PEEV: Parse Encrypt Execute Verify - A Verifiable FHE Framework

Omar Ahmed
oaaa@udel.edu

Charles Gouert
cgouert@udel.edu

Nektarios Georgios Tsoutsos
tsoutsos@udel.edu

***Abstract*—Cloud computing has been a prominent technology that allows users to store their data and outsource intensive computations. However, cloud users 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 fails to allow encrypted data processing. 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, 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. Our framework relies on homomorphic encryption techniques as well as zero-knowledge proofs to achieve verifiable privacy-preserving computation. Our approach enables low performance overheads to support practical deployments, and enables developers to express their encrypted programs in a high-level language, while abstracting all complexities related to encryption and verification.**

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 [58, 49, 51]. 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 [50]. In addition, cloud computing is susceptible to a variety of cyberattacks including network attacks and account hijacking [19, 31, 44].

While numerous research efforts have been proposed to counter cloud attacks [18, 39, 33], deploying them 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 them to remote cloud servers. However, this method is only suitable for protecting static data, which limits usability and prevents the server from performing any meaningful computation on the outsourced data. But what if end users need to process their data after being uploaded to the cloud and also preserve their privacy? In this case, traditional cryptography cannot help.

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

$$z = F(x, y) = x + y.$$

Here we assume 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', such that:

$$z' = F'(x', y') = x' + y', \quad \text{and} \quad Decrypt(z') = z = x + y.$$

Essentially, FHE ensures that

$$F(plaintext) = Decrypt(F'(ciphertext))$$

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 [54, 5, 36], 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 [61, 11, 26, 27, 46, 28].

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 [29, 56]. 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 [25]. ZKPs have gained much attention and improved over the years due to their importance in verifiable computation [34, 60, 41, 45].

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.

In this work, we combine 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, in our approach, 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 circuit is a blueprint that defines the encrypted inputs and the computation to be executed on them. 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 don't 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 verifiing the execution proof, and decrypting the result. Overall, our contributions can be summarized as follows:

1) We introduce 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 new Operations List (OpL) intermediate representation 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 provides a background on homomorphic encryption schemes and libraries, zero-knowledge proofs, and compilers for HE and ZKPs. Section III introduces our proposed approach for achieving verifiable privacy-preserving computation on encryption data, while Section IV describes the implementation details of our framework. Section V presents our experimental results over representative benchmarks, while Section VI discusses prior works addressing the problem of verifiable computation. Finally, our concluding remarks are presented in Section VII.

II. 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 [53]. Examples of PHE include the unpadded RSA, ElGamal, and Paillier cryptosystems [43].
- **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) [52] and ring learning with errors (RLWE) [42] 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) [9], the Brakerski/Fan-Vercauteren (BFV) [20], and the Cheon-Kim-Kim-Song (CKKS) [14] 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 [23]. Nevertheless, bootstrapping is a very costly technique, being over an order of magnitude slower than other HE operations. Even with proposed optimizations [24, 13, 32], 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, we focus on LHE in this work.

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 [15, 16]. 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 [62].
- **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 [8].
- **HElib**. HElib implements the BGV and CKKS schemes [30]. 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 [36].
- **SEAL**. Microsoft released its own HE library called SEAL [54]. 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 [55]. They were first introduced in 1985 [25] 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), we can conclude 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 them 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 [10].

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 [48]. 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 [28]. 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 [46]. 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 them.
- **HELM** is a privacy-preserving framework for processing data in the encrypted domain with FHE [27]. 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 [61]. 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.

III. 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, 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.

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. An 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 handwritten encrypted implementation.
- Optimizing the code or making future modifications can be done rapidly and in a straightforward manner.
- Eliminating the complexity of tracking each operation's input and output and the creation of the R1CS needed for verifiability.
- The user does not need to manually initialize any HE or ZKP parameters.

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

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 them 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 and ZKP libraries. For instance, our framework 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 his 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 we are using a zk-SNARK proof system, the key generation algorithm requires a trusted setup environment, which could be acquired through a trusted third party.

IV. 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

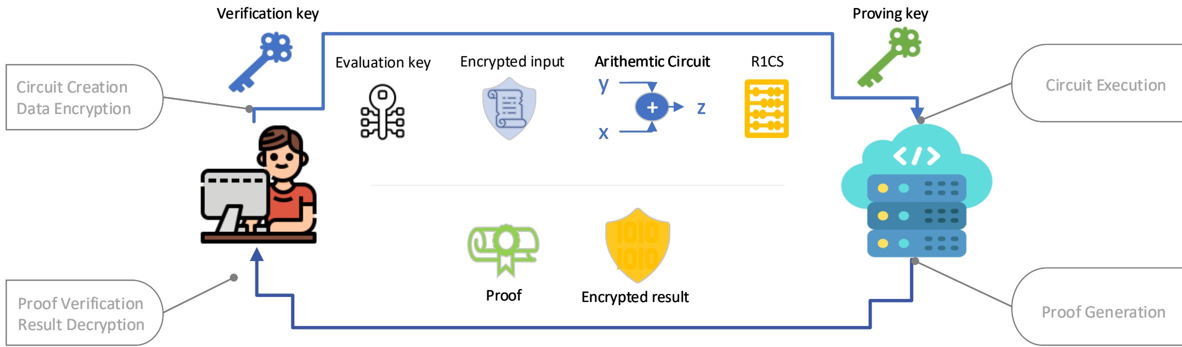


Fig. 1: **Client-Server Communication:** The client creates the arithmetic circuit, R1CS, and evaluation key, encrypts the input, and sends them to the server. The server runs the computation, generates the proof, and sends the result and the proof back to the client. The client checks if the proof is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

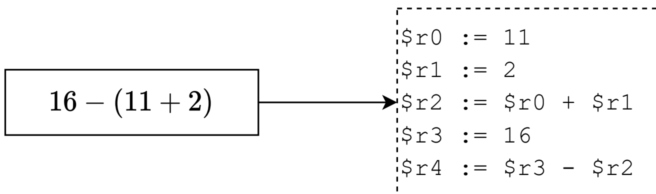


Fig. 2: YAP flattens the equation $16-(11+2)$ into a sequence of operations. The OpL lists every single operation in a single line, and subsequent lines are dependent on previous lines.

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 them in other libraries such as TFHE, which operates on bits.

For enabling verifiable computation, we use Rinocchio as our back-end ZKP system [22]. 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 [57].

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 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.

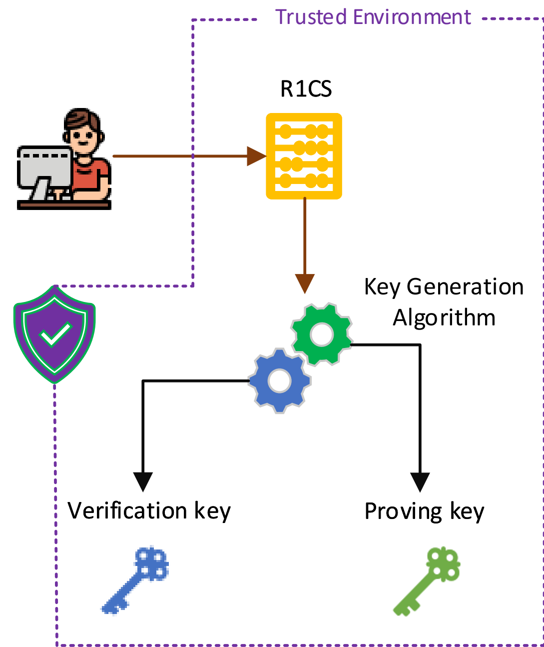


Fig. 3: The cloud user inputs the R1CS to the ZKP key generation algorithm. The key generation process requires a trusted environment, which can be achieved through a third-party who offers a trusted setup. The algorithm outputs two keys: one for the cloud to generate the proof, and the other for the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

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 such as relinearization, and providing the front-end for executing the operations on encrypted data. We remark that *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 [20]. 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.

We employ the BGV scheme in our system to perform leveled HE. 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 our 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). We use such a large polynomial modulus degree for SEAL to allow more complicated encrypted computations.

V. 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. We run these tests on a Windows laptop with a 6th generation Intel i5 processor at 2.30 GHz and 16 GB of RAM.

We remark that 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.

Table I summarizes the execution times of PEEV evaluated across 26 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*

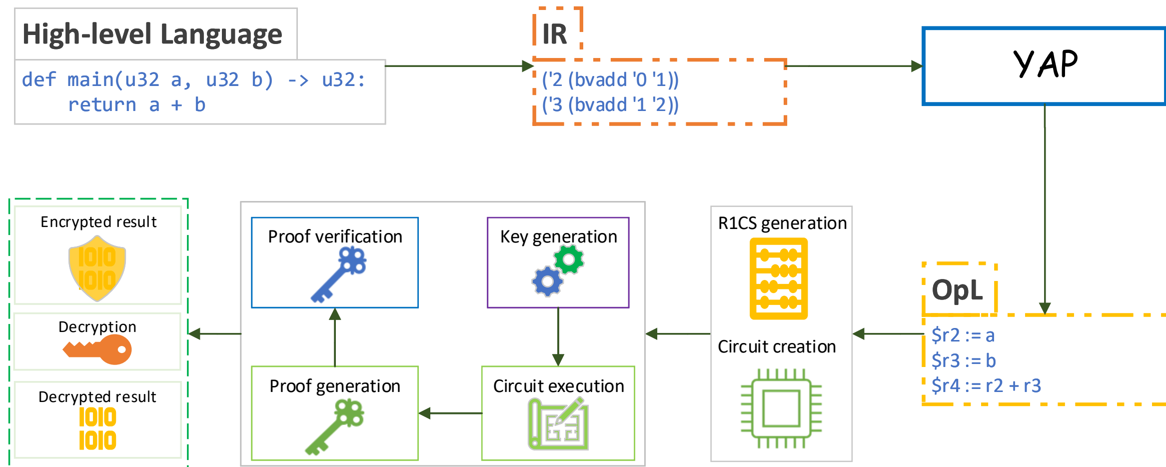


Fig. 4: Workflow of our verifiable privacy-preserving computation model. Starting with ZoKrates code, the program is converted to CirC's IR, before YAP transforms the IR to OpL. Then, PEEV uses the OpL to generate the arithmetic circuit and the R1CS. After executing the circuit, a proof is generated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

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).

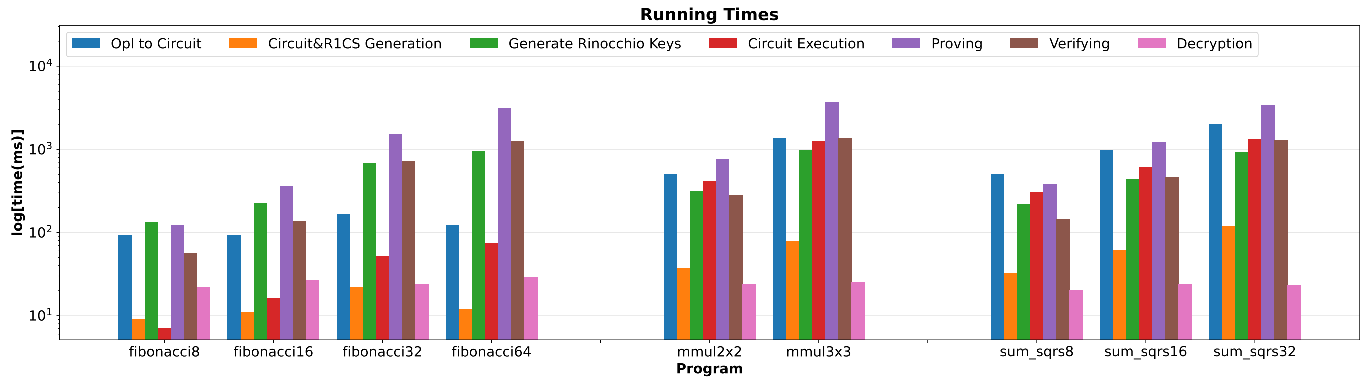
In Figure 5, we visualize running times (ms) on a logarithmic scale of each benchmark. The most expensive component of the entire process is the proof generation; for Fibonacci v64, 3×3 matrix multiplication, sum v64, vector dot product v32, and factorial v12, proof generation takes more than 3 seconds. 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 our 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.

VI. Related Work

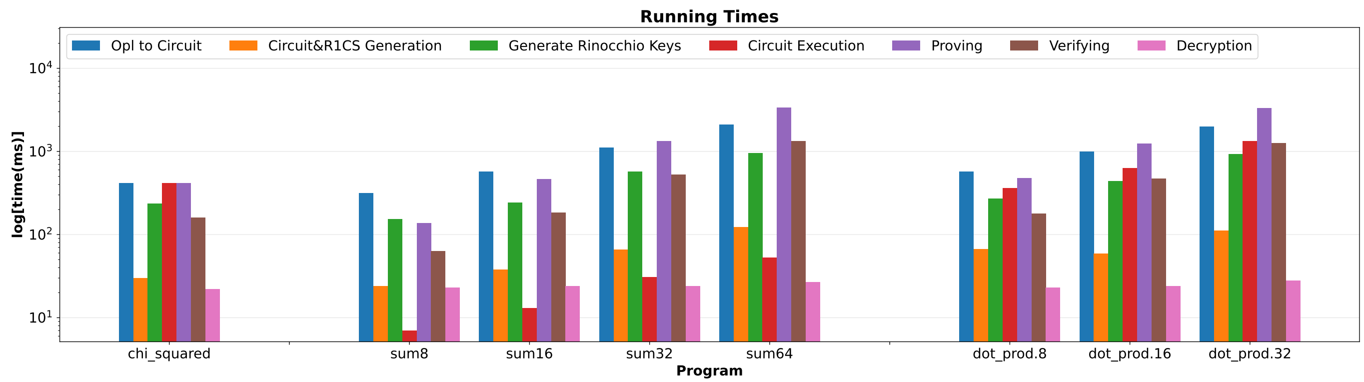
In 2012, the authors of [2] introduced a cryptographic primitive called delegatable homomorphic encryption (DHE) that allows one party to delegate the computation of a circuit with encrypted data to an untrusted party. This work was somewhat limited, as it could only handle functions that took one encrypted 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 related cryptographic primitive is a homomorphic encrypted authenticator (HEA) [35] 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 [37, 1]. On the contrary, our framework can be practically adopted thanks to the fast performance of our primitives.

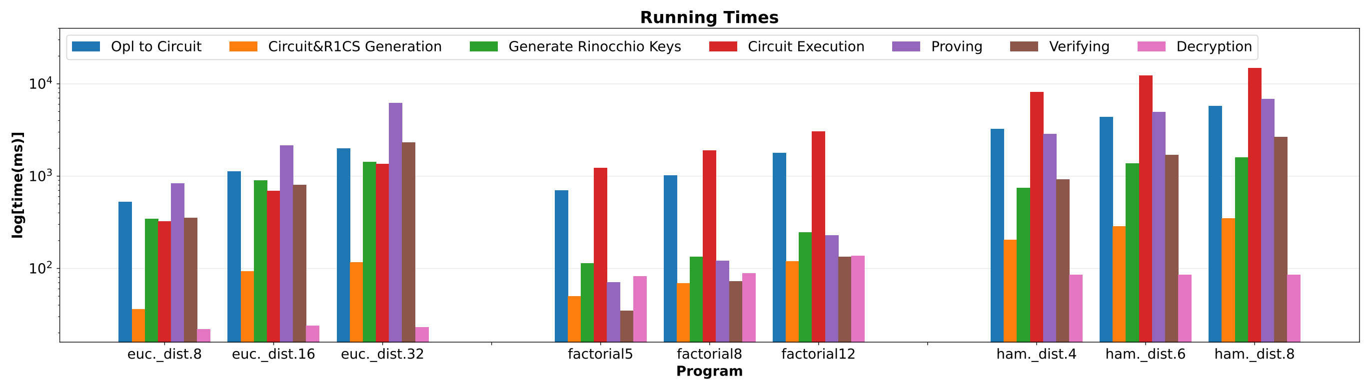
In 2014, Fiore *et al.* proposed an efficient construction for verifiable computation (VC) that enabled authenticating computations on encrypted data [21]. Its efficiency came from a homomorphic hashing technique, which could verify the computations on ciphertext data at the same cost as plaintext data. Nevertheless, the initial generic construction introduced efficiency issues when the FHE ciphertext space does not match the message space supported by the VC scheme. These challenges were later mitigated in derivative works [7].



(a)



(b)



(b)

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

Program	OpL to Circuit	Circuit&R1CS Generation	Rinocchio Keys Generation	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

TABLE I: Execution times of PEEV across different sets of programs that include mathematical operations such as additions, subtractions, and multiplications (all time 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.

In particular, the authors of [7] 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 our PEEV framework offers support for non-interactive proofs.

In 2018, Luo *et al.* proposed a methodology for ensuring the decryption correctness for BGV ciphertexts [40]. 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.

Finally, recent works for providing integrity with homomorphic encryption include verifying FHE computations by utilizing trusted execution environments (TEEs) [17, 47, 59], as well as verifying the integrity of a computation based on MACs [38, 12]. Nevertheless, these approaches rely on different approaches than our PEEV framework, which enables both integrity and confidentiality using FHE and ZKPs.

VII. Conclusion

In this paper we introduce 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 bespoke YAP parser that enables translation from a high-level language into our 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 26 encrypted programs, and report low performance overheads both for encrypted computation and proof generation.

REFERENCES

[1] Alexandros Bampoulidis et al. "Privately connecting mobility to infectious diseases via applied cryptography". In: *arXiv preprint arXiv:2005.02061* (2020).

[2] Manuel Barbosa and Pooya Farshim. "Delegatable homomorphic encryption with applications to secure outsourcing of computation". In: *Topics in Cryptology–CT-RSA 2012: The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27–March 2, 2012. Proceedings*. Springer. 2012, pp. 296–312.

[3] Eli Ben-Sasson et al. "Scalable zero knowledge with no trusted setup". In: *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer. 2019, pp. 701–732.

[4] Eli Ben-Sasson et al. "SNARKs for C: Verifying program executions succinctly and in zero knowledge". In: *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*. Springer. 2013, pp. 90–108.

[5] Ayoub Benaissa et al. *TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption*. 2021. arXiv: 2104.03152 [cs.CR].

[6] Manuel Blum, Paul Feldman, and Silvio Micali. "Non-interactive zero-knowledge and its applications". In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. 1988, pp. 103–112.

[7] Alexandre Bois et al. "Flexible and efficient verifiable computation on encrypted data". In: *IACR International Conference on Public-Key Cryptography*. Springer. 2021, pp. 528–558.

[8] Charlotte Bonte et al. *FINAL: Faster FHE instantiated with NTRU and LWE*. Cryptology ePrint Archive, Paper 2022/074. https://eprint.iacr.org/2022/074. 2022.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ACM Transactions on Computation Theory (TOCT)* 6.3 (2014), pp. 1–36.

[10] Benedikt Bünz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020.

[11] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. "Armadillo: a compilation chain for privacy preserving applications". In: *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. 2015, pp. 13–19.

[12] Sylvain Chatel et al. "Verifiable encodings for secure homomorphic analytics". In: *arXiv preprint arXiv:2207.14071* (2022).

[13] Hao Chen, Ilaria Chillotti, and Yongsoo Song. "Improved bootstrapping for approximate homomorphic encryption". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2019, pp. 34–54.

[14] Jung Hee Cheon et al. "Homomorphic encryption for arithmetic of approximate numbers". In: *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer. 2017, pp. 409–437.

[15] Ilaria Chillotti et al. "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds". In: *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*. Springer. 2016, pp. 3–33.

[16] Ilaria Chillotti et al. "TFHE: fast fully homomorphic encryption over the torus". In: *Journal of Cryptology* 33.1 (2020), pp. 34–91.

[17] Luigi Coppolino et al. "Vise: Combining intel sgx and homomorphic encryption for cloud industrial control systems". In: *IEEE Transactions on Computers* 70.5 (2020), pp. 711–724.

[18] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal hardware extensions for strong software isolation". In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 857–874.

[19] Adrian J Duncan, Sadie Creese, and Michael Goldsmith. "Insider attacks in cloud computing". In: *2012 IEEE 11th international conference on trust, security and privacy in computing and communications*. IEEE. 2012, pp. 857–862.

[20] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144. 2012.

[21] Dario Fiore, Rosario Gennaro, and Valerio Pastro. "Efficiently verifiable computation on encrypted data". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 844–855.

[22] Chaya Ganesh, Anca Nitulescu, and Eduardo Soria-Vazquez. *Rinocchio: SNARKs for Ring Arithmetic*. Cryptology ePrint Archive, Paper 2021/322. https://eprint.iacr.org/2021/322. 2021.

[23] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 2009, pp. 169–178.

[24] Craig Gentry, Shai Halevi, and Nigel P Smart. "Better bootstrapping in fully homomorphic encryption". In: *International Workshop on Public Key Cryptography*. Springer. 2012, pp. 1–16.

[25] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. "The knowledge complexity of interactive proof-systems". In: *Providing sound foundations for cryp-*

tography: On the work of shafi goldwasser and silvio micali. 2019, pp. 203–225.

[26] Shruthi Gorantala et al. *A General Purpose Transpiler for Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2021/811. https://eprint.iacr.org/2021/811. 2021.

[27] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. *HELM: Navigating Homomorphic Encryption through Gates and Lookup Tables*. Cryptology ePrint Archive, Paper 2023/1382. https://eprint.iacr.org/2023/1382. 2023.

[28] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. "SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks". In: *Proceedings on Privacy Enhancing Technologies* 2023.3 (July 2023), pp. 154–172. DOI: 10.56553/popets-2023-0075.

[29] Ioannis Grigoriadis et al. "Machine Learning as a Service (MLaaS)—An Enterprise Perspective". In: *Proceedings of International Conference on Data Science and Applications: ICDSA 2022, Volume 2*. Springer. 2023, pp. 261–273.

[30] Shai Halevi and Victor Shoup. "Design and implementation of HElib: a homomorphic encryption library". In: *Cryptology ePrint Archive* (2020).

[31] Raja Mohamed Jabir et al. "Analysis of cloud computing attacks and countermeasures". In: *2016 18th international conference on advanced communication technology (ICACT)*. IEEE. 2016, pp. 117–123.

[32] Wonkyung Jung et al. "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), pp. 114–148.

[33] Vladimir Kiriansky et al. "DAWG: A defense against cache timing attacks in speculative execution processors". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 974–987.

[34] Anatoly Konkin and Sergey Zapechnikov. "Zero knowledge proof and ZK-SNARK for private blockchains". In: *Journal of Computer Virology and Hacking Techniques* (2023), pp. 1–7.

[35] Junzuo Lai et al. "Verifiable computation on outsourced encrypted data". In: *Computer Security-ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I 19*. Springer. 2014, pp. 273–291.

[36] *Lattigo v4*. Online: https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA. Aug. 2022.

[37] Shimin Li, Xin Wang, and Rui Xue. "Toward Both Privacy and Efficiency of Homomorphic MACs for Polynomial Functions and Its Applications". In: *The Computer Journal* 65.4 (2022), pp. 1020–1028.

[38] Shimin Li, Xin Wang, and Rui Zhang. "Privacy-Preserving Homomorphic MACs with Efficient Verification". In: *Web Services–ICWS 2018: 25th International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings 16*. Springer. 2018, pp. 100–115.

[39] Fangfei Liu et al. "Catalyst: Defeating last-level cache side channel attacks in cloud computing". In: *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE. 2016, pp. 406–418.

[40] Fucai Luo and Kunpeng Wang. "Verifiable decryption for fully homomorphic encryption". In: *International Conference on Information Security*. Springer. 2018, pp. 347–365.

[41] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. "Lattice-based zero-knowledge proofs and applications: shorter, simpler, and more general". In: *Annual International Cryptology Conference*. Springer. 2022, pp. 71–101.

[42] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings". In: *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*. Springer. 2010, pp. 1–23.

[43] Ganesh Kumar Mahato and Swarnendu Kumar Chakraborty. "A comparative review on homomorphic encryption for cloud security". In: *IETE Journal of Research* 69.8 (2023), pp. 5124–5133.

[44] Mohammad Masdari and Marzie Jalali. "A survey and taxonomy of DoS attacks in cloud computing". In: *Security and Communication Networks* 9.16 (2016), pp. 3724–3751.

[45] Dimitris Mouris and Nektarios Georgios Tsoutsos. "Zilch: A framework for deploying transparent zero-knowledge proofs". In: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 3269–3284.

[46] Dimitris Mouris, Nektarios Georgios Tsoutsos, and Michail Maniatakos. "TERMinator Suite: Benchmarking Privacy-Preserving Architectures". In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 122–125. DOI: 10.1109/LCA.2018.281281.

[47] Deepika Natarajan et al. "Chex-mix: Combining homomorphic encryption with trusted execution environments for two-party oblivious inference in the cloud". In: *Cryptology ePrint Archive* (2021).

[48] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. *CirC: Compiler infrastructure for proof systems, software verification, and more*. Cryptology ePrint Archive, Paper 2020/1586. https://eprint.iacr.org/2020/1586. 2020.

[49] Harikumar Pallathadka et al. "An investigation of various applications and related challenges in cloud computing". In: *Materials Today: Proceedings* 51 (2022), pp. 2245–2248.

[50] Panagiotis Papadopoulos et al. "If you are not paying for it, you are the product: How much do advertisers

pay to reach you?" In: *Proceedings of the 2017 Internet Measurement Conference*. 2017, pp. 142–156.
[51] Francisco José García Peñalvo et al. "Mobile cloud computing and sustainable development: Opportunities, challenges, and future directions". In: *International Journal of Cloud Applications and Computing (IJCAC)* 12.1 (2022), pp. 1–20.
[52] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40.
[53] Urvashi Rahul Saxena and Taj Alam. "Role-based access using partial homomorphic encryption for securing cloud data". In: *International Journal of System Assurance Engineering and Management* 14.3 (2023), pp. 950–966.
[54] *Microsoft SEAL (release 4.1)*. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. Jan. 2023.
[55] Xiaoqiang Sun et al. "A survey on zero-knowledge proof in blockchain". In: *IEEE network* 35.4 (2021), pp. 198–205.
[56] Nassima Toumi, Miloud Bagaa, and Adlen Ksentini. "Machine Learning for Service Migration: A Survey". In: *IEEE Communications Surveys & Tutorials* (2023).
[57] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. "Verifiable fully homomorphic encryption". In: *arXiv preprint arXiv:2301.07041* (2023).
[58] S Vinoth et al. "Application of cloud computing in banking and e-commerce and related security threats". In: *Materials Today: Proceedings* 51 (2022), pp. 2172–2175.
[59] Wenhao Wang et al. "Toward scalable fully homomorphic encryption through light trusted computing assistance". In: *arXiv preprint arXiv:1905.07766* (2019).
[60] Wenjing Yin. "Zero-Knowledge Proof Intelligent Recommendation System to Protect Students' Data Privacy in the Digital Age". In: *Applied Artificial Intelligence* 37.1 (2023), p. 2222495.
[61] Zama. *Concrete: TFHE Compiler that converts python programs into FHE equivalent*. https://github.com/zama-ai/concrete. 2022.
[62] Zama. *TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data*. https://github.com/zama-ai/tfhe-rs. 2022.