Compiler-assisted Semantic-aware Encryption for Efficient and Secure Serverless Computing

Bongjun Kim, Seonyeong Heo Student Member, IEEE, Jaeho Lee, Shinnung Jeong, Yongwoo Lee, Hanjun Kim, Member, IEEE

Abstract—Serverless computing like Function-as-a-Service (FaaS) is attractive for IoT service providers, liberating the providers from server maintenance. Since a data processing function is executed on the cloud instead of a dedicated server in the FaaS platform, the service users send their private data in their IoT devices to the third-party cloud, taking privacy leakage risks. Homomorphic encryption can preserve the privacy by enabling encrypted data processing on the cloud, but using homomorphic encryption for every data item incurs large computation and communication overheads. This work proposes SelectiveCrypt, a compiler-assisted semantic-aware encryption scheme that applies different cryptographic primitives depending on the operations on each data item. SelectiveCrypt homomorphically encrypts data items if arithmetic operations are applied to the data, while SelectiveCrypt encrypts data items with a symmetric key if the data are stored in the cloud without any arithmetic operation. The SelectiveCrypt framework consists of a compiler and its runtime system. The SelectiveCrypt compiler statically analyzes the data processing, determines an appropriate cryptographic primitive for each data item, and automatically transforms arithmetic operations into the homomorphic computation. The SelectiveCrypt runtime encrypts and decrypts the data items if arithmetic operations are applied to the data, while SelectiveCrypt homomorphically encrypts data items if arithmetic operations are applied to the data, while SelectiveCrypt homomorphically encrypts data items if arithmetic operations are applied to the data. The SelectiveCrypt runtime encrypts and decrypts the data items according to the static analysis result. This work evaluates the prototype SelectiveCrypt framework with five benchmarks that reflect real-world IoT scenarios. The evaluation results show that the SelectiveCrypt framework successfully reduces response time and communication overhead by 1.59 times and 9.61 times respectively compared with a homomorphic encryption scheme.

Index Terms—Cloud computing security, Internet of Things, homomorphic encryption

I. INTRODUCTION

Serverless computing like Function-as-a-Service (FaaS) facilitates to implement scalable IoT services without maintaining a dedicated server. Popular IoT platforms [1]–[4] support FaaS services such as AWS Lambda and Azure Functions. Using the FaaS services, users can load and execute a data processing function on the cloud on demand. Here, to offload data processing on the cloud, users should send plain IoT data to the cloud. For example, to execute a face recognition function on the cloud, users should send a camera image to the cloud.

Though FaaS liberates users from server maintenance, FaaS forces users to take privacy leakage risks on their IoT services. FaaS relies on third-party cloud platforms, and IoT devices send and store private user data to the untrusted cloud platforms. Since curious cloud service providers or malicious attackers may access to the plain IoT data on the cloud, the secure end-to-end communication is not enough for privacy preserving.

Previous work [5]–[10] has proposed encryption-based approaches to protect user data from the untrusted cloud. The work either uses symmetric-key encryption or homomorphic encryption. Symmetric-key encryption offers relatively fast encryption and decryption, but it disregards the data processing in the cloud. On the other hand, homomorphic encryption enables encrypted data processing on the cloud, but it incurs huge computation and communication overheads. Thus, for users to securely execute a data processing function on the untrusted cloud platforms, homomorphic encryption should be adopted in the FaaS service.

Although homomorphic encryption is unavoidable in the secure FaaS service, not all the data items require the homomorphic encryption. If a processing function in the FaaS service stores user data to the cloud storage just for archiving without any arithmetic operation, symmetric-key encryption is enough to protect the data. For example, if a processing function recognizes a known face with a camera image, and archives a video clip without any data processing, the function only needs to encrypt the camera image in homomorphic encryption. Symmetric-key encryption is enough to safely store the video clip in the cloud. Therefore, depending on the semantics of the processing function, the FaaS service can selectively encrypt user data either in homomorphic encryption.
or in symmetric-key encryption.

This work proposes SelectiveCrypt, a compiler-assisted semantic-aware encryption scheme that applies different cryptographic primitives such as homomorphic encryption and symmetric-key encryption to data items according to their usage patterns in a FaaS service. This work also implements a prototype SelectiveCrypt framework that automatically transforms a processing function in a FaaS service into a secured function, and supports encrypted data processing without modifying third-party cloud platforms. The framework consists of a compiler and its runtime system. The SelectiveCrypt compiler analyzes the function code and determines an appropriate cryptographic primitive for each data item. The compiler maps each data item into cryptographic primitives according to its semantic in the function, and passes the mapping information to the SelectiveCrypt runtime. With the mapping information, the SelectiveCrypt runtime conducts encryption and decryption on data items. To support lightweight IoT devices, the SelectiveCrypt framework also allows the devices to offload encryption and decryption to the SelectiveCrypt runtime on a trusted proxy. Figure 1 briefly illustrates the execution model of the SelectiveCrypt framework.

This work evaluates the prototype SelectiveCrypt framework with five benchmarks that reflect real-world IoT scenarios such as face recognizer, batting estimator, gas leak detector, character recognizer and sensor failure detector on the cloud. The evaluation results show that the SelectiveCrypt framework securely executes the benchmarks with 1.59 times shorter response time and 9.61 times less communication than a fully homomorphic encrypted scheme. In addition, with three microbenchmarks that implement representative IoT computations such as average, matrix multiplication and dot product, this work deeply analyzes the performance of offloading encryption and decryption from a weak device to a powerful proxy.

The contributions of this work are:
• The SelectiveCrypt scheme that selectively and efficiently encrypts user data either in homomorphic encryption or in symmetric-key encryption depending on the semantics of a processing function in a FaaS service
• The design of the SelectiveCrypt compiler that statically determines cryptographic primitives of data items and automatically transforms code to support encrypted data processing without modifying cloud platforms
• The design of the SelectiveCrypt runtime that provides seamless encryption and decryption across IoT devices and a trusted proxy

II. BACKGROUND & MOTIVATION
A. Function-as-a-Service (FaaS) in the IoT

Function-as-a-Service (FaaS) is an emerging type of cloud computing services. FaaS allows users to execute code on the cloud without building and maintaining servers. With FaaS services, users write, upload and execute a function code on demand. Unlike private cloud servers, FaaS services automatically manage multiple concurrent executions, and charge according to how many or how long the code is executed. Therefore, the FaaS service model can empower the IoT where the numbers of devices vary, and events occur sporadically.

Existing IoT platforms [1]–[4] support FaaS services such as AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions. The IoT platforms provide FaaS SDKs for users to write a function that manipulates and stores IoT device data. To execute the function in the FaaS services, the IoT platforms maintain connections with IoT devices and deliver messages from the IoT devices. In this way, the IoT platforms facilitate to implement IoT applications exploiting resources of the cloud. For example, an IoT application can use a FaaS service for face recognition, which would be a huge computational burden for IoT devices with low computation power.

B. Homomorphic Encryption

Homomorphic encryption (HE), originally proposed by Rivest et al. [11], provides homomorphism between ciphertext and plaintext that guarantees the decryption result of computed ciphertext to coincide with the computed value from plaintext. Depending on what operations and how much operations can be performed on encrypted data, there are several classes of homomorphic encryption such as additive homomorphic encryption (AHE), leveled homomorphic encryption (LHE), and fully homomorphic encryption (FHE). AHE only allows addition on ciphertext and LHE allows a certain number of multiplications depending on the security parameters. The FHE scheme, proposed by Gentry [12], allows an arbitrary
number of operations to be performed on encrypted data, but its applicability is limited due to its high execution overheads. Subsequent FHE schemes such as BGV [13], BFV [14], CKKS [15] and RNS-CKKS [16] have made significant efficiency improvement.

Among the FHE schemes, this work exploits the BFV scheme for homomorphic encryption. The security of the BFV scheme is based on the Ring Learning with Errors (RLWE) problem [17]. By utilizing the RLWE problem, the BFV scheme maps plaintext on the ring \( R_T := \mathbb{Z}[x]/(x^N + 1) \) to ciphertext on the ring \( R_Q := \mathbb{Z}[x]/(x^N + 1) \), where \( N \) and \( Q \) represents polynomial modulus degree, plaintext modulus, and ciphertext modulus, respectively. A user can guarantee a certain level of security by choosing these RLWE parameters. Since the RLWE parameters affect the multiplicative depth, encryption/decryption and computation time and security level, it is difficult for users without cryptography expertise to determine. With the advent of Microsoft SEAL [18], HELib [19], HEAAN [20], and PALISADE [21] that implement efficient FHE schemes including the BFV scheme as open source libraries, the user can easily apply the FHE schemes to their applications. For \( N \) and \( T \) selected by the user at a given security level, the SEAL library that this work uses generates secret and public keys with the corresponding hard-coded \( Q \).

C. Privacy Preserving with Encryption

Since the IoT platforms force users to send their private IoT data to the cloud in FaaS, the users should take privacy leakage risks on their IoT services. Although the IoT platforms provide an end-to-end secure channel to defend the privacy leakage against man-in-the-middle attacks, the platforms decrypt the data in the cloud to execute the processing function. Thus, the data in the cloud can be leaked by curious cloud service providers or malicious attackers. In other words, the end-to-end secure communication cannot preserve privacy from the untrusted cloud.

To securely process and store the data in the cloud, previous work [5]–[10], [22]–[24] suggests encrypting data before sending to the cloud. Some works [5]–[8] propose systems that automatically encrypt data for secure query processing in the cloud database. P²-SWAN [9] proposes a mobile computing framework that provides seamless data encryption in the edge. There also exist commercial software solutions [10], [22]–[24] that encrypt data stored in the cloud. These encryption-based approaches either use symmetric-key encryption or homomorphic encryption. Symmetric-key encryption is less computationally expensive than homomorphic encryption, but symmetric-key encryption disregards data processing in the cloud. Thus, the applicability of symmetric-key encryption is limited to cloud storage services only. On the other hand, homomorphic encryption enables to process encrypted data in the cloud. However, homomorphic encryption is much more computationally expensive than symmetric-key encryption, and thus increases performance overheads of the encryption.

Figure 2 shows the ciphertext size, encryption time, and decryption time of symmetric-key and homomorphic encryption for different plaintext sizes. This work uses AES and BF/V algorithms for symmetric encryption and homomorphic encryption, respectively. To evaluate the two encryption algorithms, this work executes the algorithms on Raspberry Pi 3B+. On average, AES obtains 4.01 times smaller ciphertext size, 58.6 times shorter encryption time, and 11.7 times shorter decryption time than BF/V. The results demonstrate that symmetric-key encryption incurs much less communication and computation overheads than homomorphic encryption.

Fortunately, the encryption overheads can be reduced because not all the data needs homomorphic encryption in the FaaS service. Homomorphic encryption is necessary only when the cloud processes data. If the cloud only archives data for logging or delivers data among devices as a bridge, homomorphic encryption is not necessary because there is no data processing in the cloud. Therefore, depending on the data usage patterns, the FaaS service can selectively apply different cryptographic primitives, reducing the homomorphic encryption overheads.

III. System Design

This work proposes an efficient semantic-aware encryption scheme for privacy preserving in the IoT, called SelectiveCrypt, and its compiler-runtime prototype framework. The SelectiveCrypt compiler analyzes a function code, decides cryptographic primitives for each data item between homomorphic encryption and symmetric-key encryption, and automatically transforms the original code to support encrypted data processing. The SelectiveCrypt runtime encrypts a data item with the corresponding cryptographic primitive before sending to the cloud. Here, without any modification on the cloud, the cloud can process the transformed code with the encrypted data. This section describes a threat model of this work, the SelectiveCrypt compiler and the SelectiveCrypt runtime in detail.

A. Threat Model

The IoT environment that the SelectiveCrypt framework operates consists of three participants: \textit{IoT device}, \textit{Proxy}, and \textit{Cloud}. An \textit{IoT device} publishes data items to topics or subscribes data items from interesting topics. When a user tries to deploy a function code, the user manually chooses and authenticates IoT devices as legitimate participants. A trusted \textit{proxy} is one of powerful IoT devices such as desktops and smartphones within the trusted network such as a home network. If there is no powerful nor trustworthy device, SelectiveCrypt only supports on-device encryption without offloading the encryption to the trusted proxy. Each IoT device and proxy runs a pre-installed SelectiveCrypt runtime that manages encryption and keys to protect application-level data. The \textit{cloud} belongs to third-party cloud service providers (e.g., Amazon AWS, Microsoft Azure, and Google Cloud) and provides various services such as computation (e.g., AWS Lambda) and storage (e.g., AWS S3, DynamoDB).

The cloud provides an authentication and authorization infrastructure to secure communication among participants and prevent man-in-the-middle attacks, but the cloud does not guarantee that it will not exploit user data beyond security
boundaries. Existing work [25]–[34] points out security concerns about the privacy and integrity of outsourced data storage and processing, assuming the cloud providers or attackers as potential threats. Especially, several papers [31]–[34] consider the cloud server as honest-but-curious, which means that the cloud server honestly executes requested services, but it may intentionally not forget user data. For example, although users request to delete their data, the cloud may not delete the data for their financial purposes. In addition, the cloud may use user data without authorization to train their object detection models or behavior prediction models.

This work assumes that the SelectiveCrypt runtime is installed on IoT devices and the trusted proxy with integrity. The deploy manager performs secure deployment processes, including key distribution and FaaS function code deployment. This work also assumes that data are delivered exactly where it should be delivered (confidentiality) without data manipulation (data integrity). Since this work considers only the cloud as a potential threat, the malicious act of either IoT devices or the trusted proxy is out of the scope of this threat model. Moreover, since the underlying SEAL library does not support comparison operations for homomorphically encrypted data, this work targets only applications where outsourced data are not used on conditional branches, and its threat model excludes the timing-based side-channel attack such as guessing the private data by exploiting unbalanced branches.

B. The SelectiveCrypt Compiler

The SelectiveCrypt compiler receives a function code that takes unencrypted inputs, and generates a function code that takes encrypted inputs. The compiler determines an appropriate cryptographic primitive for each data item in the function code, and transforms the original code to enable encrypted data processing. Note that the input function code contains computations that operate on plaintexts without any HE-specific in-

---

**Algorithm 1: The coloring algorithm**

```
1. **Input**: A data dependence graph \( G = (V, E) \)
2. **Output**: A coloring table \( CT \) that contains the assigned colors of all nodes in \( V \)
3. while \( V \) is not empty do
4.   \( v \leftarrow \text{Draw a vertex in } V \)
5.   \( CC \leftarrow \{v\} \)
6.   \( C \leftarrow \text{Color}_{\text{SYM}} \)
7.   
8.     
9.       
10.        
11.   
12.   
13.   
14.   
15.   
16.   
17.   
18.   
19.   
20.   
21.   
22.   
23.   
24.   
25.   
26.   

---

**Original Function Code**

```
1. image = get('/camera/image')
2. video = get('/camera/video')
3. result = recognize(image)
4. DB.save(video)
5. pub('/recog/result', result)
```

**Configuration File**

```
1. target_funcs: ['face_recog'],
2. receive_funcs: ['get'],
```

---

**Marked Data Dependence Graph**

```
Data Dependence Graph
1. Depends on
2. data source
3. perform arithmetic operations
```

---

**Data Dependence Graph**

```
1. Target Funcs: ['face_recog']
2. Receive Funcs: ['get']
```

---

Fig. 3. The overall compilation process of SelectiveCrypt

---

**Transformed Function Code**

```
1. def recognize(im, w1, w2):
2.   c = conv(im, w1)
3.   a1 = act(c)
4.   p = pool(a1)
5.   a2 = act(p)
6.   r = fc(a2, w2)
7.   return r
8. # Square Activation
9. def act(inp):
10.    out = []
11. for i in range(0, len(inp)):
12.     r = inp[i] * inp[i]
13.     out.append(r)
14. return out
```

---

**Encryption Table**

```
Data Identifier | Primitive
--- | ---
camera/image | HE
camera/video | SYM
```

---

**Decryption Table**

```
Data Identifier | Primitive
--- | ---
result | HE
```

---

Fig. 4. Parts of the face recognizer code example and its transformed code by SelectiveCrypt. reKey represents relinearization key.

---

(a) Original code

(b) Transformed code

---

Algorithm 1: The coloring algorithm

```
Input : A data dependence graph \( G = (V, E) \)
Output : A coloring table \( CT \) that contains the assigned colors of all nodes in \( V \)
1. while \( V \) is not empty do
2.   \( v \leftarrow \text{Draw a vertex in } V \)
3.   \( CC \leftarrow \{v\} \)
4.   \( C \leftarrow \text{Color}_{\text{SY M}} \)
5.   // Find connected components
6.   while \( Q \) is not empty do
7.     \( v \leftarrow \text{Draw a vertex in } Q \)
8.     \( CC \leftarrow CC \cup \{v\} \)
9.     if \( v \) is an arithmetic operation then
10.        \( C \leftarrow \text{Color}_{\text{HE}} \)
11.     end
12.     for \( (v', v) \in \text{Incoming}[v] \) do
13.        if \( v' \notin CC \) then
14.           \( C \leftarrow Color_{\text{HE}} \)
15.        end
16.     end
17.     for \( (v', v) \in \text{Outgoing}[v] \) do
18.        if \( v' \notin CC \) then
19.           \( C \leftarrow Color_{\text{HE}} \)
20.        end
21.     end
22.     // Update the color table
23.     for \( v \in CC \) do
24.        \( CT[v] \leftarrow C \)
25.     end
26.     \( V \leftarrow V \setminus CC \)
```

instructions. Figure 3 illustrates the overall compilation process for an example pseudo code. The example code implements a face recognition application for home security. When the face recognition function is triggered, the function acquires an image and a video clip from an IP camera (Instruction 1, 2), recognizes a face in the image (Instruction 3), archives the video clip to the database (Instruction 4), and publishes the recognition result (Instruction 5).

The compilation process consists of four major steps: (i) dependence analysis, (ii) marking, (iii) coloring, and (iv) code transformation. The first step is analyzing data dependences on a function code. When an instruction uses the result of the other instruction, the compiler marks the two instructions dependent. With the dependence analysis results, the compiler builds a data dependence graph, where each vertex represents an instruction and each edge indicates the dependence between two instructions.

The second step is marking arithmetic instructions, data sources and data sinks in the data dependence graph. An arithmetic instruction is an instruction that performs arithmetic operations. A data source is an instruction that extracts a data item from an event or acquires a data item from an IoT device. In the example code, Instruction 1 can be a data source because Instruction 1 extracts the image data from the inputs. A data sink is an instruction that publishes a data item. In the example code, Instruction 5 can be a data sink because Instruction 5 publishes the face detection result. In general, the compiler can find data sources and data sinks in the function code based on the communication API that the cloud platform provides.

The third step, coloring, is assigning appropriate cryptographic primitives for each instruction with the marked data dependence graph. Algorithm 1 explains how the compiler colors instructions. For each instruction that performs arithmetic operations, the compiler finds its reachable data sources and sinks in the dependence graph, and colors all the relevant instructions with HE (homomorphic encryption). Then, the compiler colors all the uncolored instructions including the uncolored data sources and sinks as SYM (symmetric-key encryption).

The final step is transforming the original code to support encrypted data processing. For data annotated with HE in the colored data dependence graph, the compiler transforms their source instructions to receive encrypted data. For example, since the input image is colored with HE in Figure 1, the compiler inserts an instruction that initializes the image object to contain ciphertext objects. Note that no transformation is required for the arithmetic operations between their sources and sinks because the Microsoft SEAL library [18], utilized by the SelectiveCrypt compiler as the backend, implements operator overloading for its arithmetic operations such as multiplication, addition, subtraction, squaring and exponentiation.

However, since the size of the ciphertext increases after each homomorphic multiplication, the compiler applies the HE-specific operation relinearize to the multiplication result immediately after each homomorphic multiplication. For example, Figure 4 shows how the compiler transforms parts of the recognize function code in the face recognizer into HE enabled code. The compiler inserts the relinearize instruction immediately after the homomorphic multiplication like Line 15 at Figure 4 (b).

Lastly, the compiler adds their sinks in the description table to inform that the published data are in form of ciphertexts.

As the compilation result, the compiler returns the transformed function code, an encryption table, and a decryption table. The encryption and decryption tables describes cryptographic primitives for each data item. Here, this work assumes that each data item has a unique data identifier, and the compiler can extract the data identifiers from the function code. Note that the format of data identifiers can differ across different IoT platforms. For example, for the IoT platforms that use the MQTT protocol, a data identifier can be an MQTT topic such as /camera/image.

C. The SelectiveCrypt Runtime

The SelectiveCrypt runtime deploys public keys to IoT devices, encrypts data items before sending the data items to the cloud, and decrypts encrypted data items received from the cloud. Figure 5 illustrates the overall process of the SelectiveCrypt runtime. The SelectiveCrypt runtime runs on IoT devices and trusted proxy. If an IoT device can have too weak computation power to support encryption and decryption, the SelectiveCrypt framework allows the device to offload encryption and decryption to a trusted proxy. In this case, the trusted proxy acts like a communication broker that
collects plain data items from IoT devices, encrypts the data items, and publishes the encrypted data items to the cloud on behalf of the IoT devices.

The deploy manager in the SelectiveCrypt deploys compiled results in four steps. First, the deploy manager uploads the transformed function code to the cloud (Step 1). Second, the deploy manager prompts the user to choose a master IoT device that will decrypt data items in the decryption table, and makes the IoT device generate keys for encryption and decryption (Step 2 and 3). Finally, the deploy manager distributes the encryption table, the decryption table, and the keys to IoT devices (Step 4).

The deploy manager uses different key distribution mechanisms for symmetric-key encryption and homomorphic encryption. For symmetric-key encryption, the deploy manager distributes the same symmetric key to the IoT devices that will encrypt or decrypt data items with symmetric-key encryption. For homomorphic encryption, the deploy manager distributes the same public key to the devices that will encrypt data items with homomorphic encryption. In addition, the deploy manager delivers the public key to the cloud because the cloud also needs a public key to process the data items.

For the example in Figure 5, the deploy manager requests the door lock to generate a pair of private and public keys. According to the request, the door lock generates keys, and passes the public key to the deploy manager. The deploy manager distributes the public key for other devices to apply homomorphic encryption.

The deploy manager keeps a private key for decryption in an only one IoT device, called master device. Since only the master IoT device has a private key, the device can decrypt data items encrypted with homomorphic encryption, and the other devices including the cloud can only process data items without decryption. This work introduces the master runtime that mediates data decryption as an extension, and installs the runtime extension on the master device.

The SelectiveCrypt runtime automatically encrypts and decrypts data items according to the encryption and decryption tables. The online part of Figure 5 illustrates how the runtime encrypts and decrypts data items for the example function in Figure 3. Before sending a data item to the cloud, the runtime checks whether the data item is in the encryption table, and encrypts the data item using the prescribed cryptographic primitive. For example, after the runtime triggers the lambda function (Step \( \Theta \)), the runtime on the device encrypts the data such as video and image on the device (Step \( \Theta \) and \( \Theta \)).

When receiving a data item from the cloud, the runtime checks whether the data item is in the decryption table. If so, the runtime decrypts the data item using the prescribed cryptographic primitive. For example, the trusted proxy receives a data item which identifier is \( \text{/recog\_result} \), it finds the data item in the decryption table and decrypts the data item using a private key on behalf of a resource-constrained doorlock (Step \( \Theta \)). If the runtime has no key to decrypt the data item, the runtime reports an error that it fails to decrypt the data item. Here, while the SelectiveCrypt runtime triggers an event and receives a termination signal through the MQTT protocol, the runtime sends program input data and receives output data through the S3 storage service due to the payload size limitations of the MQTT protocol itself.

If an IoT device cannot afford SelectiveCrypt runtime, the device offloads its cryptographic task into the trusted proxy. For example, since the doorlock is too weak to perform cryptographic tasks, the deploy manager installs the decryption table in the trusted proxy and delegates decryption authority to the trusted proxy. On the other hand, when the device has strong computation power like the IP camera, its runtime encrypts the data such as video and image on the device.

IV. Security Analysis

This section proves that the SelectiveCrypt framework maintains the higher or equal level of security compared to the existing homomorphic-only encryption. Since the SelectiveCrypt framework encrypts parts of data using symmetric-key encryption rather than homomorphic encryption, the problem to prove is whether the semantic-aware hybrid encryption scheme can have the higher or equal security level compared to homomorphic-only encryption.

Table 1 shows notations used in this section. With given encryption parameters \( N \), \( Q \) and \( T \), the homomorphic encryption scheme generates a key \( hk \). Here, \( N \) is polynomial modulus degree for polynomial modulus \( x^N + 1 \). \( Q \) and \( T \) are coefficient modulus and plaintext modulus respectively. The symmetric encryption scheme generates a key \( sk \) with a key size, \( m \)-bits. \( E_{hk}(d_i) \) and \( E_{sk}(d_i) \) stand for ciphertexts that the homomorphic encryption and symmetric encryption schemes encrypt \( i \)-th data \( d_i \), respectively. \( \lambda_{E_{hk}}(d_i) \) and \( \lambda_{E_{sk}}(d_i) \) are security levels measured in bits. \( \lambda_{E_{hk}}(d_i) \) and \( \lambda_{E_{sk}}(d_i) \) mean that \( 2^{\lambda_{E_{hk}}(d_i)} \) and \( 2^{\lambda_{E_{sk}}(d_i)} \) operations are required to break the homomorphic encryption and symmetric-key encryption of the data \( d_i \) encrypted with the secret keys \( hk \) and \( sk \), respectively.

**Problem statement:** The semantic-aware hybrid encryption preserves a security level that is equal to or higher than homomorphic encryption scheme.

**Assumptions:** For the security analysis, this work assumes followings:

- **Security parameters:** A user can guarantee a certain level of security by choosing encryption parameters such as \( N \), \( Q \), \( T \) for homomorphic encryption scheme and \( m \) for symmetric-key encryption scheme. For example, this work chooses \( N \) as 8192 for polynomial modulus \( x^N + 1 \) and uses hard-coded default coefficient modulus \( Q \) to guarantee 256-bits security level corresponding to AES-256-bit security level. For symmetric encryption, by specifying a key size, \( m \)-bit, as 256 or higher bits, the following holds true.

\[
\lambda_{E_{sk}}(d_i) \geq \lambda_{E_{hk}}(d_i)
\] (1)

- **Security key and level of multiple cryptographic primitives:** The security level of a system that uses multiple encryption schemes is the security level of a scheme that requires the least amount of operations to break it.

**Theorem 1.** For the adversary to attack the semantic-aware encrypted ciphertexts, the number of required operations is
A key for homomorphic or symmetric encryption

Homomorphic encryption security parameters

Symmetric encryption key size

λ

Homomorphically encrypted

Notation

lectiveCrypt framework with Amazon Web Services (AWS),

and

E

We will prove Theorem 1 by induction that for

min

ck

is

A key for homomorphic encryption

A key for symmetric encryption

A key for homomorphic or symmetric encryption

E

E

E

Thus, Theorem 1 holds for all the positive integer

Proof.

Lemma 1.1.

Thus, \( \lambda_{E_{ck}(d_i)} \geq \lambda_{E_{hk}(d_i)} \)

Base Case: When \( n = 1 \), \( \min(\lambda_{E_{ck}(d_1)}) \geq \min(\lambda_{E_{hk}(d_1)}) \)

From Lemma 1.1,

\[
\min(\lambda_{E_{ck}(d_i)}) = \lambda_{E_{hk}(d_i)} \geq \lambda_{E_{hk}(d_i)} = \min(\lambda_{E_{hk}(d_i)})
\]

\( \square \)

\( \square \)

equal to or higher than the number of required operations to

attack homomorphically encrypted ciphertexts.

\[
\min(\lambda_{E_{ck}(d_1)}, \lambda_{E_{ck}(d_2)}, \ldots, \lambda_{E_{ck}(d_n)}) \\
\geq \min(\lambda_{E_{hk}(d_1)}, \lambda_{E_{hk}(d_2)}, \ldots, \lambda_{E_{hk}(d_n)}) \quad (2)
\]

for \( n \) data items. Here, \( ck \) is either \( hk \) or \( sk \).

\( \lambda_{E_{ck}(d_i)} \geq \lambda_{E_{hk}(d_i)} \)

Proof. We will prove Theorem 1 by induction that for \( n \) data items, \( \min(\lambda_{E_{ck}(d_1)}, \ldots, \lambda_{E_{ck}(d_n)}) \geq \min(\lambda_{E_{hk}(d_1)}, \ldots, \lambda_{E_{hk}(d_n)}) \).

Case \( ck \) is \( hk \): \( \lambda_{E_{ck}(d_i)} = \lambda_{E_{hk}(d_i)} \)

Case \( ck \) is \( sk \): \( \lambda_{E_{ck}(d_i)} = \lambda_{E_{hk}(d_i)} \geq \lambda_{E_{hk}(d_i)} \) from Eq. (1).

To simplify the equation, let us introduce two variables, \( E_{ck} \) and \( E_{hk} \).

\[
E_{ck} = \min(\lambda_{E_{ck}(d_1)}, \ldots, \lambda_{E_{ck}(d_i)}) \\
E_{hk} = \min(\lambda_{E_{hk}(d_1)}, \ldots, \lambda_{E_{hk}(d_i)})
\]

From the induction hypothesis, \( E_{ck} \geq E_{hk} \).

From Lemma 1.1, \( \lambda_{E_{ck}(d_{k+1})} \geq \lambda_{E_{hk}(d_{k+1})} \).

Case \( E_{ck} \geq \lambda_{E_{ck}(d_{k+1})} \):

\[
\min(\lambda_{E_{ck}}, \lambda_{E_{ck}(d_{k+1})}) = \lambda_{E_{ck}(d_{k+1})} \\
\geq \lambda_{E_{hk}(d_{k+1})} \\
\geq \min(\lambda_{E_{hk}}, \lambda_{E_{hk}(d_{k+1})}) \quad (3)
\]

Case \( \lambda_{E_{ck}(d_{k+1})} \geq E_{ck} \):

\[
\min(\lambda_{E_{ck}}, \lambda_{E_{ck}(d_{k+1})}) = \lambda_{E_{ck}} \\
\geq \lambda_{E_{hk}} \\
\geq \min(\lambda_{E_{hk}}, \lambda_{E_{hk}(d_{k+1})}) \quad (4)
\]

From Eq. (4) and (5), the induction hypothesis is true. Thus, Theorem 1 holds for all the positive integer \( n \).

V. EVALUATION

This work implements and evaluates the prototype SelectiveCrypt framework with Amazon Web Services (AWS),

which supports both IoT and FaaS services. This work uses the AES-256 implementation in the OpenSSL library for symmetric-key encryption and the BFV [14] implementations in Microsoft SEAL library [18] for homomorphic encryption and its python wrapper Pyfhel library [35] and pyHeal library [36]. We have made a prototype implementation of SelectiveCrypt publicly available via GitHub [37].

To ensure the \( \lambda \)-bit security level in homomorphic encryption scheme, this work chooses the encryption parameters by referring to the recommendation parameter tables listed in [18], [38]. However, encryption parameters affect not only the security level, but also the number of multiplications that can be performed on a ciphertext, that is, multiplicative depth. Since the FaaS service is designed for tasks that are executed for a short time on requests and provides limited memory usage and short time-out, it is suitable for running relatively shallow artificial neural networks. Therefore, this work chooses fixed encryption parameters to be multiplicatively deep enough to execute the neural network defined in Microsoft CryptoNets [39] correctly. If the user wants to support arbitrary multiplicative depth, SelectiveCrypt can be extended to use bootstrapping to support the FHE scheme.

This work chooses 8192 for polynomial modulus degree \( N \) where polynomial modulus is a form of \( x^N + 1 \). By choosing polynomial modulus degree \( N \) and desired security level (256-bit), SEAL library gives hard-coded default values for coefficient modulus \( Q \) where \( Q \) is 118 bits in length. For plaintext modulus \( T \), this work chooses 1099511922689 where \( T \) can be any positive integer at most 60 bits in length.

To compare the computation efficiency of different computing models, this work incorporates three different IoT system participants: a cloud platform, a trusted proxy, and IoT devices. This work uses various cloud platform services such as Amazon AWS Lambda, IoT Core, and S3 services, which provide function execution, communication management, and cloud storage, respectively. This work also uses a desktop server and an embedded board, Dell XPS 8700 (Intel Core i7-4770) and Odroid N2 (ARM Cortex-A53) as trusted proxies, and small compute boards, Raspberry Pi 3B+ as IoT devices.

A. Benchmark Description

To evaluate the SelectiveCrypt framework, this work implements five benchmarks that reflect real-world IoT scenarios.

- Privacy-preserving Face Recognizer receives an image and a video clip from an IoT device, uses CryptoNets [39] to perform inference, and saves the video clip to the cloud storage. CryptoNets is a deep neural network that operates on homomorphically encrypted data. The SelectiveCrypt

<table>
<thead>
<tr>
<th>Table I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notation</td>
</tr>
<tr>
<td>( N, Q )</td>
</tr>
<tr>
<td>( m )</td>
</tr>
<tr>
<td>( hk )</td>
</tr>
<tr>
<td>( sk )</td>
</tr>
<tr>
<td>( ck )</td>
</tr>
</tbody>
</table>
encrypts the image with homomorphic encryption and the video clip with symmetric-key encryption.

- **Batting Estimator in Sport** receives hitting data of a baseball player, and runs a multi-layer perceptron to predict next turn behavior. Perceptron is a single layer neural network that consists of one output layer and activation function. SelectiveCrypt encrypts the hitting data with homomorphic encryption while encrypting all the strategies for the player with symmetric-key encryption.

- **Gas Leak Detector** receives encrypted data from gas sensors, and uses a perceptron to validate gas leak. SelectiveCrypt encrypts sensor values with homomorphic encryption while encrypting their sensor position and nearby temperatures with symmetric-key.

- **Hand-written Character Recognizer** receives encrypted hand-written numbers on the check and return digits using homomorphic logistic regression. In addition, the metadata of the user are encrypted by symmetric-key and saved in the cloud.

- **Sensor Failure Detector** analyzes 2,000 sensors using linear regression, and notifies a failure to the owner if a sensor shows strange behavior. SelectiveCrypt encrypts the sensor values with homomorphic encryption, and their metadata with symmetric-key encryption.

**B. Execution Time and Communication Amount**

For the benchmarks, this work measures the execution time and communication amount of the IoT device for six different computing models. In order to analyze the impact of semantic-awareness for a single transaction, this work measures the execution time and communication amount of the IoT device by setting their batch size as one.

- **Homomorphic encryption on device without proxy (HE-D):** An IoT device encrypts all data items with homomorphic encryption and directly sends the encrypted data items to the cloud. The cloud executes the benchmark with the data items and returns the encrypted result to the device. The IoT device decrypts the result.

- **Homomorphic encryption with weak/powerful proxy (HE-WP/HE-PP):** An IoT device sends plain data items to the computationally weak or powerful proxy. The proxy encrypts all the data items with homomorphic encryption and sends the encrypted data items to the cloud. The cloud executes the benchmark with the data items and returns the encrypted result to the proxy. The proxy decrypts and sends the result to the IoT device.

- **Semantic-aware encryption on device without proxy (SE-D):** An IoT device encrypts data items with different cryptographic primitives according to the usage of the data items in the benchmark. Then, the IoT device directly sends the encrypted data items to the cloud. The cloud executes the benchmark with the data items and returns the encrypted result to the device. The IoT device decrypts the result.

- **Semantic-aware encryption with weak/powerful proxy (SE-WP/SE-PP):** An IoT device sends plain data items to
Fig. 8. The encryption time and ciphertext size for different homomorphic and symmetric-key encryption ratios

the proxy. The proxy encrypts data items with different cryptographic primitives according to the usage of the data items, and sends the encrypted data items to the cloud. The cloud executes the benchmark with the data items, and returns the encrypted result to the proxy. The proxy decrypts and sends the result to the IoT device.

Figure 6 shows the execution time breakdown of the benchmarks for different computing models. This work partitions the total execution time into initialization, device-proxy communication, encryption, cloud communication and computation. Initialization includes the setup time of homomorphic encryption. Device-proxy communication means communication latency between an end device and the proxy. Encryption consists of homomorphic and symmetric-key encryption time. Cloud communication means communication latency between a device or the proxy and the cloud. Cloud computation means the computation cost to execute the benchmark on the cloud.

Figure 6 shows that SelectiveCrypt reduces total execution time by 1.50 times and 1.59 times on geomean for SE-WP and SE-PP models compared with HE-D model respectively. SelectiveCrypt reduces encryption and communication latency by avoiding homomorphic encryption for storage only data items. Moreover, proxy computing further reduces the encryption time by offloading time-consuming encryption to the proxy.

SelectiveCrypt reduces encryption and communication latency by avoiding homomorphic encryption for storage only data items. Moreover, proxy computing further reduces the encryption time by offloading time-consuming encryption to the proxy.

Figure 7 shows the communication amount breakdown of the benchmark execution for different computing models in perspective of the IoT device. The SE-D model obtains 9.61 times less communication amount on geomean than HE-D model by encrypting a data item with symmetric-key encryption instead of homomorphic encryption. Here, the communication amounts of HE-WP, HE-PP, SE-WP and SE-PP are small because the IoT device sends plain data items to the proxy without any encryption. The evaluation results show that the semantic-aware encryption models largely reduce the time for sending the encrypted data items to the cloud because homomorphic encryption generates much larger ciphertext than symmetric-key encryption from the same plaintext.

Fig. 9. The execution time breakdown of each microbenchmark for different element sizes

C. Sensitivity Analysis on Encryption Types

Since SelectiveCrypt reduces the response time by replacing homomorphic encryption into symmetric-key encryption, the ratio between homomorphic and symmetric-key encryption highly affects the overall performance. Figure 8 shows that the encryption time and ciphertext sizes for different encryption ratios. As the ratio of symmetric-key encryption increases, the total encryption time and ciphertext size linearly decrease. Considering the ciphertext size affects the communication overhead, applying symmetric-key encryption as much as possible maximizes the performance improvement.

D. Performance Analysis on Proxy-oriented Outsourcing

This work designs three microbenchmarks to evaluate how the trusted proxy can reduce computation overheads of encryption and decryption, considering the frequently used computations in IoT applications.

- **Average (AV)**: AV calculates the average for floating point numbers. AV is for evaluating the addition performance.
- **Matrix multiplication (MM)**: MM conducts two-dimensional matrix multiplication. MM is for evaluating the multiplication and batching performance.
- **Dot product (DOT)**: DOT calculates the dot product of two-dimensional matrices.

For each microbenchmark, this work measures the execution time for two different computing models:

- **Device encryption (D)**: An IoT device encrypts all data items and directly sends the encrypted data items to the cloud. The cloud executes a microbenchmark with the encrypted data items and returns the encrypted result to the device. The IoT device decrypts the result.
- **Proxy encryption (P)**: An IoT device sends plain data items to the proxy. The proxy encrypts the data items, and sends the encrypted data items to the cloud. The cloud executes a microbenchmark with the encrypted data and returns the encrypted result to the proxy. The proxy decrypts and sends the result to the IoT device.

Note that all the microbenchmarks consist of pure functional computations, and thus all data items are supposed to be encrypted with homomorphic encryption.

Figure 9 shows the execution time breakdown of each microbenchmark for different element sizes. This work partitions
Introduction of a new cryptographic primitive for join operations, CryptDB can protect data from compromised database or administrators without harming the semantics of queries. Due to large computational cost, CryptDB avoids homomorphic encryption for all and uses homomorphic encryptions for specific operations only.

Extending CryptDB [5], Shafagh et al. [6], [7] propose IoT systems to prevent data breach from database compromise and network-based attacks. For resource-constrained IoT devices, Talos [6] and Poster [7] optimize order-preserving and homomorphic encryption algorithms by tailoring the algorithms for integers. To exploit near-user computation with edge computing [41], [42], Talos and Poster allow IoT devices to encrypt and decrypt their own data without a trusted proxy, which offers more privacy.

Pilatus [8] further extends Talos [6] to enable secure data sharing between different users. With Pilatus, a user generates a token for the cloud to re-encrypt data to share the data with another user. Pilatus allows users to revoke their keys to terminate data sharing. Compared with CryptDB [5] and Talos [6], Pilatus largely reduces the decryption time by optimizing the partial homomorphic encryption algorithm with Chinese Remainder Theorem. The previous work on CryptDB [5], [6], [6], [7] targets SQL query processing, whereas this work considers general IoT applications.

There exist other work [25]–[34] that propose secure cloud storage based on user-side encryption. Since data in the cloud are encrypted by users, the cloud cannot read the content of the data without decryption keys. Under the assumption of the ‘honest but curious’ cloud server model, these work let users control their private data through their own key management schemes. Commercial software solutions [22]–[24] are also available for seamless data encryption on the broker using symmetric-key encryption. However, they disregard data processing in the cloud and mainly support cloud storage services only, limiting their applicability.

Confidential cloud computing: Cloud providers offer confidential cloud computing [43]–[47] based on trusted execution environments (TEEs) such as Intel SGX [48]. Using TEEs, confidential cloud computing protects privacy data from privileged accesses of malicious insiders, host OSes, and hypervisors. Whereas confidential cloud computing requires special security hardware modules, the SelectiveCrypt framework enables private function execution using encryption without any security hardware module.
Information flow analysis: Information flow analysis is to detect security violation when sensitive data propagate through program execution. JFlow [49] is a language extension that permits static checking of information flow annotations for privacy preserving. Extending JFlow [50], Liantian et al. [51] apply the language-based techniques for distributed systems. Their work enables secure program partitioning when unreliable hosts exist by replicating computation and data. Other work [52] proposes a compiler that automatically partitions a program with locality annotations into distributed programs. To protect variable updates among distributed programs, the compiler inserts cryptographic operations. With information flow analysis, this work can extend the SelectiveCrypt compiler to perform more precise information flow analysis of privacy data by introducing extra annotations at the language level.

General-purpose compilers for privacy-preserving computation using HE: Previous work [53]–[56] proposes general-purpose compilers for privacy-preserving computation with various optimizations using different HE schemes. Cingulata [53] compiles C++ programs to BFV [14] and TFHE [57] scheme-based privacy-preserving implementation. ALCHEMY [54] compiles DSL on Haskell using BGV [13] scheme-based Haskell library as a backend. ALCHEMY selects proper parameters for HE library using its statically typed system. RAMPARTS [55] compiles a Julia program into PALISADE [21]-based implementation applying common optimizations such as sub-expression elimination and constant folding. EVA [56] proposes its own intermediate representation and compiles a program into SEAL [18]-based implementation. EVA optimizes arithmetic circuits based on graph rewriting rules, finds optimal parameters, and automatically inserts HE maintenance operations like relinearization and rescaling. The existing compilers proposed in previous work, however, are not aware of domain knowledge to selectively apply different cryptographic schemes depending on data usage patterns.

VII. CONCLUSION

This work proposes a new compiler-assisted semantic-aware encryption scheme named SelectiveCrypt. SelectiveCrypt applies different cryptographic primitives to each data item depending on their operations, thus allowing efficient and secure computation in FaaS services. By offloading encryption and decryption from a lightweight IoT device to a powerful trusted proxy, SelectiveCrypt can further reduce the encryption overheads. The evaluation with five benchmarks shows that SelectiveCrypt successfully reduces computation and communication overheads compared with a homomorphic encryption only scheme.

ACKNOWLEDGMENT

We thank the anonymous reviewers and editors for their insightful comments and suggestions. We also thank the Core-Lab for their support and feedback during this work. This work is supported by IITP-2017-0-00195, IITP-2018-0-01392 and IITP-2020-0-01847 through the Institute of Information and Communication Technology Planning and Evaluation (IITP) funded by the Ministry of Science and ICT. (Corresponding author: Hanjun Kim)

REFERENCES


**Jaeho Lee** is a Ph.D. student in the Department of Electrical and Electronic Engineering at Yonsei University. His research focuses on system security for storage devices and high-level synthesis optimization. He received his B.S. in electrical and electronic engineering from Yonsei University in 2020. Contact him at ejaho0103@yonsei.ac.kr.

**Shinnung Jeong** is a Ph.D. student in the Department of Electrical and Electronic Engineering at Yonsei University. She received his B.S. in Creative IT Engineering from Pohang University of Science and Technology (POSTECH) in 2019. Her research interests are compiler framework for many-core neural processing unit and parallel programming in compiler. Contact her at shin0403@yonsei.ac.kr.

**Yongwoo Lee** is a Ph.D. student in the Department of Electrical and Electronic Engineering at Yonsei University. He received his B.S. in computer science and engineering from POSTECH in 2019. His research focuses on compiler and runtime systems for homomorphic encryption. Contact him at drgonrain96@yonsei.ac.kr.

**Hanjun Kim** is an associate professor in the Department of Electrical and Electronic Engineering at Yonsei University. He received his B.S. in electrical engineering from Seoul National University in 2007, and his M.A. and Ph.D. in computer science from Princeton University in 2009 and 2013. From 2013 to 2018, he was an assistant and associate professor at the Departments of Creative IT Engineering and Computer Science and Engineering at POSTECH. He was awarded the Intel Corporation Ph.D. Fellowship and the Siebel Scholarship at 2012. His research interests include computer architecture and compiler optimization for distributed and emerging systems. Contact him at hanjun@yonsei.ac.kr.