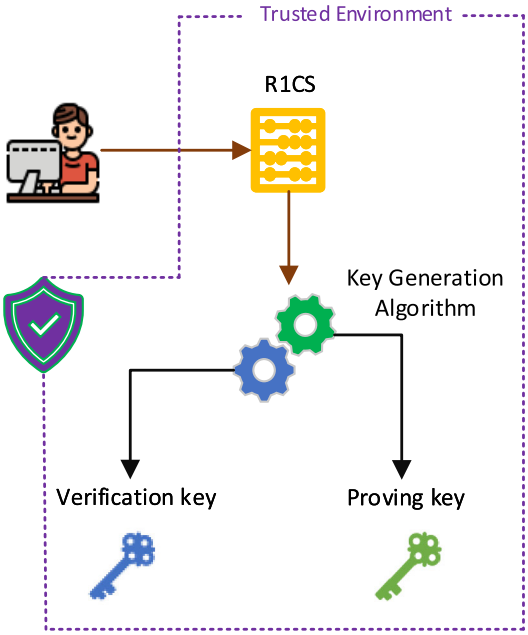


FIGURE 3. The cloud user inputs the R1CS to the ZKP key generation algorithm. The key generation process requires a trusted environment, which can be achieved through a third-party who offers a trusted setup. The algorithm outputs two keys: one for the cloud to generate the proof, and the other for the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

and ZKP libraries. For instance, PEEV employs SEAL and 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.

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.

C. 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 $f : X \longrightarrow Y$. 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.

D. 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 [25] and the BGV scheme [10]; for more comprehensive information about both schemes, we refer the reader to the original papers.

1) BGV HOMOMORPHIC ENCRYPTION SCHEME

Encryption schemes inherent 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:

Definition 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 $R \; mod \; q$ 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 $b_i =< \mathbf{a}_i, \mathbf{s} > +e_i$.

  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 $N = n \times polylog(q)$.
- 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 $R_q^{N \times n}$, $\mathbf{b} = \mathbf{Bt} + 2\mathbf{e}$, 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.

2) 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^λ, 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 $\mathbb{F}$. 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 right input polynomial $V(x)$, the left input polynomial $W(x)$, and the output polynomial $Y(x)$, one must find the

target polynomial $t(x)$ that divides $(V(x) \cdot W(x) - Y(x))$.

$$t(x)|(V(x) \cdot W(x) - Y(x)) \tag{5}$$

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.

3) PEEV SECURITY

Putting it all together, PEEV is a secure VPPC framework as long as the underlying assumptions of the BGV scheme and the secure encoding of Rinocchio hold. Specifically, if there exists an adversary $\mathbb{A}$ who can compute a solution to the LWE or RLWE problem, then $\mathbb{A}$ would break the security assumptions of the BGV scheme, and thus PEEV. Nevertheless, this is a contradiction, since LWE/RLWE are assumed to be intractable, so BGV and PEEV are therefore secure. Likewise, if we assume the instantiation of Rinocchio with polynomial rings or the QRP structure is vulnerable, then PEEV would also insecure; however, this is a contradiction since Rinocchio is provably secure. In short, the security assumptions cascade from LWE and RLWE to the BGV scheme, and from the BGV scheme to PEEV; similarly, the security of the QRP cascades to the Rinocchio protocol and is inherited in PEEV.

V. IMPLEMENTATION DETAILS

A. 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 [25]. 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 [65].

B. 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, performing ciphertext maintenance operations

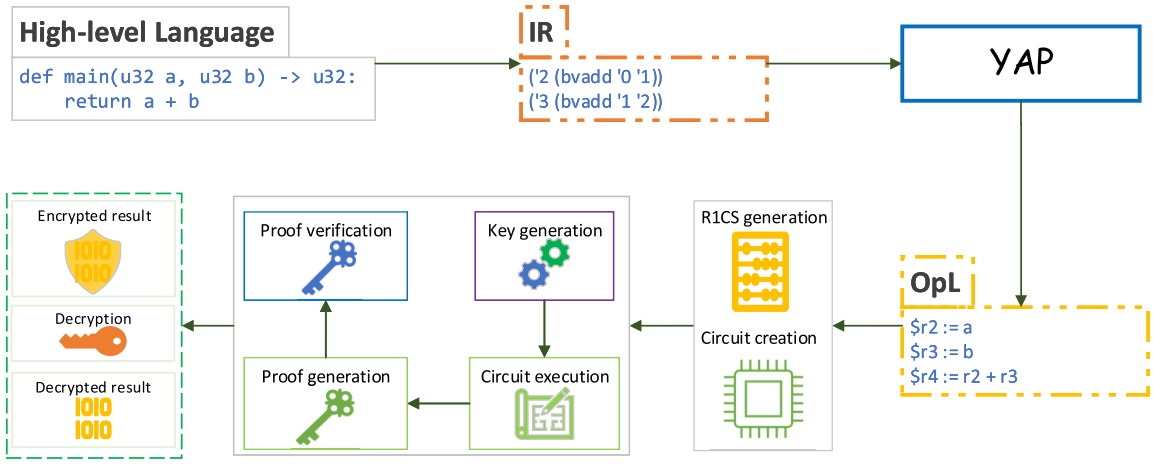


FIGURE 4. Workflow of our verifiable privacy-preserving computation model. Starting with ZoKrates code, the program is converted to CirC's IR, before YAP transforms the IR to OpL. Then, PEEV uses the OpL to generate the arithmetic circuit and the R1CS. After executing the circuit, a proof is generated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

such as relinearization, and providing the front-end for executing the operations on encrypted data. Remarkably, the *relinearization* is a necessary step in homomorphic computation that solves a key issue when multiplying two ciphertexts. After ciphertext multiplication, the product ciphertext will be approximately 50% larger in size and can no longer be decrypted under the original secret key. Relinearization will map the larger product ciphertext back to the original ciphertext size and also result in a valid ciphertext encrypted under the original key [22]. In addition, the Circuit class provides necessary functions for generating and verifying ZKPs and returning the decrypted result.

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.

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. For the experimental evaluation (next section), we use a polynomial modulus degree of 2^{14} and plaintext precision of 30 bits in SEAL, which yields 128 bits of security. Meanwhile, Rinocchio uses a polynomial modulus degree of 2^{11} and plaintext precision of 30 bits (also 128 bits of security). PEEV uses such a large polynomial modulus degree for SEAL to allow more complicated encrypted computations.

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.

VI. 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, 32, and 64 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, chi-squared, the summation of 8, 16, 32, and 64 values, vector dot-product of 8, 16, and 32 values, the squared Euclidean distance of 8, 16, and 32 values, the factorial for $n = 5$, 8, and 12, and the Hamming distance of 4, 6, and 8 values.

Additionally, we have implemented three common machine learning algorithms, including the logistic regression inference for three data points of 4, 8, and 10 features, the cubic spline regression given 4 and 8 data points, and the perceptron training algorithm for three data points with 4 features for one iteration.

PEEV was evaluated using a Windows laptop equipped with a 6th generation Intel i5 processor at 2.30 GHz and 16 GB 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

[1] https://github.com/TrustworthyComputing/PEEV-verifiableFHE

also available as open-source software on GitHub.[2] Both repositories include instructions for installing and running the framework.

The factorial and Hamming distance programs use different parameters from other programs. For the factorial, we employ larger parameters to support larger plaintext precision in SEAL and avoid overflow; we set the polynomial modulus degree to 2^{15} and the plaintext bit size to 32 bits in this case. Meanwhile, the Hamming distance program uses a polynomial modulus degree of 2^{11} with plaintext precision of 20 bits for Rinocchio, and a polynomial modulus degree of 2^{15} with plaintext modulus value of 13 for SEAL. The Hamming distance uses a small plaintext modulus value and a large polynomial modulus degree to support computing the equality check operation (e.g., $x == y$); this operation requires exponentiation of the encrypted difference between two values to the value of the plaintext modulus -1; this yields 1 if the two values are not equal and 0 otherwise. Notably, the multiplicative depth of the equality operation scales linearly with the plaintext modulus, necessarily requiring a smaller precision for efficient LHE operations. Also, we disabled batching for the Hamming distance program, as it requires the plaintext modulus to be a prime number congruent to 1 modulo $2 \times N$, where N is the polynomial modulus degree. Similarly, the logistic regression, the cubic spline, and the perceptron programs use another set of parameters to support larger plaintext and deeper circuits; they use a plaintext bit size of 42 bits for both SEAL and Rinocchio and a polynomial degree 2^{15} for SEAL.

Table 1 summarizes the execution times of PEEV evaluated across 32 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).

[2]https://github.com/TrustworthyComputing/YAParser

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; for Fibonacci v64, 3×3 matrix multiplication, sum v64, vector dot product v32, and factorial v12, proof generation takes more than 3 seconds. However, it takes about 6 seconds for the Euclidean distance v32 and the hamming distance v8. The time needed for executing each circuit is short, which reflects the benefits of SEAL as a HE back-end; the longest execution time is about 3 seconds for the factorial program for 12 encrypted values. The execution time of the Hamming distance program is orders of magnitude longer than other programs due to the fact that batching needed to be disabled to enable feasible equality. For the logistic regression, the cubic spline, and the perceptron programs the proof generation takes time less than the execution of the circuit; this is due to larger plaintext size.

We remark that the squared Euclidean distance can serve as the basis to perform privacy-preserving facial recognition [7]. Meanwhile, both logistic regression and perceptrons are well-suited for binary classification tasks, while cubic spline regression can model non-linear functions.

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

Our experimental results demonstrate how PEEV exhibits optimized performance for key applications: for instance, the Hamming distance benchmark is widely used in error detection and correction algorithms [20], machine learning algorithms such as clustering [66], and cryptography. 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.

VII. 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 32 values; if PEEV omits the proving step, the server will only execute the circuit, reducing the server's time to 1329 ms. Similarly, on the user side, omitting the verification step will decrease the user's total time to 2127 ms.

TABLE 1. Execution times of PEEV across different sets of programs that include mathematical operations such as additions, subtractions, and multiplications (all times are in milliseconds). The last two columns show the total time needed for the client and the server, respectively. All the benchmarks use a plaintext bit size of 30 bits, a polynomial modulus degree of 2^{11} for Rinocchio, and 2^{14} for SEAL, except factorial that uses a plaintext bit size of 32 bits and a polynomial modulus degree of 2^{15} for SEAL, and hamming distance that uses a plaintext modulus value of 13 and a polynomial modulus degree of 2^{15} for SEAL. Logistic regression, cubic spline, and perceptron use 42 bits for the plaintext size.

Program	OpL to Circuit	Circuit&R1CS Generation	Rinocchio Keygen	Circuit Execution	Proving	Verifying	Decryption	Client Time(ms)	Server Time (ms)
fibonacci v8	94	9	133	7	124	56	22	181	131
fibonacci v16	94	11	225	16	365	138	27	270	381
fibonacci v32	166	22	679	52	1507	724	24	936	1559
fibonacci v64	124	12	942	75	3161	1267	29	1432	3236
mmul 2x2	504	37	314	413	768	282	24	847	1181
mmul 3x3	1343	79	972	1260	3685	1346	25	2793	4945
sum sqrs v8	508	32	217	306	381	144	20	704	687
sum sqrs v16	983	61	432	616	1223	468	24	1536	1839
sum sqrs v32	1985	119	917	1329	3371	1295	23	3422	4700
chi squared	419	30	235	415	416	161	22	632	831
sum v8	317	24	153	7	137	63	23	427	144
sum v16	571	38	244	13	465	183	24	816	478
sum v32	1111	66	574	31	1335	524	24	1725	1366
sum v64	2110	123	956	53	3354	1341	27	3601	3407
dot product v8	575	67	271	360	478	178	23	843	838
dot product v16	999	59	438	627	1248	474	24	1556	1875
dot product v32	1979	112	932	1329	3304	1266	28	3385	4633
euc. distance v8	529	36	346	323	833	355	22	942	1156
euc. distance v16	1131	93	898	693	2146	802	24	2050	2839
euc. distance v32	1999	117	1434	1357	6208	2307	23	4446	7565
factorial v5	699	50	114	1233	71	35	82	866	1304
factorial v8	1016	69	134	1890	121	73	89	1247	2011
factorial v12	1790	120	246	3038	228	133	138	2181	3266
ham. dist. v4	3245	205	744	8148	2878	922	85	4457	11026
ham. dist. v6	4403	285	1377	12283	4971	1696	85	6469	17254
ham. dist. v8	5762	348	1602	14776	6890	2658	85	8853	21666
logistic reg. v4	2629	215	937	5250	1473	560	89	3493	6723
logistic reg. v8	5294	278	1065	8930	4193	1749	87	7408	13123
logistic reg. v10	6115	323	1583	10671	5583	2064	86	8588	16254
cubic spline v4	2651	158	598	5550	1862	764	84	3657	7412
cubic spline v8	4772	274	1343	11234	5384	2005	87	7138	16618
perceptron	3672	222	1159	8278	4408	1668	84	5646	12686

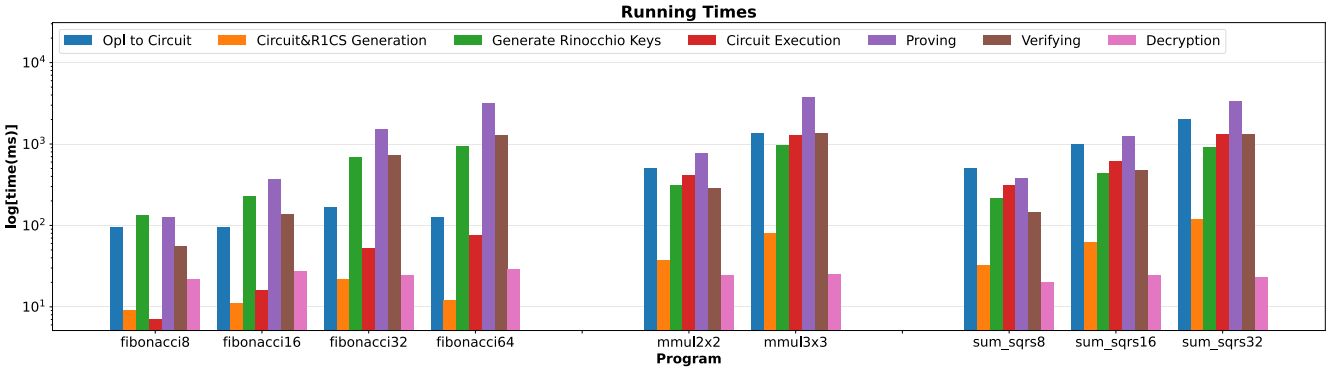
As for SEAL keys generation, the overhead is negligible and can be performed locally at the user side.

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 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.

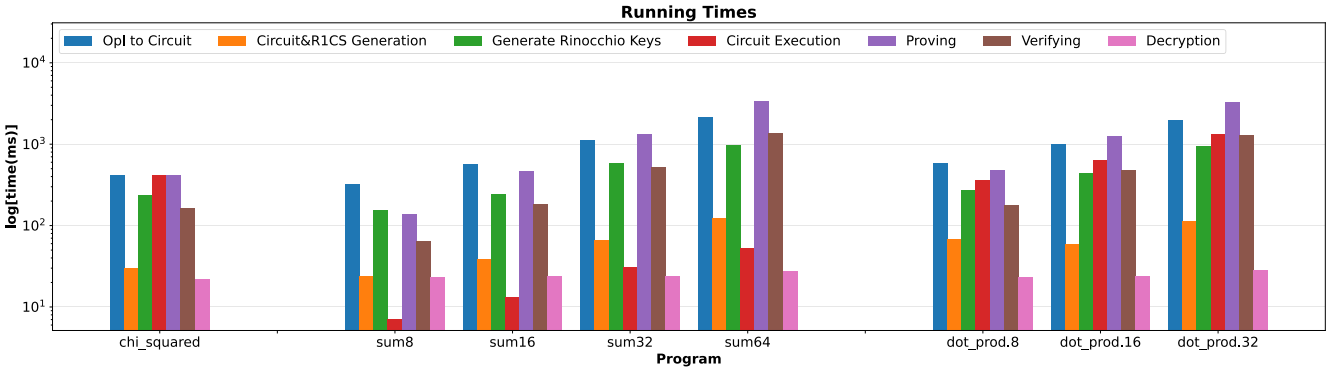
The proof size is 26,542,080 bits (approximately 3,318 KB) for all programs, since Rinocchio proofs are constant size by design. Conversely, the ciphertext size depends on the number of values in the circuit and the polynomial modulus degree. For instance, the Fibonacci program uses a polynomial size degree of 2^{14}, which, according to SEAL documentation, yields coefficients of size 438 bits. Therefore, for Fibonacci v64, the ciphertext size is $438 \times 64 = 28{,}032$ bits (approximately 3.5 KB).

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.

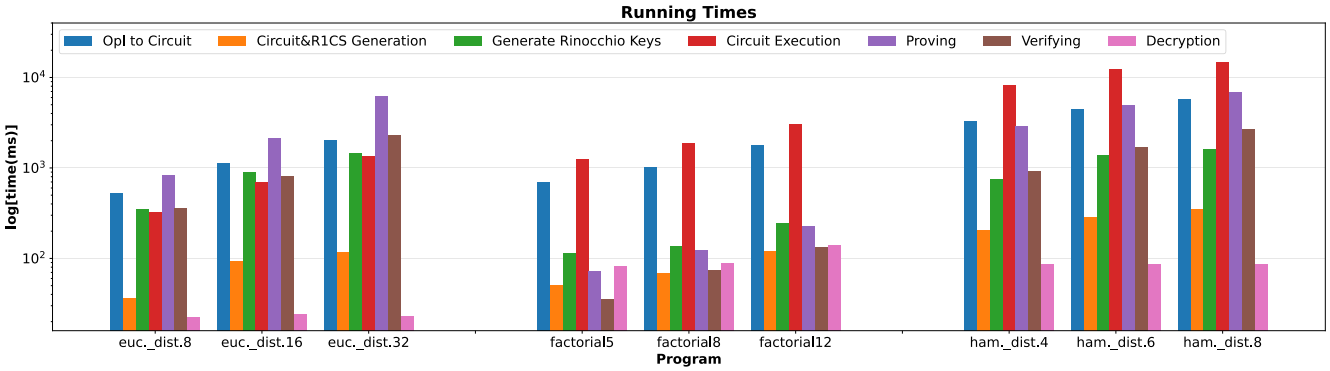
With respect to baseline computation costs, this is typically assessed without the additional privacy and integrity measures provided by homomorphic encryption and ZKP protocols. In such cases, evaluations are performed on unencrypted arithmetic circuits, and thus, computations are carried out in the 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 32 values requires 1,329,000 microseconds



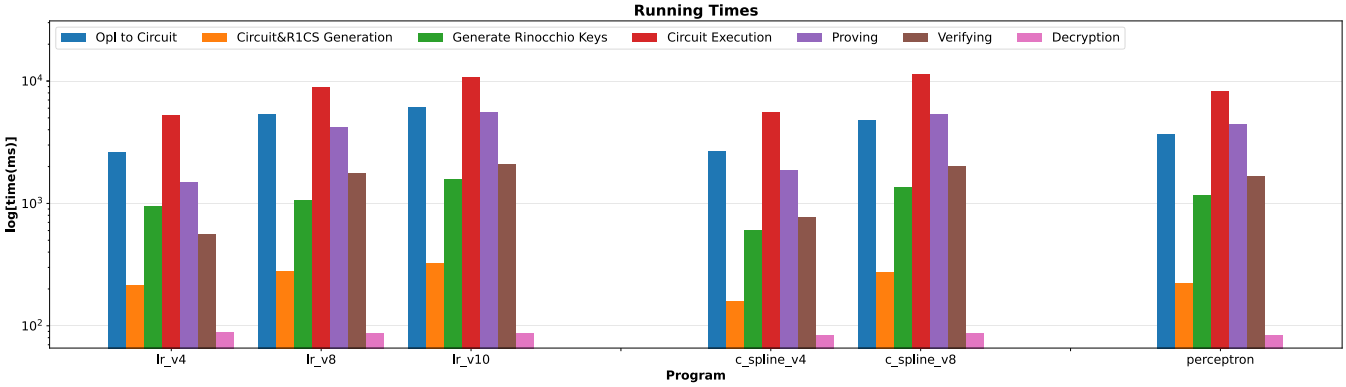
(a)



(b)



(c)



(d)

FIGURE 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.

TABLE 2. A comparison between our proposed system, PEEV, and the seven most recent similar works reveals 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	General-purpose	Leveled - BGV	Rinocchio	YES
HELM [30]	General-purpose	Fully - CGGI	N/A	NO
ArctyrEX [32]	General-purpose	Fully - CGGI	N/A	YES
GALA [73]	Neural networks	Leveled - BFV	N/A	NO
REDsec [24]	Neural networks	Fully - CGGI	N/A	NO
Zilch [53]	Verifiable computations	N/A	zk-STARK	YES
[35]	SVM training	Partial - Paillier Cryptosystem	Verification tags	NO
pvCNN [69]	CNN	Leveled HE	Groth16	NO

(Table 1). Conversely, running the same computation without encryption and integrity protections on the same machine requires 903 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 [30] 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 [32] is a general-purpose framework that tarhet a fully HE scheme and offers a high-level front-end, but also lacks a verification mechanism. GALA [73] and REDsec [24] 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 [53] 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 [35] 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 [69] is a specific-purpose scheme for convolutional 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.

VIII. PRACTICAL CONSIDERATIONS

The proposed methodology can readily scale to any problem size, however thresholds arise from both the SEAL HE backend as well as Rinocchio. Although SEAL can execute very deep circuits with enough noise budget to assure accurate results with very high probability, Rinocchio is more restrictive. The benchmarks introduced in the previous section exploit the maximum parameter set compatible with Rinocchio, which corresponds to a polynomial modulus degree of 2^{11}. Increasing this value further leads to incorrect results.

IX. 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 32 encrypted programs, and report low performance overheads both for encrypted computation and proof generation.

Some open challenges for future work include enabling PEEV's back-end to execute circuits of arbitrary size. One way to address this challenge is by splitting a larger circuit into smaller ones and aggregating the results. Another challenge involves accelerating the evaluation times, which could be addressed by parallelizing the computations performed by PEEV.

REFERENCES

[1] A. Bampoulidis, A. Bruni, L. Helminger, D. Kales, C. Rechberger, and R. Walch, "Privately connecting mobility to infectious diseases via applied cryptography," 2020, *arXiv:2005.02061*.

[2] M. Barbosa and P. Farshim, "Delegatable homomorphic encryption with applications to secure outsourcing of computation,"in *Proc. Cryptographers' Track RSA Conf.*, San Francisco, CA, USA. Cham, Switzerland: Springer, Feb. 2012, pp. 296–312.

[3] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable zero knowledge with no trusted setup,"' in *Proc. 39th Annu. Int. Cryptol. Conf. Adv. Cryptol. (CRYPTO)*, Santa Barbara, CA, USA. Cham, Switzerland: Springer, Aug. 2019, pp. 701–732.

[4] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge,"in *Proc. 33rd Annu. Cryptol. Conf. Adv. Cryptol. (CRYPTO)*, Santa Barbara, CA, USA. Cham, Switzerland: Springer, Aug. 2013, pp. 90–108.

[5] A. Benaissa, B. Retiat, B. Cebere, and A. E. Belfedhal, "TenSEAL: A library for encrypted tensor operations using homomorphic encryption," 2021, *arXiv:2104.03152*.

[6] M. Blum, P. Feldman, and S. Micali, "Non-interactive zero-knowledge and its applications," in *Proc. 20th Annu. ACM Symp. Theory Comput. (STOC)*, 1988, pp. 103–112.

[7] V. N. Boddeti, "Secure face matching using fully homomorphic encryption," in *Proc. IEEE 9th Int. Conf. Biometrics Theory, Appl. Syst. (BTAS)*, Oct. 2018, pp. 1–10.

[8] A. Bois, I. Cascudo, D. Fiore, and D. Kim, "Flexible and efficient verifiable computation on encrypted data," in *Proc. IACR Int. Conf. Public-Key Cryptogr.* Cham, Switzerland: Springer, 2021, pp. 528–558.

[9] C. Bonte, I. Iliashenko, J. Park, H. V. L. Pereira, and N. P. Smart, "FINAL: Faster FHE instantiated with NTRU and LWE," Cryptol. ePrint Arch., Santa Barbara, CA, USA, Tech. Rep. Paper 2022/074, 2022. [Online]. Available: https://eprint.iacr.org/2022/074

[10] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 1–36, Jul. 2014.

[11] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 315–334.

[12] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," in *Proc. 3rd Int. Workshop Secur. Cloud Comput.*, Apr. 2015, pp. 13–19.

[13] S. Chatel, C. Knabenhans, A. Pyrgelis, C. Troncoso, and J.-P. Hubaux, "Verifiable encodings for secure homomorphic analytics," 2022, *arXiv:2207.14071*.

[14] H. Chen, I. Chillotti, and Y. Song, "Improved bootstrapping for approximate homomorphic encryption," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.* Cham, Switzerland: Springer, 2019, pp. 34–54.

[15] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. 23rd Int. Conf. Theory Appl. Cryptol. Inf. Secur. Adv. Cryptol. (ASIACRYPT)*, Hong Kong. Cham, Switzerland: Springer, Dec. 2017, pp. 409–437.

[16] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds,"in *Proc. 22nd Int. Conf. Theory Appl. Cryptol. Inf. Secur. Adv. Cryptol. (ASIACRYPT)*, Hanoi, Vietnam. Cham, Switzerland: Springer, Dec. 2016, pp. 3–33.

[17] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, Jan. 2020.

[18] L. Coppolino, S. D'Antonio, V. Formicola, G. Mazzeo, and L. Romano, "VISE: Combining Intel SGX and homomorphic encryption for cloud industrial control systems," *IEEE Trans. Comput.*, vol. 70, no. 5, pp. 711–724, May 2021.

[19] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proc. 25th USENIX Secur. Symp. (USENIX Secur.)*, 2016, pp. 857–874.

[20] S. Dolev, S. Frenkel, and D. E. Tamir, "Error correction based on Hamming distance preserving in arithmetical and logical operations," in *Proc. IEEE 27th Conv. Electr. Electron. Eng. Isr.*, Nov. 2012, pp. 1–5.

[21] A. J. Duncan, S. Creese, and M. Goldsmith, "Insider attacks in cloud computing," in *Proc. IEEE 11th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Jun. 2012, pp. 857–862.

[22] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptol. ePrint Arch., Santa Barbara, CA, USA, Tech. Rep. 2012/144, 2012. [Online]. Available: https://eprint.iacr.org/2012/144

[23] D. Fiore, R. Gennaro, and V. Pastro, "Efficiently verifiable computation on encrypted data," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 844–855.

[24] L. Folkerts, C. Gouert, and N. G. Tsoutsos, "REDsec: Running encrypted discretized neural networks in seconds," Cryptol. ePrint Arch., 2021, Paper 2021/1100. [Online]. Available: https://eprint.iacr.org/2021/1100

[25] C. Ganesh, A. Nitulescu, and E. Soria-Vazquez, "Rinocchio: SNARKs for ring arithmetic," Cryptol. ePrint Arch., Santa Barbara, CA, USA, Tech. Rep. 2021/322, 2021. [Online]. Available: https://eprint.iacr.org/2021/322

[26] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, May 2009, pp. 169–178.

[27] C. Gentry, S. Halevi, and N. P. Smart, "Better bootstrapping in fully homomorphic encryption," in *Proc. Int. Workshop Public Key Cryptogr.* Cham, Switzerland: Springer, 2012, pp. 1–16.

[28] 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*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 203–225.

[29] S. Gorantala et al., "A general purpose transpiler for fully homomorphic encryption," Cryptol. ePrint Arch., Santa Barbara, CA, USA, Tech. Rep. 2021/811, 2021. [Online]. Available: https://eprint.iacr.org/2021/811

[30] C. Gouert, D. Mouris, and N. G. Tsoutsos, "HELM: Navigating homomorphic encryption through gates and lookup tables," Cryptol. ePrint Arch., Santa Barbara, CA, USA, Tech. Rep. 2023/1382, 2023. [Online]. Available: https://eprint.iacr.org/2023/1382

[31] C. Gouert, D. Mouris, and N. Tsoutsos, "SoK: New insights into fully homomorphic encryption libraries via standardized benchmarks," *Proc. Privacy Enhancing Technol.*, vol. 2023, no. 3, pp. 154–172, Jul. 2023.

[32] C. Gouert, V. Joseph, S. Dalton, C. Augonnet, M. Garland, and N. G. Tsoutsos, "ArctyrEX: Accelerated encrypted execution of general-purpose applications," 2023, *arXiv:2306.11006*.

[33] I. Grigoriadis, E. Vrochidou, I. Tsiatsiou, and G. A. Papakostas, "Machine learning as a service (MLaaS)—An enterprise perspective," in *Proc. Int. Conf. Data Sci. Appl. (ICDSA)*, vol. 2. Singapore: Springer, 2023, pp. 261–273.

[34] S. Halevi and V. Shoup, "Design and implementation of HElib: A homomorphic encryption library," Cryptol. ePrint Arch., Santa Barbara, CA, USA, Tech. Rep. 2020/1481, 2020.

[35] C. Hu, C. Zhang, D. Lei, T. Wu, X. Liu, and L. Zhu, "Achieving privacy-preserving and verifiable support vector machine training in the cloud," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 3476–3491, 2023.

[36] R. M. Jabir, S. I. R. Khanji, L. A. Ahmad, O. Alfandi, and H. Said, "Analysis of cloud computing attacks and countermeasures," in *Proc. 18th Int. Conf. Adv. Commun. Technol. (ICACT)*, Jan. 2016, pp. 117–123.

[37] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, no. 4, pp. 114–148, Aug. 2021.

[38] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 974–987.

[39] A. Konkin and S. Zapechnikov, "Zero knowledge proof and ZK-SNARK for private blockchains," *J. Comput. Virol. Hacking Techn.*, vol. 19, no. 3, pp. 443–449, Mar. 2023.

[40] J. Lai, R. H. Deng, H. Pang, and J. Weng, "Verifiable computation on outsourced encrypted data," in *Proc. 19th Eur. Symp. Res. Comput. Secur. Comput. Secur. (ESORICS)*, Wroclaw, Poland. Cham, Switzerland: Springer, Sep. 2014, pp. 273–291.

[41] Tune Insight. (Aug. 2022). *Lattigo v4*. [Online]. Available: https://github.com/tuneinsight/lattigo

[42] S. Li, X. Wang, and R. Xue, "Toward both privacy and efficiency of homomorphic MACs for polynomial functions and its applications," *Comput. J.*, vol. 65, no. 4, pp. 1020–1028, Apr. 2022.

[43] S. Li, X. Wang, and R. Zhang, "Privacy-preserving homomorphic MACs with efficient verification,"in *Proc. 25th Int. Conf., Held Services Conf. Federation Web Services (ICWS)*, Seattle, WA, USA. Cham, Switzerland: Springer, Jun. 2018, pp. 100–115.

[44] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 406–418.
[45] Z. Liu, L. Wan, J. Guo, F. Huang, X. Feng, L. Wang, and J. Ma, "PPRU: A privacy-preserving reputation updating scheme for cloud-assisted vehicular networks," *IEEE Trans. Veh. Technol.*, early access, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10349949
[46] Q. Lou, M. Santriaji, A. W. B. Yudha, J. Xue, and Y. Solihin, "VFHE: Verifiable fully homomorphic encryption with blind hash," 2023, *arXiv:2303.08886*.
[47] F. Luo et al., "Verifiable decryption for fully homomorphic encryption," in *Proc. Int. Conf. Inf. Secur.* Cham, Switzerland: Springer, 2018, pp. 347–365.
[48] V. Lyubashevsky, N. K. Nguyen, and M. Plançon, "Lattice-based zero-knowledge proofs and applications: Shorter, simpler, and more general," in *Proc. Annu. Int. Cryptol. Conf.* Cham, Switzerland: Springer, 2022, pp. 71–101.
[49] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Proc. 29th Annu. Int. Conf. Theory Appl. Cryptograph. Techn. Adv. Cryptol. (EUROCRYPT)*, French Riviera, France. Berlin, Germany: Springer, Jun. 2010, pp. 1–23.
[50] G. K. Mahato and S. K. Chakraborty, "A comparative review on homomorphic encryption for cloud security," *IETE J. Res.*, vol. 69, no. 8, pp. 5124–5133, Sep. 2023.
[51] M. Masdari and M. Jalali, "A survey and taxonomy of DoS attacks in cloud computing," *Secur. Commun. Netw.*, vol. 9, no. 16, pp. 3724–3751, Nov. 2016.
[52] Y. Miao, Y. Yang, X. Li, Z. Liu, H. Li, K. R. Choo, and R. H. Deng, "Efficient privacy-preserving spatial range query over outsourced encrypted data," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 3921–3933, 2023.
[53] D. Mouris and N. G. Tsoutsos, "Zilch: A framework for deploying transparent zero-knowledge proofs," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 3269–3284, 2021.
[54] D. Mouris, N. G. Tsoutsos, and M. Maniatakos, "TERMinator suite: Benchmarking privacy-preserving architectures," *IEEE Comput. Archit. Lett.*, vol. 17, no. 2, pp. 122–125, Jul. 2018.
[55] D. Natarajan, A. Loveless, W. Dai, and R. Dreslinski, "CHEX-MIX: Combining homomorphic encryption with trusted execution environments for two-party oblivious inference in the cloud," Cryptol. ePrint Arch., Santa Barbara, CA, USA, Tech. Rep. 2021/1603, 2021.
[56] A. Ozdemir, F. Brown, and R. S. Wahby, "CirC: Compiler infrastructure for proof systems, software verification, and more," Cryptol. ePrint Arch., Santa Barbara, CA, USA, Tech. Rep. 2020/1586, 2020. [Online]. Available: https://eprint.iacr.org/2020/1586
[57] H. Pallathadka, G. S. Sajja, K. Phasinam, M. Ritonga, M. Naved, R. Bansal, and J. Quiñonez-Choquecota, "An investigation of various applications and related challenges in cloud computing," *Mater. Today, Proc.*, vol. 51, pp. 2245–2248, Jan. 2022.
[58] P. Papadopoulos, N. Kourtellis, P. R. Rodriguez, and N. Laoutaris, "If you are not paying for it, you are the product: How much do advertisers pay to reach you?" in *Proc. Internet Meas. Conf.*, Nov. 2017, pp. 142–156.
[59] F. J. G. Peñalvo, A. Sharma, A. Chhabra, S. K. Singh, S. Kumar, V. Arya, and A. Gaurav, "Mobile cloud computing and sustainable development: Opportunities, challenges, and future directions," *Int. J. Cloud Appl. Comput.*, vol. 12, no. 1, pp. 1–20, Oct. 2022.
[60] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *J. ACM*, vol. 56, no. 6, pp. 1–40, Sep. 2009.
[61] U. R. Saxena and T. Alam, "Role-based access using partial homomorphic encryption for securing cloud data," *Int. J. Syst. Assurance Eng. Manage.*, vol. 14, no. 3, pp. 950–966, Jun. 2023.
[62] *Microsoft SEAL (release 4.1)*, Microsoft Research, Redmond, WA, USA, Jan. 2023. [Online]. Available: https://github.com/Microsoft/SEAL
[63] X. Sun, F. R. Yu, P. Zhang, Z. Sun, W. Xie, and X. Peng, "A survey on zero-knowledge proof in blockchain," *IEEE Netw.*, vol. 35, no. 4, pp. 198–205, Jul. 2021.
[64] N. Toumi, M. Bagaa, and A. Ksentini, "Machine learning for service migration: A survey," *IEEE Commun. Surveys Tuts.*, vol. 25, no. 3, pp. 1991–2020, 3rd Quart., 2023.
[65] A. Viand, C. Knabenhans, and A. Hithnawi, "Verifiable fully homomorphic encryption," 2023, *arXiv:2301.07041*.
[66] R. Vijay, P. Mahajan, and R. Kandwal, "Hamming distance based clustering algorithm," *Int. J. Inf. Retr. Res.*, vol. 2, no. 1, pp. 11–20, Jan. 2012.
[67] S. Vinoth, H. L. Vemula, B. Haralayya, P. Mamgain, M. F. Hasan, and M. Naved, "Application of cloud computing in banking and e-commerce and related security threats," *Mater. Today, Proc.*, vol. 51, pp. 2172–2175, Jan. 2022.
[68] W. Wang, Y. Jiang, Q. Shen, W. Huang, H. Chen, S. Wang, X. Wang, H. Tang, K. Chen, K. Lauter, and D. Lin, "Toward scalable fully homomorphic encryption through light trusted computing assistance," 2019, *arXiv:1905.07766*.
[69] J. Weng, J. Weng, G. Tang, A. Yang, M. Li, and J.-N. Liu, "PvCNN: Privacy-preserving and verifiable convolutional neural network testing," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 2218–2233, 2023.
[70] W. Yin, "Zero-knowledge proof intelligent recommendation system to protect students' data privacy in the digital age," *Appl. Artif. Intell.*, vol. 37, no. 1, Dec. 2023, Art. no. 2222495.
[71] Zama. (2022). *Concrete: TFHE Compiler That Converts Python Programs Into FHE Equivalent*. [Online]. Available: https://github.com/zama-ai/concrete
[72] Zama. (2022). *TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data*. [Online]. Available: https://github.com/zama-ai/tfhe-rs
[73] Q. Zhang, C. Xin, and H. Wu, "GALA: Greedy computAtion for Linear Algebra in privacy-preserved neural networks," 2021, *arXiv:2105.01827*.

OMAR AHMED received the B.Sc. degree in computer science from Luxor University, Luxor, Egypt, in 2020. He is currently pursuing the M.Sc. degree in cybersecurity with the Electrical and Computer Engineering Department, University of Delaware, Newark, DE, USA. He is also a Fulbright Scholarship Grantee. His research interest includes cybersecurity, with a focus on applied cryptography and steganography.

CHARLES GOUERT (Student Member, IEEE) received the B.Sc. degree in electrical and computer engineering from the University of Delaware, Newark, DE, USA, in 2018, where he is currently pursuing the Ph.D. degree in computer engineering with the Electrical and Computer Engineering Department. He is also the global challenge Co-Lead of the international Embedded Security Challenge (ESC) competition that is held annually during the Cyber Security Awareness Worldwide (CSAW) Event. His current research interests include applied cryptography and private outsourcing.

NEKTARIOS GEORGIOS TSOUTSOS (Member, IEEE) received the M.Sc. degree in computer engineering from Columbia University and the Ph.D. degree in computer science from New York University. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Delaware, and the Department of Computer and Information Sciences. He has authored multiple articles in IEEE TRANSACTIONS and conference proceedings. He serves on the program committee for several international conferences. He is also a Faculty Organizer of the international Embedded Security Challenge (ESC) competition that is held annually during the Cyber Security Awareness Worldwide (CSAW) Event. His research interests include cybersecurity and applied cryptography, with a special focus on hardware security, trustworthy computing, and privacy outsourcing, and holds a patent on encrypted computation using homomorphic encryption.