# Proteus: A Fully Homomorphic Authenticated Transciphering Protocol

Lars Wolfgang Folkerts and Nektarios Georgios Tsoutsos

University of Delaware, Newark, DE 19716, USA
{folkerts,tsoutsos}@udel.edu

**Abstract.** Fully Homomorphic Encryption (FHE) is a powerful technology that allows a cloud server to perform computations directly on ciphertexts. To overcome the overhead of sending and storing large FHE ciphertexts, the concept of FHE transciphering was introduced, allowing symmetric key encrypted ciphertexts to be transformed into FHE ciphertexts by deploying symmetric key decryption homomorphically. However, existing FHE transciphering schemes remain unauthenticated and malleable, allowing attackers to manipulate data and remain undetected. This work introduces Proteus, a new methodology for authenticated transciphering, which enables oblivious access control, preventing users from downloading unauthenticated or malicious data. Our protocol implementation adopts ASCON, NIST's new standard for lightweight cryptography, to enable homomorphic hashing and authenticated transciphering. Our ASCON transcipher is paired with the TFHE encryption scheme, which is well suited to perform encrypted rotation and bitwise operations. We evaluate our approach with a variety of real-life privacy-preserving applications, including URL phishing detection, private content moderation of hate speech, and biometric authentication.

**Keywords:** Fully Homomorphic Encryption · Transciphering · Hybrid Homomorphic Encryption.

## 1 Introduction

In the digital smartphone world, users are required to trust cloud entities with personally identifiable information such as biometric data, financial information, and government identification numbers. Even conversations between individuals require trust that the host service is not compromised. This trust, however, is often violated by data breaches [21,4]. Likewise, recent examples of government infiltration highlights vulnerabilities in private communication, including the Encrochat messaging app [27], PhantomSecure phones [11] and Anom phones [12]. After such devastating breaches, technology companies are pressured to adopt new privacy solutions and protocols to protect user data and communications. For example, Meta recently announced that they will roll out end-to-end encryption as default on their messenger application in 2024 [13]. Their publicly released whitepapers detail a new Messenger protocol that prevents Meta from

accessing the messaging data, even in the case of a compromised server or a government subpoena [22,23].

While these are welcome improvements, existing protocols are only designed to protect *data at rest* and do not readily support applications that require the cloud server to perform computations on user data. In addition, existing private data backup methods cannot prevent bad actors from storing and disseminating illegal or malicious data via the service (e.g., copyrighted materials). By relying on symmetric key encryption alone, the only way to support these applications would be to allow the cloud server access to the decryption key, *contradicting the privacy requirements of these schemes.*

A promising solution to these challenges is Fully Homomorphic Encryption (FHE), which allows remote servers to perform outsourced computations on encrypted data. By using FHE ciphertexts instead of symmetric key encryption, a server can perform the aforementioned content moderation, as well as biometric authentication algorithms, without learning any information about the user. Several powerful FHE-based frameworks have already been developed, including convolutional neural network image classification [16,34] and sentiment analysis classification [25].

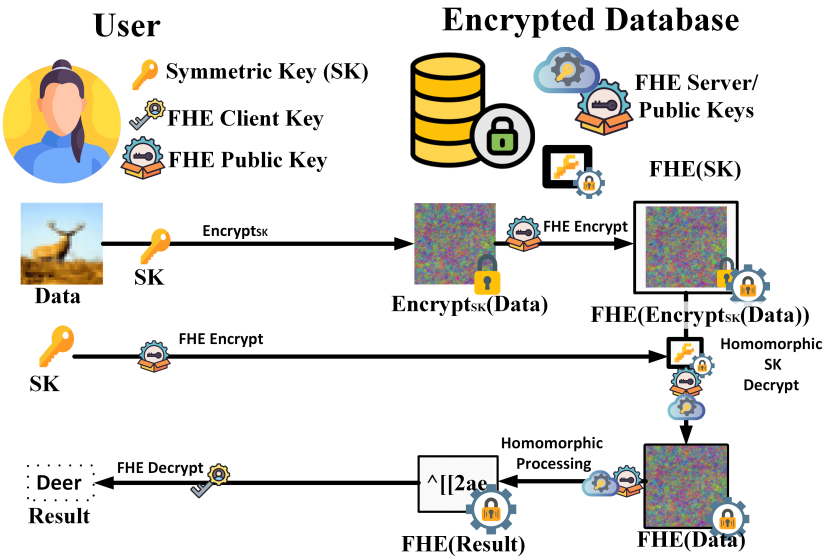


Fig. 1: **Basic Transciphering:** Compressed TFHE ciphertexts are about 400 times larger than their plaintext counterparts, making storage less practical. Transciphering allows a reduced communication overhead by transmitting of symmetric ciphertexts, and converting them to FHE ciphertexts on the server for further processing. Our work (see Figure 6) proposes a new protocol, Proteus, intended for long-term data storage and oblivious data verification.

One optimization that has recently garnered attention is FHE transciphering [10,2,19], also known as hybrid homomorphic encryption, which focuses on reduc-

ing communication overheads caused by the substantial bitsize of FHE ciphertexts. FHE ciphertexts can grow by two or three orders of magnitude compared to their plaintext counterparts. This ciphertext inflation creates excess communication overheads when sending FHE data to a remote server [10,2]. FHE transciphering addresses this problem by allowing a server to transform symmetric ciphertexts into FHE ciphertexts by performing *symmetric key decryption homomorphically*. In the basic blueprint, illustrated in Figure 1, a client encrypts their plaintext data with a symmetric key cipher and sends the encrypted data to the server. The client also homomorphically encrypts the symmetric key and sends it to the server, along with the FHE public key; the server uses this FHE public key to apply an outer FHE-encryption layer to the symmetric ciphertexts, before homomorphically decrypting the ciphertext with the FHE-encrypted symmetric key of the user. In effect, this step removes the inner layer of symmetric key encryption altogether, leaving the server with a pure FHE ciphertext of the user data. Finally, the FHE data can be processed and the homomorphic results are returned to the user.

In this work, we propose Proteus, a new FHE-transciphering methodology with support for *authenticated data storage* and *oblivious data verification*. Proteus relies on three main primitives:

- an efficient FHE transciphering method for authenticated encryption,
- feasible private computation algorithms for data verification, and
- the server's ability to obliviously manipulate the encrypted result.

Proteus' first primitive requires *authenticated FHE transciphering* on stored data. This deviates from previous works [10,2,20], which focus on protecting data in transit, with many works relying on *unauthenticated* stream ciphers. Unauthenticated stream ciphers are malleable using simple XOR operations. This allows attackers who can guess the underlying plaintext data to manipulate it in meaningful ways. Thus, transciphering based on stream ciphers alone lacks integrity protections, and requires a separate homomorphic message authentication code (MAC) solution to verify the integrity of the data. Toward this end, we leverage the versatile TFHE homomorphic encryption scheme [9] and evaluate the ASCON symmetric key encryption algorithm [14,28] to perform authenticated encryption and hashing homomorphically. Our choice of ASCON is further aligned with NIST's standard for lightweight cryptography [32]. In particular, ASCON's efficient use of bitwise operations makes it ideal for Boolean homomorphic evaluation using TFHE; likewise, since its design does not contain any complex mathematical operations, it allows the use of simpler FHE encryption parameters, yielding more efficient transciphering and hashing.

For the second Proteus primitive, the private computation algorithms depend on the context of the problem to be solved. In our work, we demonstrate real-life applications of private content moderation, targeting URL phishing detection and hate speech moderation datasets, as well as biometric authentication. The private content moderation application relies on token comparisons of data, whereas the biometric authentication relies on the L2 distance between biometric feature vectors.

Finally, Proteus' third primitive allows the server to take action against inauthentic or malicious data, as flagged by the first two primitives. This is done by locking the data through a second encryption layer, and homomorphically destroying the decryption key if the data was flagged.

Overall, our contributions can be summarized as follows:

- We analyze the benefits and address the challenges of the ASCON authenticated cipher and hash family to enable homomorphic transciphering using TFHE, and define a set of efficient FHE parameters tailored to efficient transciphering;
- We propose Proteus, a novel zero-trust protocol for data access control, which enables a remote server to homomorphically verify encrypted data and block downloads of malicious data or unauthorized data accesses;
- We evaluate the effectiveness and practicality of our methodology using real-life applications, including private content moderation, URL phishing detection, and privacy-preserving biometric authentication.

**Roadmap:** The rest of the paper is organised as follows: Section 2 discusses required preliminaries of ASCON and FHE encryption, while Section 3 defines the security considerations. Section 4 provides an overview of the Proteus protocol, followed by detailed implementation details in Section 5, while in Section 6 we discuss and evaluate three real-life applications. Finally, Section 7 compares our approach to related work, followed by our concluding remarks in Section 8.

## 2 Preliminaries

### 2.1 The ASCON Cipher Family

ASCON [14,15] is an authenticated cipher for lightweight encryption on constrained devices like embedded systems. In February 2023, NIST selected ASCON as the future standard for lightweight cryptography [28]. Here, we provide an overview of ASCON for FHE transciphering and hashing, and refer readers to the specification for full details [14,15].

ASCON uses a sponge construction with an internal state of five 64-bit integers, $x_0$ to $x_4$. ASCON's core comprises 4 steps: state initialization that consumes a key, a nonce and an IV, processing of associated data, data digest (encryption/decryption), and finally tag generation for authentication. During decryption (Figure 3), each ASCON block uses the internal state and the previous block's ciphertext for several permutation rounds, resulting in a modified internal state and a pseudorandom block to XOR with the next ciphertext.

**Permutation Rounds.** Each round has three steps: addition of a constant, substitution operations and linear diffusion.

- **Addition of a Constant:** ASCON XORs the least significant bits of state $x_2$ with a plaintext constant: $x_2 = x_2 \oplus const$.

– **S-box:** ASCON transposes the state's five 64-bit integers into sixty-four 5-bit slices for a 5-bit S-box. This S-box can be computed using logic gates (11 XOR gates, 5 AND gates and 6 NOT gates), allowing parallel bitwise operations on the 64-bit integers. Such a small substitution table makes ASCON ideal for Boolean FHE evaluation.

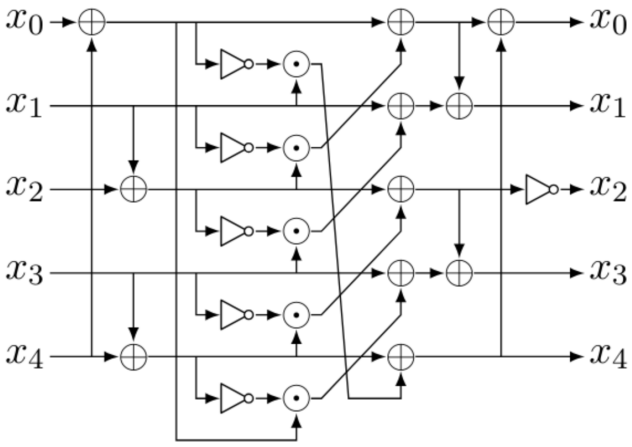


Fig. 2: **The ASCON S-box:** Circuit representation of ASCON S-box (v1.2), based on the CEASER competition submission [14].

– **Linear Diffusion Layer:** This step uses bitwise rotation and XOR operations to update all five 64-bit integers.

$$\begin{aligned} x_0 &= x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \\ x_1 &= x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \\ x_2 &= x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \\ x_3 &= x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \\ x_4 &= x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41) \end{aligned} \tag{1}$$

The total number of rounds depends on the ASCON variant, typically 12 rounds ($p^a$) for the initialization and finalization permutations, and 6-8 rounds ($p^b$) for processing the associated and plaintext data.

**ASCON Hashes.** The ASCON family includes variants ASCON-Hash and ASCON-HashA, differing by the number of permutations $p^b$ (12 vs. 8) for each block. Initialization uses a constant IV loaded into ASCON's state buffer $\{x_0, ..., x_4\}$, with specific constants for each hash variant [28]. During each digest step (Figure 3), 64-bit message blocks are XORed with state $x_0$ and undergo $p^b$ permutations. Hash generation (squeezing) involves $p^a$ permutations for the first

64-bit partial hash, followed by $p^b$ permutations for the remaining three, forming a 256-bit digest. Input padding uses a bit pattern of 1 followed by 0s, and at least one padding byte is required. Finally, both ASCON-Hash and ASCON-HashA can be turned into eXtendable Output Functions (called ASCON-XOF and ASCON-XOFA) to create arbitrary-size hashes. In this case, the initialization state differs for the two XOF variants, compared to the original hashes that yield 256-bit fixed-size digests.

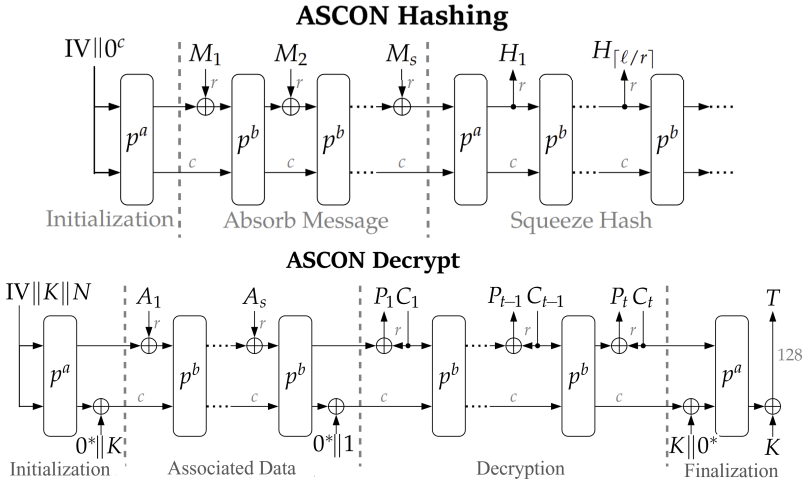


Fig. 3: **ASCON Cipher Operations:** ASCON supports both sponge-based hashing and decryption, using permutation boxes of $p^a$ and $p^b$ rounds. For FHE transciphering, we are primarily concerned with performing the ASCON decryption homomorphically.

**ASCON ciphers.** The ASCON family includes ASCON128, ASCON128a, and ASCON80pq for authenticated encryption with associated data (AEAD). ASCON128 and ASCON128a differ in block size (64 vs. 128 bits) and permutations $p^b$ (6 vs. 8), while ASCON80pq uses a 160-bit key and 64-bit block size. The initialization concatenates the IV, the key, and the nonce, followed by $p^a$ permutations and XORing the state with the key. Any associated data blocks are digested via XORs and $p^b$ permutations, and this step is always finalized by XORing $x_4$ with 1. For encryption, input blocks are XORed to produce ciphertexts, while decryption places ciphertexts into $x_0$ (for ASCON-128) or $\{x_0, x_1\}$ (for ASCON128a) and the state undergoes $p^b$ permutations before XORing each ciphertext with the previous block. Finally, authentication tags are generated by XORing the state with the key, applying $p^a$ permutations, and XORing $\{x_3, x_4\}$ with the key once again. Padding is similar to the ASCON-hash, except no padding is needed if associated data is not used.

### 2.2 Fully Homomorphic Encryption

**The LWE Problem.** The security of TFHE is based on the learning with errors (LWE) problem and its variants (e.g., Ring LWE and General LWE). We focus on the Torus LWE (TLWE) representation to discuss basic FHE concepts. More details on LWE ciphertexts can be found in [9].

The real torus $\mathbb{Z} = \mathbb{R}/\mathbb{Z} \bmod 1$ is the set of real numbers modulo 1. Here, we denote an inner product as $\langle \cdot, \cdot \rangle$. In TLWE, a message $m$ is encoded into plaintext $p$ by shifting $m$ by $\Delta$ and adding Gaussian noise $e$, so that $p = \Delta \cdot m + e$, as illustrated in Figure 4 (left). This is encrypted with a secret key $\boldsymbol{s}$ and a random **mask** $\boldsymbol{a}$, computing the body $b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + p$, resulting in the ciphertext $(\boldsymbol{a}, b)$. Decryption uses $\boldsymbol{s}$ to recover $p = b - \langle \boldsymbol{a}, \boldsymbol{s} \rangle = \Delta \cdot m + e$. Multiple values can be encrypted under the same $\boldsymbol{s}$ with different $\boldsymbol{a}$ and $e$.


Fig. 4: **(left)** *TFHE Shortint Ciphertext of 2-bit message:* This is a visualization of a TFHE ciphertext, where each square represents one bit. The illustration shows that the TFHE message bits are encoded in the upper bits, with noise added to the lower bits; **(right)** *Addition of Two Ciphertexts:* This figure shows an sample operation. Both the message size and noise grows; note that if too many operations are performed, the noise will start overwriting the message bits and corrupt the output. Noise can be reset with a bootstrapping operation.

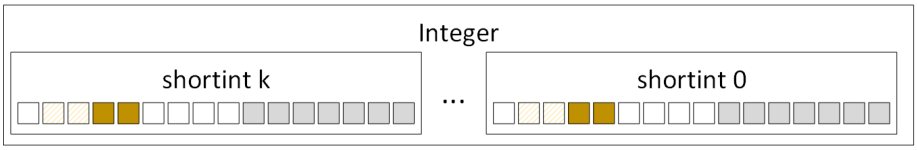


Fig. 5: TFHE Integer Ciphertext: This figure shows a TFHE integer representation, which is an array of shortints.

Alternative ciphertexts formats can also increase the number of dimensions in a message; for example, Torus Ring LWE encrypt data over a polynomial torus $\mathbb{T}_N[X] = \mathbb{R}[X] \bmod (X^N + 1) \bmod 1$, where N is a power of 2. Notably, use of linear equations allows an evaluation server to perform addition and multiplication on $m$, but this increases the noise $e$; if $e$ becomes too large, $m$ is unrecoverable.

**TFHE Bootstrapping and Keys.** Bootstrapping resets the noise in a TFHE ciphertext [8], allowing continued computations, but can incur high latency that depends on TFHE parameters. Functional bootstrapping reduces noise, while programmable bootstrapping (PBS) further allows lookup table (LUT) evaluations during bootstrapping, requiring more complex ciphertexts. Bootstrapping

also includes a *key switching* step that allows switching between ciphertext types, but can also enable switching encryption parameters between secret keys. Here, the client must provide a key switching key to the evaluation server. Overall, we use four main FHE keys: the client key for encryption/decryption, the public key for server encryption of constants, the server key for functional bootstrapping, and an optional key switching key for FHE parameter switching.

**TFHE-rs Encoding Types.** TFHE-rs offers three ciphertext encodings: Boolean, Shortint, and Integer [8,35]. Booleans encode messages as bits ($m \in \{0, 1\}$), while Shortints encode messages up to 8 bits with a carry buffer (allowing overflow) that requires periodic clearing; the default is 2-bit integers and 2-bit carry. Integer ciphertexts are arrays of shortints (either signed or unsigned), and are abstracted to types like FheUint8 to FheUint256. These use either a Chinese remainder theorem or a radix encoding, with radix allowing rotations while being slower.

In terms of computational complexity using FheUInt64 [35], bitwise operations (shift, rotate, AND, XOR) are most efficient (<20ms), while comparison operations (with encrypted boolean output) are slightly more complex (50-100ms). Addition and min/max functions incur higher cost (100-150ms), while multiplication and division (with PBS if the divisor is known) have the highest cost (200+ms). Notably, division by another ciphertext requires a slower bivariate PBS or repeated subtractions [36]. Table 1 shows the cost of TFHE FheUint64 operations based on TFHE-rs benchmarks [35].

Table 1: TFHE benchmark time: We report the cost of various TFHE operations on 64-bit integer ciphertexts. For certain operations, the latency is dependent on whether both arguments are ciphertexts, or if one is a ciphertext and the other a plaintext constant. The bold values are the operations used in ASCON, all of which are low latency.

| **Operation** | **f(ctxt, const)** | **f(ctxt, ctxt)** |
|---|---|---|
| Shift $\{\gg, \ll\}$ | 17.9 ms | 160 ms |
| Rotate $\{\ggg, \lll, \}$ | **18.4 ms** | 158 ms |
| Bitwise $\{\&, \vee, \oplus\}$ | **18.2 ms** | **17.8 ms** |
| Equality $\{=, \neq\}$ | 49.1 ms | 50.9 ms |
| Comparisons $\{\geq, >, \leq, <\}$ | 89.2 ms | 102 ms |
| Additions $\{+, -\}$ | 117 ms | 128 ms |
| Min/Max | 130 ms | 133 ms |
| Multiplication | 227 ms | 366 ms |
| Division | 391 ms | 8.83 s |

**Homomorphic Applications.** To provide better context for the possible applications of our Proteus protocol, we briefly discuss common homomorphic

applications. Notably, many complex algorithms have been implemented with TFHE, especially in machine learning. Basic statistics applications include linear regression, logistic regression, matrix multiplication, Manhattan distance, Euclidean distance, and Hamming distance, which are efficient benchmarks in TFHE [18,26]. Likewise, VGG-style convolutional neural networks (CNNs) using TFHE have been proposed: REDsec [16] classifies ImageNet-sized images in TFHE, while recent works focus on CIFAR10 [34] and MNIST classification (using a multilayer perceptron network) [29,5]. Here, the challenge is discretizing weights from floating point to n-bit integers, balancing weight size and FHE latency. REDsec uses 1-bit and 2-bit weights, while ConcreteML recommends 2-bit to 4-bit weights.

Recurrent neural networks (RNNs) for text processing remain an open research area. Trama *et al.* [31] developed long-term short memory (LTSM) blocks using TFHE, but without concrete applications. Zama built an attention module in ConcreteML [24] for a single layer in a larger GPT model. Random Forest and XGB-boost algorithms are efficient, and can run on the scale of seconds when precision is limited to 4 bits; sentiment analysis applications can use XGB-boost on a transformer-based output for text classification [25]. Likewise, Zuber *et al.* [37] developed a K-Nearest Neighbor algorithm, based on ConcreteML [34].

## 3 Threat Model Considerations

Our application assumes two parties: a client who aims to safeguard their data, and a server protecting their algorithm's intellectual property. While many FHE works only consider an honest-but-curious (i.e., semi-honest) model, our work will also consider scenarios with malicious clients and servers that can deviate from the protocol. In particular, a *semi-honest server* is incentivized to learn their clients' data (e.g., for advertising purposes), but will always execute the algorithm faithfully. Conversely, we assume that a *malicious server* is incentivized to extract the clients' data and is not guaranteed to faithfully execute the agreed-upon protocol (e.g., a hacked or coerced server). Similarly, *semi-honest clients* may attempt to download malicious or unauthenticated data, while still following the agreed protocol. Finally, we assume that *malicious clients* are motivated to bypass the server's security checks and are not guaranteed to follow the agreed protocol (for example, they can attempt to store malicious data on the server and attempt to bypass detection, or drop out of the protocol altogether).

**Key Management Considerations.** For our Proteus protocol, we assume the following: The client encrypts their data with the symmetric key and sends the ciphertext to the server. The client also generates four FHE keys: *client*, *server*, *public*, and *key-switching*. The client encrypts the symmetric key with their *client* FHE key and sends the FHE-encrypted symmetric key to the server. The client also sends the *server*, *public*, and *key-switching* FHE keys so the server can transcipher the symmetric ciphertext. Notably, the server never accesses the

unencrypted symmetric key or the client's FHE key, and cannot recover the plaintext.

In our implementation, we inherit the security of the ASCON variants, which operate at 128 bits of security, for data storage. We also rely on TFHE operating at 128 bits of security for Proteus' transciphering and data processing.

## 4 The Proteus Protocol

### 4.1 Motivation for Private Content Moderation

To provide context for the Proteus protocol, let us first motivate our private content moderation applications.

As we adapt to a digital world, private chat rooms have become a cornerstone for trusted communications. By offering secure channels, we foster individual autonomy, empower marginalized voices to be their genuine selves, protect witnesses and whistleblowers, and facilitate activism in oppressive regimes. While the right to privacy needs to be guaranteed to allow for protections and societal advancement, many nefarious actors often abuse this technology. Many private messaging communities are rife with objectively horrible things such as the trading of dangerous substances, disseminating hate speech, and sexual abuse and extortion. The presence of malicious behavior within these chat rooms can draw unwanted political and reputational scrutiny, posing a risk of potential bans or shutdowns for the platform. It is crucial to address and mitigate such behavior to safeguard the longevity and integrity of these integral web services. For example, after violent attacks, Telegram was forced to close public extremist channels after facing political and corporate pressure. However, no good solution has been available for monitoring closed private groups.

In our proposed approach, we allow a user post encrypted messages to a server. When computational resources are available, the server can use FHE transciphering to verify the data homomorphically. If the data is classified as malicious, the server will prevent the user (or any party holding the ASCON and FHE client keys) from downloading the data.

### 4.2 Overview of our Protocol

To introduce Proteus, we first assume the baseline case comprising a *semi-honest* user and a *semi-honest* server (malicious cases are discussed in section 5). In addition to the key ownership defined in Section 3, we will also use a second symmetric key, denoted $SK_2$, owned by the server. Our protocol is presented in Figure 6, and the corresponding steps are summarized as follows:

1. **User Symmetric Encryption:** The user encrypts their private data $M$ with a symmetric encryption key $SK_1$ and sends the encrypted data $E_1(M)$ to the server. They never share the symmetric encryption key with the server.

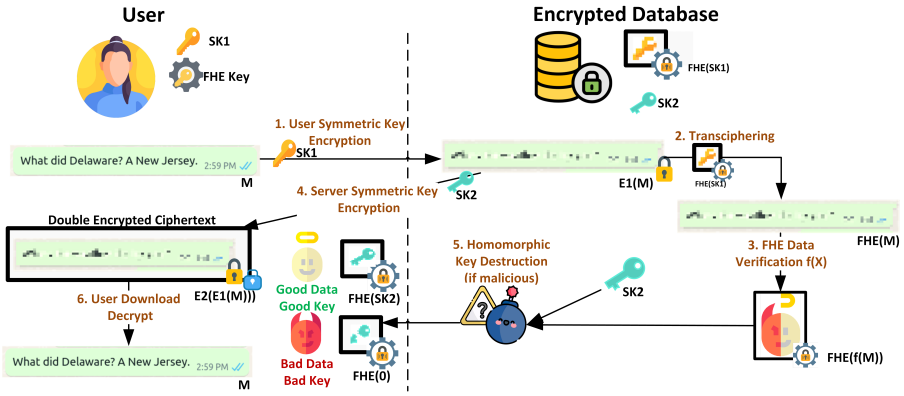


Fig. 6: **FHE Data Verification Overview:** Here we show an overview of our Proteus protocol. A server can store encrypted user inputs; when computational resources become available, the server can convert the ciphertext to FHE and homomorphically test if the data is valid and not malicious. Then, when the user wants to download the data, the server adds a second encryption layer and only reveals the outer key $SK_2$ to the user if the data is not labeled as malicious. By blocking downloads of malicious data, users are discouraged from storing malicious data on the server.

2. **Server Transciphering:** When the server is ready to process the data, they will need to ask the user to share a homomorphically encrypted symmetric key $FHE(SK_1)$. The server uses this key to "decrypt" the ciphertext homomorphically and retrieve the homomorphically encrypted message $FHE(M)$.
3. **Server Data Verification:** The server can verify the ASCON tag and perform any data verification algorithm $f(X)$. Convolutional neural networks and string comparisons are two such algorithms utilized in our work.
4. **Server Symmetric Encryption:** The server will generate a new secret key $SK_2$ and use a second symmetric encryption scheme to double-encrypt the original user data $E_2(E_1(M))$. This data can be sent to anyone that knows the user's symmetric key $SK_1$ and the user's FHE decryption key.
5. **Server Symmetric Key Destruction:** The server can homomorphically destroy its symmetric key if the data verification result fails by multiplying with $FHE(0)$; otherwise, the symmetric key remains intact $FHE(SK_2)$. In practice, this is possible using a homomorphic bitwise AND with the verification *result* (i.e., homomorphic 0 or 1). This yields an FHE encryption of a secret key, which we denote $FHE(SK_2 \cdot result)$, and is sent to the user.
6. **User Download/Decrypt:** The user now has $E_2(E_1(M))$, $SK_1$, $K_{FHE}$, and either $FHE(SK_2)$ or $FHE(0)$. If the data passes the verification step, the user can (a) FHE decrypt $FHE(SK_2 \cdot result)$ to retrieve $SK_2$, (b) decrypt $E_2(E_1(M))$ with $SK_2$ to remove the server's encryption layer $E_1(M)$, and (c) use their $SK_1$ to retrieve the message $M$.

# 5 Proteus Implementation

## 5.1 User Symmetric Encryption

In the first step, the user encrypts their data using a symmetric key cipher. As emphasized by our discussion, the choice of cipher is very important, as the server must decrypt the ciphertext homomorphically. The user then sends the encrypted message and message authentication tag to the server. A message authentication method is also required since this data will be stored in an online database and is a potential target for malicious parties; an attacker on a compromised server can modify user data without detection, violating data integrity. Therefore, Proteus utilizes ASCON for our authenticated symmetric key cipher, which we homomorphically evaluate in TFHE-rs.

As discussed in Section 2.1, ASCON's internal state is represented by an state array of five 64-bit integers. Using the TFHE-rs integer abstraction, this state can be represented with the FheUint64 data type. The default TFHE-rs parameterization of FheUint64 contains an array of 32 radix-encoded shortint ciphertext with 2 carry bits and 2 message bits. This option, which we call *parameter set A*, is useful for downstream operations, such as neural network evaluations. However, since ASCON's core uses bitwise operators and rotations operations exclusively, this allows for an even simpler parameter set for ciphertexts with 1 message bit and 1 carry bit (which we call *parameter set B*).

Still, the default parameters will also need to support multiplications and comparisons, which are essential for evaluating downstream algorithms on the FHE data, such as our private content moderation and biometric authentication. In this case, we can perform a *key switching operation* on FHE ciphertexts to switch between parameter sets. Key switching is possible during bootstrapping using the client's key switching key.

Table 2: TFHE Parameters for ASCON Evaluation.

| | ParamsA Integer | ParamsB Integer | ParamsC Boolean |
|---|---|---|---|
| LWE Dimension | 761 | 702 | 811 |
| GLWE Dimension | 1 | 3 | 3 |
| Polynomial Size | 2048 | 512 | 512 |
| LWE Noise | 6.37E-06 | 19E-05 | 5.28E-06 |
| GLWE Noise | 3.15E-16 | 3.97E-12 | 9.38E-10 |
| PBS Base | 23 | 18 | 10 |
| PBS Level | 1 | 1 | 2 |
| KS Base | 2 | 4 | 3 |
| KS Level | 5 | 3 | 5 |
| Message Bits | 2 | 1 | N/A |
| Carry Bits | 2 | 1 | N/A |

### 5.2 Server Transciphering

For the transciphering step, the server requires $FHE(SK_1)$, along with the FHE *server* and *public* keys. The server key is 26MB, which can limit long term storage. Thus, the server would only need to receive this key if the user asks to access/download the data. Luckily, once the user data is verified, the server can discard the keys and keep only the $FHE(SK_2)$ value, which is around 6kB for a 128-bit key.

In TFHE-rs, the output ciphertext can be encoded as an FheUint64 variable with limited carry padding or as a stream of bits. It is likely that a key switch should be performed to convert the ciphertext back into the default integer parameter set for future processing. The data can also be cast from FheUint64 to the appropriate data type, such as FheUint8 or FheUint16.

### 5.3 Server Data Verification

To tolerate malicious users, this step requires three distinct checks: tag verification, bounds verification, and malicious data detection. Each check returns an encrypted bit, which can be combined together with a homomorphic OR operation for a single verification bit. If any of the tests fail, the decryption key $SK_2$ will be locked, preventing the user from accessing the data.

The tag verification check will confirm the integrity of the data. This check will fail if the server received an incorrect tag, or an incorrect FHE-protected decryption key, or if an attacker compromised the server's database. This check can be implemented as a homomorphic comparison between the server-computed decryption tag (from transciphering) and the user-provided tag (after being FHE-encrypted). Overall, this comparison takes 198ms using a single thread.

The second check (bounds verification) is required to prevent certain attacks from malicious users, which are incentivized to provide false information in order to store malicious data on the server. For example, they can give a fake symmetric key, $FHE(SK_1^{forged})$, and forge a corresponding fake tag, so the tag verification test will pass. The goal of giving a fake key and tag is to confuse the malicious data detection algorithm and convince the server to release $FHE(SK_2)$. Then, the user will be able to decrypt with the valid $SK_2$ and their own $SK_1$.

The exact nature of the bounds verification test will depend on the nature of the data. If a fake key is given, the output ciphertext will be a pseudo-random stream of bits, which is straightforward to compare against patterned data. In our private content-moderation examples, we encode 15-bit tokens as 16-bit integers, so the verification is a simple homomorphic OR of randomly sampled most significant bits. Using 16 random bits gives a $2^{-16}$ chance of a false negative and takes only 312ms to compute. As an additional example, patterns of colors in picture data can be utilized to detect randomness (e.g., by sampling adjacent pixels and calculating their difference); if many adjacent pixels exhibit major differences in color, the test fails.

The third step focuses on malicious data verification. This test would require one the FHE-friendly applications mentioned at the end of section 2.2. For our

case study on text data, we employ string comparisons, while image data can rely on FHE convolutional neural networks. Notably, this computation will take up the bulk of the verification time.

Finally, the server must account for false positives for most implementations and applications. This needs to be implemented into a public security policy that will be application-dependent and is outside the scope of our Proteus protocol. However, we discuss two mitigating recommendations that can be adopted to prevent an unwarranted data lockout. In a nutshell, the server can use a threshold and block a user's data only if multiple violations are detected. Alternatively, the server can periodically send the original user a copy of their (encrypted) data verification results, which gives them a short grace period to download the data before locking it for everyone.

### 5.4 Server Symmetric Key Encryption

The server can double encrypt the data with $SK_2$ (allowing users to download the nested ciphertext at any time) and store both $E_2(E_1(M))$ and $SK_2$. Contrary to the constraints for selecting the transciphering algorithm $E_1$, the server can select any secure symmetric key algorithm $E_2$, as the user will run the $E_2$ decryption in the clear (i.e., not homomorphically). When the user is ready to download the data, the server can send the double encrypted message back to them.

For private content moderation applications, the data verified and the data downloaded are the exactly same. However, we remark that this does not need to be the general case in the Proteus protocol, as we showcase in Figure 8. Specifically, Proteus can be used to verify one set of data to provide access to another. One example would be biometric verification, where a server stores both encrypted biometric information and sensitive data. The user can send in an encrypted biometric scan and request a homomorphic biometric comparison for data verification. Then, the server can double-lock sensitive information only if the user biometrics are verified.

### 5.5 Server Symmetric Key Destruction

When the server computes the result of a homomorphic data verification, user privacy is preserved since the server is unable to read the FHE encrypted result, and only the user would be able to decrypt this information. As a case study, consider a server that asks the user to decrypt the boolean result and send it back to the server. There are two major issues here: First, a malicious user can lie to the server. To mitigate this, the server could obfuscate the result (e.g., XOR the result with a random bit) and homomorphically sign the boolean result.

A second and bigger issue would be the case of a malicious server. In our threat model, the server should not learn any information about user data, including whether the data is malicious or not, because this power can be abused to correlate/leak other private information. In this case, a malicious server could simply ask for the user to decrypt a bit of the ASCON key instead of the actual result, weakening the ASCON cipher security and allowing for a full key recovery after repeated queries.

Our proposed double encryption method allows the server to *obliviously lock* sensitive information when the data verification test fails. In more details, we homomorphically expand the verification check results (single FHE bit) into a vector of encrypted 1s for data that passed the test, or a vector of encrypted 0s for data that failed the test. We then use an AND operation to destroy the FHE-encrypted $SK_2$ key when data verification fails. This operation takes only 50ms on a single core. The server has the option to compress and store the homomorphically ANDed key, which is 6kB in size, to send to the user later. Doing so allows the server to delete the large 25MB TFHE-rs *server* and *public* keys.

### 5.6 User Download Decrypt

The server can now send the double-encrypted data and homomorphically encrypted $SK_2$ key to the user when the user is ready. The user needs to maintain a copy of the FHE client key and the symmetric ASCON key to be able to recover their plaintext data (unless they are destroyed).

## 6 Applications and Experimental Evaluation

In this section we evaluate several target applications, as well as the ASCON cipher family. We use a 24-core Intel i9 desktop for our experiments.

### 6.1 Application: Private Content Moderation

We demonstrate our Proteus protocol on two content moderation tasks on video game chat from the game "Dota 2". We target two forms of content moderation, focusing on english language chat messages. The first task is URL detection, to prevent phishing and spamming of the chat logs. Here, we aim to block 5 common URL words (HTTP, HTTPS, WWW, COM, TWITCH) to discourage users from spamming the chats. The second task expands this to a dataset of 211 hate speech and swear words from Surge AI [1] (which is compressed to 169 words during pre-processing). The goal is to employ the Proteus protocol to enable private chatrooms, while discouraging rooms that promote hate speech to continue to operate on the chat server.

**Data Encoding.** For data encoding, we use the BERT tokenizer API to convert our data into 15-bit tokens. Each token represents a series of letters; for example "Proteus" gets decomposed into 15-bit tokens {4013,2618,2271} representing {"Pro", "te", "us"}. This encoding leads to a compression of 23.1% compared to ASCII storage of our video game dataset, which contains lots of misspellings and leet-speak. More conventional text is expected to achieve higher compression, with BERT reaching 51.8% on Charles Dickens' "A Tale of Two Cities."

The blocked words are encoded as BERT tokens in the same way. For the URL detection example, all of the 5 sample words were encoded as a single token. For hate speech, we first filtered out 35 words that contained BERT token substrings that were also in the list. The resulting list contained 169 words comprising of {11, 84, 43, 21, 5, 0, 1} token strings of length {1,2,3,4,5,6,7} respectively.

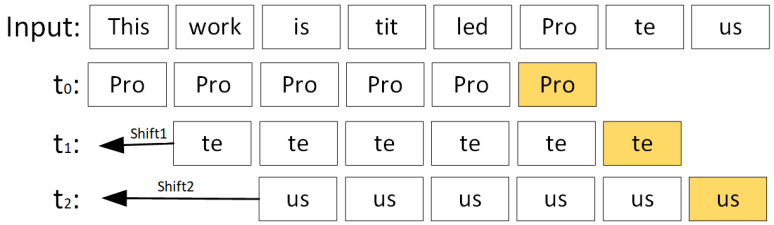


Fig. 7: **Homomorphic String Comparison:** This figure illustrates homomorphic string comparison with the tokens {"Pro", "te", "us"}.

**Homomorphic Data Verification.** After transciphering, the server utilizes a private set intersection algorithm on the tokens, using a sliding window. For the URL detection dataset, we employ a vectorized comparison of each token in the user message with the blocked tokens. Then, the server ORs this list together, for an inverted result, before applying the NOT-operation to get the expected encoding of 0 being malicious.

For hate speech, the blocked words span $t$ tokens, which increases computational complexity. In this case, the server performs four overlapping steps. First, the server applies a vectorized comparison of the input string for each unique token, resulting in $t$ output vectors. Then the server shifts the vectors by the offset, so $t_0$ remains the same, $t_1$ shifts by one space left, $t_2$ shifts two places left, and so on. These steps are shown in Figure 7. Next, the server ANDs the shifted vectors together to get a positive result at indexes where swear words are located. A homomorphic OR is finally performed operation across the vectors, followed by the NOT inversion.

**Evaluation.** Using the techniques proposed above, along with the ASCON128a protocol for $E_1$ and AES128 for $E_2$, we performed private content moderation on the Dota 2 video game dataset. For ASCON, we assumed the user sent the FHE-encrypted key and the full warmup period was calculated. We used message sizes of 1024 tokens for our ASCON decryption. For the URL detection, we were able to perform ASCON decryption and blocked word detection every 1.87 seconds per token, corresponding to about 1.44 seconds per character. For hate speech detection, we were able to achieve 10.00 seconds per token or about 7.68 seconds per character.

### 6.2 Application: Private Biometric Authentication

We also demonstrate how to leverage our protocol to protect biometric data in the cloud, which requires minor changes to the base Proteus scheme. Hosting biometric data on the cloud requires high privacy guarantees, as biometric features are permanently tied to an individual and data leaks may cause irreparable harm. Using Proteus, a user's biometric data can be protected using ASCON encryption, without the need to ever share the ASCON key with the cloud server, which guarantees user privacy. Later, the user can provide a current biometric scan to authenticate themselves. In this application, we assume the user has some private data encrypted on the server $E_3(M)$, and the user is the only one

possessing the symmetric encryption key $SK_3$. For example, the private data can be a secure token to activate other devices of the user, or document stored by the user, such as a password file.

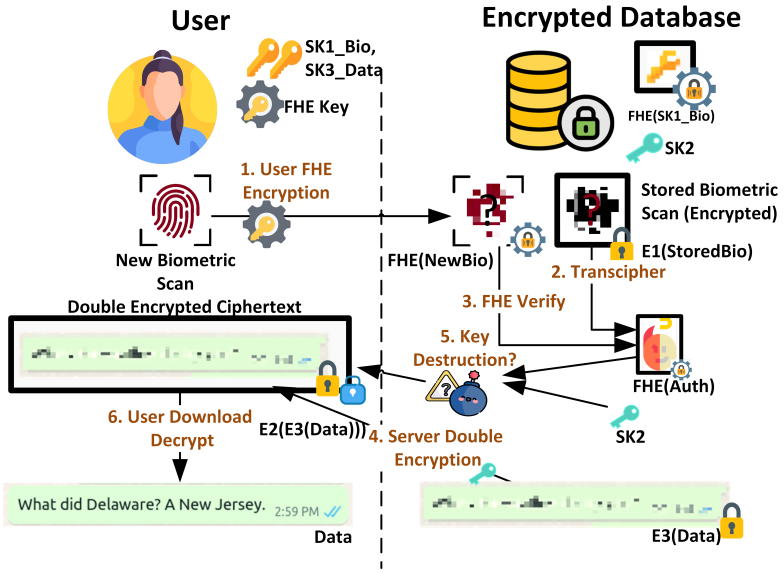


Fig. 8: **Biometrics with Proteus:** This is a variant of the base protocol, where we assume the server stores one set of encrypted data to verify and one set of encrypted data to download. Using the Proteus protocol, the user will only be able to open the data if they have been verified.

**Data Encoding and Registration.** For our dataset, the user scans their face and utilizes a neural network to extract a vector of 256 facial features. Our target neural network model is the nn4.small2.v1.t7, provided by the open computer vision library. For secure registrations, the user simply sends their biometric data $E_1(Bio_{reg})$ to the cloud, encrypted with the user's ASCON key (registration steps not visible in Figure 8). Notably, this approach avoids long-term storage of FHE data on the server to minimize storage requirements.

**Homomorphic Biometric Authentication.** For secure and private biometric authentication, the user will start the facial scan and extract the a new feature vector $Bio_{auth}$. As discussed, the goal is to authenticate the user and provide them with $E_3(M)$, if the verification check passes. The protocol steps are as follows:

1. **Client FHE Encryption:** The user will FHE-encrypt the new feature vector $FHE(Bio_{auth})$ and their original ASCON key $FHE(SK_1)$.
2. **Transciphering:** The server applies the Proteus protocol to transcipher the ASCON biometric feature vector (from registration) into $FHE(Bio_{reg})$.

Table 3: ASCON family variants: Parameters bit-length and number of rounds; $p^a$ denotes initialization rounds, while $p^b$ denotes digesting rounds.

| **Ciphers** | **Key** | **IV/Tag** | **Block** | $p^a$ | $p^b$ |
|---|---|---|---|---|---|
| ASCON128 | 128 | 128 | 64 | 12 | 6 |
| ASCON128a | 128 | 128 | 128 | 12 | 8 |
| ASCON80pq | 160 | 128 | 64 | 12 | 6 |
| **Hashes** | | **Hash** | **Block** | $p^a$ | $p^b$ |
| ASCON-Hash | | 256 | 64 | 12 | 12 |
| ASCON-Hasha | | 256 | 64 | 12 | 8 |

3. **FHE Authentication:** The server will first verify the authentication tag to ensure the underlying data was not modified. The server can then homomorphically compute the squared Euclidean distance between the registration and authentication vectors and compare it to a static threshold.
4. **Server Double Encryption**: The server will then double-encrypt the protected user data with $SK_2$ to compute $E_2(E_3(M))$ to be sent to the user.
5. **Homomorphic Key Destruction:** If both the tag verification and facial recognition tests pass (i.e., *result* is $FHE(1)$, the server can obliviously release user data using the Proteus double-encryption scheme; that is, the user gets either the decryption key or a zero vector depending on the homomorphic test result $FHE(SK_2 \cdot result)$.
6. **User Decryption:** If the biometric authentication passes, the user now has both $SK_2$ and $SK_3$ and can decrypt $E_2(E_3(M))$ to retrieve $M$.

**Evaluation.** We evaluated this case study using ASCON128a for encryption $E_1$, and AES128 for encryption $E_2$ and $E_3$. For ASCON, the user sent the FHE-encrypted key and the full warmup period was calculated. The private biometric authentication took 600.3 seconds, including 128.7 seconds for ASCON decryption of the 512-byte feature vector and 471.6 seconds for evaluating Euclidean distance on a vector of 256 16-bit integers. We remark that the user has the option to upload $E_1(Bio_{auth})$ and transciphering to obtain $FHE(Bio_{auth})$ (instead of uploading $FHE(Bio_{auth})$ directly). While this saves about 28kB in communication overhead, it incurs an additional 128 seconds of computation.

### 6.3 Performance Evaluation of ASCON under FHE

In our analysis, we evaluated four algorithms from the ASCON family: ASCON128, ASCON128A, ASCON-Hash and ASCON-HashA. Each algorithm was evaluated on message sizes from 1 block (64 or 128 bits) up to 2 kilobits. Since we aligned our messages with the block size, each message had 1 additional block of padding. We measured the warmup, throughput, padding, and cooling times across all message sizes, and the digest time for 1 kilobit was determined through

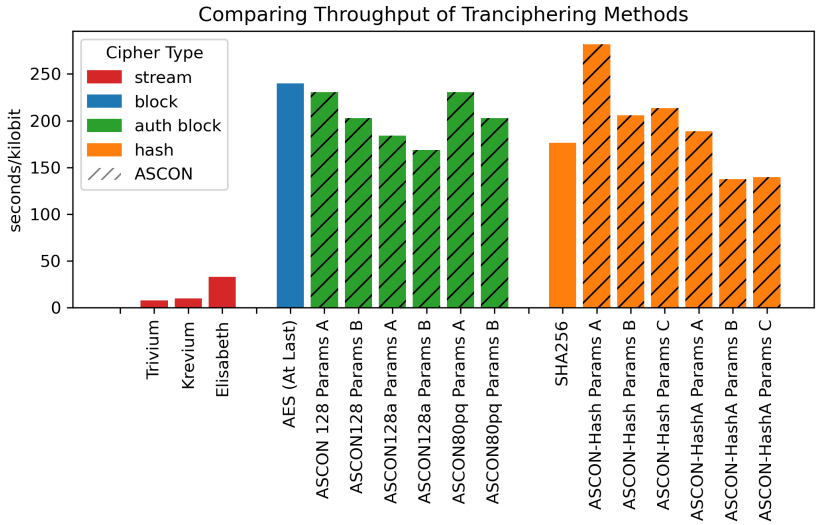


Fig. 9: **Digest Times** for 1 kilobit of FHE transciphering.

linear regression across all sample sizes. Our parameters are summarized in Table 4 and our evaluation results in Figure 9.

**Ciphertext Parameterization:** Our analysis shows that the simpler FHE ciphertexts achieved a significant speedup. Compared to the default (parameter set A), simpler parameter sets (i.e., B and C) offer a 8.3% to 12% speedup for decryption and 27.1% speedup for hashing.

**Choice of Cipher:** We further observe that ASCON-128a has a significantly lower latency than ASCON-128 by requiring fewer permutation rounds (i.e., 8 permutations on one 128-bit block, instead of 6+6 across two 64-bit blocks). A possible drawback of using ASCON128a is the potential for longer padding; our results show that messages aligned to 256 bytes were still faster for ASCON-128a. Regarding the warmup time, the state can be pre-initialized by the user, who would then share an FHE-encrypted 320-bit ASCON state instead of an FHE-encrypted 128-bit key. This incurs an extra 2.5x memory overhead, increasing the data size from 6 *kilobytes* to 18 *kilobytes*. This is negligible, however, if the FHE-encrypted ASCON keys are not stored on the server and is beneficial for smaller messages.

**Choice of Hash:** Lastly, we report that the ASCON-HashA is 33.1% faster than ASCON-Hash, since it requires fewer permutation rounds (i.e., 8 instead of 12) for a 64-bit digest. We also note that the hash times are longer than the authenticated cipher, since the ASCON hash algorithm requires more permutations to meet stronger security bounds compared to authentication tags. Notably, ASCON hashes do not require a warm-up phase since the initial state is constant and does not depend on a key or nonce.

# 7 Related Works

## 7.1 FHE Block Ciphers

AES is a common FHE benchmark and recent work has focused on optimizing FHE transciphering with AES [33,30]. One of the major challenges with FHE-AES implementations is the large 8-bit S-box. In plaintext, this can be implemented as a lookup table (LUT), since small LUTs are computationally feasible in TFHE. In FHE, however, even small LUTs require large polynomials, resulting in performance slowdowns.

Recent work entitled "At Last" [30] builds an AES implementation by packing each of the 8 output bits into a single lookup table, then further packing sets of input bits into a large polynomial. The authors experimentally determined that their 8-bit LUT can best be replaced by smaller 4-bit LUTs put into a tree structure. This method uses 16 LUTs per S-box lookup. The authors also use 32 smaller-sized LUTs for multiplications along with XORs in the MixColumns step, and XORs in the add round key step. In terms of performance, our results (Table 4 show that "At Last" requires 240 seconds per kilobit (without key expansion), while ASCON128a (params B) only requires 168.6 seconds.

In Fregata [33], the authors built an AES implementation by packing each of the 8 output bits into a single lookup table, then further packing sets of input bits into a large polynomial. This translates to 8 LUTs per S-box lookup. The drawback of this approach is that the ciphertexts are much bigger, making computations on them significantly slower. Fregata attempts to mitigate this by key switching to a less complex polynomial for the add round key and mix columns steps of AES, which are both implemented as XOR and rotate. In terms of runtime, Fregata requires 86 seconds for AES (we remark that since Fregata is not open source, was not possible to verify their results on the same hardware configuration as our work). Moreover, the runtime result do not include homomorphic key expansion or a mode of operation so the ciphertexts are unauthenticated and prone to tampering.

## 7.2 FHE Stream Ciphers

Stream ciphers are generally less computationally expensive than block ciphers, as they generate a stream of random bits to be XORed with a plaintext. Despite their efficiency, unauthenticated stream ciphers are not recommended as it is trivial for an adversary to change a bit without detection via XOR. In this case, a separate algorithm would be needed to generate a secure message authentication code (MAC). Despite these limitation, recent works in the FHE community have focused on developing FHE-friendly stream ciphers.

One major class of stream ciphers are nonlinear feedback shift registers (NFSR) based ciphers, which include Trivium and Kreyvium. Trivium is an ISO-standardized stream cipher that was selected in 2005 as part of European eStream low hardware-footprint cipher. It was designed to have 80 bits of security, and later, a 128-bit variant, Kreyvium, was created for homomorphic

Table 4: Evaluation and timimg comparisons to other ciphers. ASCON is the fastest block cipher and hash function, and is the only cipher that is authenticated. All times are in seconds per kilobit.

| | **Type** | **Warmup** | **Digest** | **Padding** | **Tag** | **Note** |
|---|---|---|---|---|---|---|
| Trivium [2] | Stream | 12.9 | 7.8 | - | - | 80-bits security |
| Kreyvium [6] | Stream | 17.1 | 9.7 | - | - | |
| Elisabeth | Stream | - | 32.8 | - | - | |
| AES [30] | Block | N/A | 240.0 | N/A | N/A | |
| ASCON 128 Params A | Block | 22.1 | 230.6 | 14.4 | 22.1 | Authenticated |
| ASCON128 Params B | Block | 19.3 | 202.8 | 12.7 | 19.3 | Authenticated |
| ASCON128a Params A | Block | 20.6 | 183.9 | 23.0 | 20.7 | Authenticated |
| ASCON128a Params B | Block | 18.1 | **168.6** | 21.1 | 18.1 | Authenticated |
| ASCON80pq Params A | Block | 22.4 | 230.4 | 14.2 | 22.1 | Authenticated |
| ASCON80pq Params B | Block | 19.9 | 203.0 | 12.7 | 19.1 | Authenticated |
| SHA256 | Hash | - | 176.5 | 88.3 | - | |
| ASCON-Hash Params A | Hash | - | 281.8 | 17.6 | 86.9 | |
| ASCON-Hash Params B | Hash | - | 205.7 | 12.9 | 75.7 | |
| ASCON-Hash Params C | Hash | - | 213.3 | 13.5 | 89.7 | |
| ASCON-HashA Params A | Hash | - | 188.5 | 11.8 | 74.9 | |
| ASCON-HashA Params B | Hash | - | **137.4** | 8.6 | 67.2 | |
| ASCON-HashA Params C | Hash | - | 139.2 | 8.4 | 69.3 | |

encryption applications. Both ciphers use a 288-bit state to generate an output bit. Trivium and Kreyvium are designed such that only state bits older than the previous 66 operations are used, allowing 64-bit streams can be computed in parallel. NFSRs are vulnerable to cube attacks after reducing the initialization from 1152 rounds down to 820 or fewer rounds [7,17]. These ciphers are still considered secure with the full number of rounds [2].

Another class of homomorphic-friendly stream ciphers is group filter permutators. These ciphers use a pseudo-random number generator (PRNG), seeded with a public initialization vector, to sample and permutate FHE-key bits before sending them to a filter function. Here, only the filter function needs to be implemented homomorphically. The first group filter permutator is FiLIP [20], which adds a whitening step in between the permutation and the filter function, where the permutated bits are XORed the PRNG-generated bits. A threshold function to produce a final output bit.

A second cipher, named Elisabeth-4 [10], was implemented with TFHE in mind, where data was packed efficiently in the polynomial fields of the ciphertext. This allows for an efficient 4-bit S-box based filter function. Gilbert *et al.* published a cryptanalysis of Elisabeth-4 a year after its release, and reduced its security from 128-bits to 88-bits. The concern was a known-IV linearization attack due to a vulnerability in Elizabeth's S-boxes.

### 7.3 Hash Functions

There is still limited work on TFHE-friendly hash functions. The authors of the TFHE library provide code for SHA-256 evaluation as one of their examples. Moreover, Bendoukha [3] proposes using off the shelf block ciphers such as Prince, SIMON, Speck, and LowMC for hashing. However, their results can only hash 128-bits in the scale of minutes, compared to ASCON and SHA256 which we achieve a kilobits in the same time (Table 4).

## 8 Conclusion

In this work, we present the Proteus methodology to allow for private but verifiable data access control on remotely stored data. The methodology guarantees the user privacy, even in the case of a malicious server that attempts to deviate from the protocol. Our implementation relies on the security guarantees on the state-of-the-art TFHE cryptosystem for homomorphic computation. Proteus also leverages the new ASCON cipher suite to protect the integrity of the data stored on the server. We demonstrated Proteus' capabilities with three case studies, successfully allowing a server to obliviously lock data for malicious or unauthenticated users.

## Acknowledgments

L. Folkerts and N.G. Tsoutsos would like to acknowledge the support of the National Science Foundation (Award 2239334).

## References

1. AI, S.: The obscenity list (2021), `https://github.com/surge-ai/profanity/tree/main`
2. Balenbois, T., Orfila, J.B., Smart, N.: Trivial transciphering with trivium and tfhe. In: Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 69–78 (2023)
3. Bendoukha, A.A., Stan, O., Sirdey, R., Quero, N., Freitas, L.: Practical homomorphic evaluation of block-cipher-based hash functions with applications. In: International Symposium on Foundations and Practice of Security. pp. 88–103. Springer (2022)
4. Bluefin: The Biggest Data Breaches of the Year (2024). `https://www.bluefin.com/bluefin-news/biggest-data-breaches-year-2024/` (2024), [Accessed 30-08-2024]
5. Bourse, F., Minelli, M., Minihold, M., Paillier, P.: Fast homomorphic evaluation of deep discretized neural networks. In: Annual International Cryptology Conference. pp. 483–512. Springer (2018)
6. Canteaut, A., Carpov, S., Fontaine, C., Lepoint, T., Naya-Plasencia, M., Paillier, P., Sirdey, R.: Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. Journal of Cryptology **31**(3), 885–916 (2018)

7. Che, C., Tian, T.: An experimentally verified attack on 820-round trivium (full version). Cryptology ePrint Archive (2022)
8. Chillotti, I.: Tfhe deep dive - part i - ciphertext types (May 2022), https://www.zama.ai/post/tfhe-deep-dive-part-1
9. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Tfhe: fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020)
10. Cosseron, O., Hoffmann, C., Méaux, P., Standaert, F.X.: Towards case-optimized hybrid homomorphic encryption: Featuring the elisabeth stream cipher. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 32–67. Springer (2022)
11. Cox, J.: The FBI tried to plant a backdoor in an encrypted phone network (2019), https://www.vice.com/en/article/pa73dz/fbi-tried-to-plant-backdoor-in-encrypted-phone-phantom-secure
12. Cox, J.: Dark Wire: The incredible true story of the largest sting operation ever. PublicAffairs, New York, NY (Jun 2024)
13. Crisan, L.: Launching default end-to-end encryption on messenger (2023), https://about.fb.com/news/2023/12/default-end-to-end-encryption-on-messenger/
14. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1. 2. submission to the caesar competition. Institute for Applied Information Processing and Communications, Graz (2016)
15. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1. 2: Lightweight authenticated encryption and hashing. Journal of Cryptology **34**, 1–42 (2021)
16. Folkerts, L., Gouert, C., Tsoutsos, N.G.: REDsec: Running Encrypted Discretized Neural Networks in Seconds. In: Network and Distributed System Security Symposium (NDSS). pp. 1–17 (2023)
17. Fouque, P.A., Vannet, T.: Improving key recovery to 784 and 799 rounds of trivium using optimized cube attacks. In: International Workshop on Fast Software Encryption. pp. 502–517. Springer (2013)
18. Gouert, C., Mouris, D., Tsoutsos, N.G.: SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks. Proceedings on Privacy Enhancing Technologies **2023**(3), 154–172 (Jul 2023). https://doi.org/10.56553/popets-2023-0075
19. Gouert, C., Tsoutsos, N.G.: Romeo: conversion and evaluation of hdl designs in the encrypted domain. In: 2020 57th ACM/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2020)
20. Hoffmann, C., Méaux, P., Ricosset, T.: Transciphering, using filip and tfhe for an efficient delegation of computation. In: Progress in Cryptology–INDOCRYPT 2020: 21st International Conference on Cryptology in India, Bangalore, India, December 13–16, 2020, Proceedings 21. pp. 39–61. Springer (2020)
21. Medida, N.: Amazon web services S3-leaks. https://github.com/nagwww/s3-leaks (2023)
22. Meta: Messenger end-to-end encryption overview (2023), https://engineering.fb.com/wp-content/uploads/2023/12/MessengerEnd-to-EndEncryptionOverview_12-6-2023.pdf
23. Meta: The Labyrinth encrypted message storage protocol v1 (2023), https://engineering.fb.com/wp-content/uploads/2023/12/TheLabyrinthEncryptedMessageStorageProtocol_12-6-2023.pdf
24. Meyre, A., Chevallier-Mames, B., Frery, J., Stoian, A., Bredehoft, R., Montero, L., Kherfallah, C.: Secure large language models using fully homomorphic en-

cryption (fhe). https://github.com/zama-ai/concrete-ml/tree/release/1.1.x/use_case_examples/llm (2023)
25. Meyre, A., Chevallier-Mames, B., Frery, J., Stoian, A., Bredehoft, R., Montero, L., Kherfallah, C.: Sentiment analysis with fhe. https://github.com/zama-ai/concrete-ml/blob/release/1.1.x/use_case_examples/sentiment_analysis_with_transformer/SentimentClassification.ipynb (2023)
26. Mouris, D., Tsoutsos, N.G., Maniatakos, M.: TERMinator Suite: Benchmarking Privacy-Preserving Architectures. IEEE Computer Architecture Letters **17**(2), 122–125 (2018). https://doi.org/10.1109/LCA.2018.281281
27. Nast, C.: Cops hacked thousands of phones. Was it legal? (2023), https://www.wired.com/story/encrochat-phone-police-hacking-encryption-drugs/
28. NIST: NIST Selects 'Lightweight Cryptography' Algorithms to Protect Small Devices. https://www.nist.gov/news-events/news/2023/02/nist-selects-lightweight-cryptography-algorithms-protect-small-devices (2023)
29. Sanyal, A., Kusner, M., Gascon, A., Kanade, V.: TAPAS: Tricks to accelerate (encrypted) prediction as a service. In: International Conference on Machine Learning. pp. 4490–4499. PMLR (2018)
30. Trama, D., Clet, P.E., Boudguiga, A., Sirdey, R.: At last! a homomorphic aes evaluation in less than 30 seconds by means of tfhe. Cryptology ePrint Archive (2023)
31. Trama, D., Clet, P.E., Boudguiga, A., Sirdey, R.: Building blocks for lstm homomorphic evaluation with tfhe. In: International Symposium on Cyber Security, Cryptology, and Machine Learning. pp. 117–134. Springer (2023)
32. Turan, M.S., Turan, M.S., McKay, K., Chang, D., Bassham, L.E., Kang, J., Waller, N.D., Kelsey, J.M., Hong, D.: Status report on the final round of the NIST lightweight cryptography standardization process. US Department of Commerce, National Institute of Standards and Technology (2023)
33. Wei, B., Wang, R., Li, Z., Liu, Q., Lu, X.: Fregata: Faster homomorphic evaluation of aes via tfhe. In: International Conference on Information Security. pp. 392–412. Springer (2023)
34. Zama: Concrete ML: a privacy-preserving machine learning library using fully homomorphic encryption for data scientists (2022), https://github.com/zama-ai/concrete-ml
35. Zama: Tfhe-rs v0.6 benchmarks (2024), https://docs.zama.ai/tfhe-rs/get-started/benchmarks
36. Zama: Which division algorithm does zama use for tfhe? https://community.zama.ai/t/which-division-algorithm-does-zama-for-tfhe/1019 (2024), [Accessed 09-07-2024]
37. Zuber, M., Sirdey, R.: Efficient homomorphic evaluation of k-nn classifiers. Proc. Priv. Enhancing Technol. **2021**(2), 111–129 (2021)