







Oblivious Online Monitoring for Safety LTL Specification via Fully Homomorphic Encryption

Ryotaro Banno¹, Kotaro Matsuoka¹, Naoki Matsumoto¹, Song Bian²,
Masaki Waga¹, and Kohei Suenaga¹

¹ Kyoto University, Kyoto, Japan

² Beihang University, Beijing, China

Abstract. In many Internet of Things (IoT) applications, data sensed by an IoT device are continuously sent to the server and monitored against a specification. Since the data often contain sensitive information, and the monitored specification is usually proprietary, both must be kept private from the other end. We propose a protocol to conduct *oblivious online monitoring*—online monitoring conducted without revealing the private information of each party to the other—against a safety LTL specification. In our protocol, we first convert a safety LTL formula into a DFA and conduct online monitoring with the DFA. Based on *fully homomorphic encryption (FHE)*, we propose two online algorithms (REVERSE and BLOCK) to run a DFA obliviously. We prove the correctness and security of our entire protocol. We also show the scalability of our algorithms theoretically and empirically. Our case study shows that our algorithms are fast enough to monitor blood glucose levels online, demonstrating our protocol’s practical relevance.

1 Introduction

Internet of Things (IoT) [5] devices enable various service providers to monitor personal data of their users and to provide useful feedback to the users. For example, a smart home system can save lives by raising an alarm when a gas stove is left on to prevent a fire. Such a system is realized by the continuous monitoring of the data from the IoT devices in the house [9, 22]. Another application of IoT devices is medical IoT (MIoT) [19]. In MIoT applications, biological information, such as electrocardiograms or blood glucose levels, is monitored, and the user is notified when an abnormality is detected (such as arrhythmia or hyperglycemia).

In many IoT applications [11], monitoring must be conducted *online*, i.e., a stream of sensed data is continuously monitored, and the violation of the monitoring specification must be reported even before the entire data are obtained. In the smart home and MIoT applications, online monitoring is usually required, as continuous sensing is crucial for the immediate notifications to emergency responders, such as police officers or doctors, for the ongoing abnormal situations.

As specifications generally contain proprietary information or sensitive parameters learned from private data (e.g., with specification mining [33]), *the specifications must be kept secret*. One of the approaches for this privacy is to

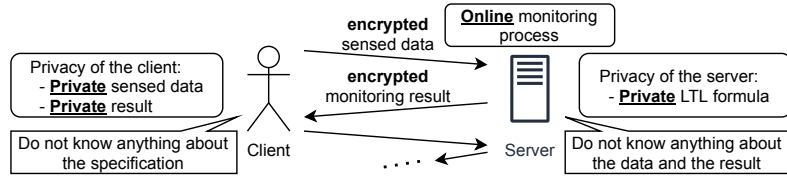
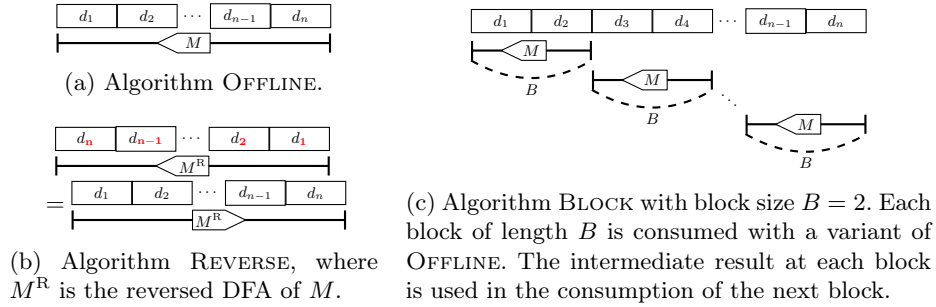


Fig. 1: The proposed oblivious online LTL monitoring protocol.

Fig. 2: How our algorithms consume the data d_1, d_2, \dots, d_n with the DFA M .

adopt the client-server model to the monitoring system. In such a model, the sensing device sends the collected data to a server, where the server performs the necessary analyses and returns the results to the device. Since the client does not have access to the specification, the server’s privacy is preserved.

However, the client-server model does *not* inherently protect the client’s privacy from the servers, as the data collected from and results sent back to the users are revealed to the servers in this model; that is to say, a user has to *trust* the server. This trust is problematic if, for example, the server itself intentionally or unintentionally leaks sensitive data of device users to an unauthorized party. Thus, we argue that a monitoring procedure should achieve the following goals:

Online monitoring. The monitored data need not be known beforehand.

Client’s Privacy. The server shall not know the monitored data and results.

Server’s Privacy. The client shall not know what property is monitored.

We call a monitoring scheme with these properties *oblivious online monitoring*. By an oblivious online monitoring procedure, 1) a user can get a monitoring result hiding her sensitive data and the result itself from a server, and 2) a server can conduct online monitoring hiding the specification from the user.

Contribution. In this paper, we propose a novel protocol (Fig. 1) for oblivious online monitoring against a specification in *linear temporal logic (LTL)* [40]. More precisely, we use a *safety LTL formula* [32] as a specification, which can be translated to a deterministic finite automaton (DFA) [43]. In our protocol, we first convert a safety LTL formula into a DFA and conduct online monitoring with

the DFA. For online and oblivious execution of a DFA, we propose two algorithms based on *fully homomorphic encryption* (FHE). FHE allows us to evaluate an arbitrary function over ciphertexts, and there is an FHE-based algorithm to evaluate a DFA obliviously [16]. However, this algorithm is limited to *leveled* homomorphic, i.e., the FHE parameters are dependent on the number of the monitored ciphertexts and thus not applicable to online monitoring.

In this work, we first present a *fully* homomorphic *offline* DFA evaluation algorithm (OFFLINE) by extending the leveled homomorphic algorithm in [16]. Although we can remove the parameter dependence using this method, OFFLINE consumes the ciphertexts from back to front (Fig. 2a). As a result, OFFLINE is still limited to offline usage only. To truly enable online monitoring, we propose two new algorithms based on OFFLINE: REVERSE and BLOCK. In REVERSE, we *reverse* the DFA and apply OFFLINE to the reversed DFA (Fig. 2b). In BLOCK, we split the monitored ciphertexts into fixed-length *blocks* and process each block sequentially with OFFLINE (Fig. 2c). We prove that both of the algorithms have *linear* time complexity and *constant* space complexity to the length of the monitored ciphertexts, which guarantees the scalability of our entire protocol.

On top of our online algorithms, we propose a protocol for oblivious online LTL monitoring. We assume that the client is *malicious*, i.e., the client can deviate arbitrarily from the protocol, while the server is *honest-but-curious*, i.e., the server honestly follows the protocol but tries to learn the client’s private data by exploiting the obtained information. We show that the privacy of both parties can be protected under the standard IND-CPA security of FHE schemes with the addition of *shielded randomness leakage* (SRL) security [12, 25].

We implemented our algorithms for DFA evaluation in C++20 and evaluated their performance. Our experiment results confirm the scalability of our algorithms. Moreover, through a case study on blood glucose levels monitoring, we also show that our algorithms run fast enough for online monitoring, i.e., our algorithms are faster than the sampling interval of the current commercial devices that samples glucose levels.

Our contributions are summarized as follows:

- We propose two *online* algorithms to run a DFA obliviously.
- We propose the first protocol for oblivious online LTL monitoring.
- We proved the correctness and security of our protocol.
- Our experiments show the scalability and practicality of our algorithms.

Related work. There are various works on DFA execution without revealing the monitored data (See Table 1 for a summary). However, to our knowledge, there is no existing work achieving all of our three goals (i.e., *online monitoring*, *privacy of the client*, and *privacy of the server*) simultaneously. Therefore, none of them is applicable to oblivious online LTL monitoring.

Homomorphic encryption, which we also utilize, has been used to run a DFA obliviously [16, 30]. Among different homomorphic encryption schemes, our algorithm is based on the algorithm in [16]. Although these algorithms guarantee the *privacy of the client* and the *privacy of the server*, all of the homomorphic-encryption-based algorithms are limited to offline DFA execution and do not

Table 1: Related work on DFA execution with *privacy of the client*.

Work	[44]	[24]	[10]	[42]	[38]	[26]	[30]	[16]	[1]	Ours
Support online monitoring	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Private the client's monitored data	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Private DFA, except for its number of the states	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Private DFA's number of the states	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓
Performance report	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓

achieve *online monitoring*. We note that the extension of [16] for online DFA execution is one of our technical contributions.

In [1], the authors propose an LTL runtime verification algorithm without revealing the monitored data to the server. They propose both offline and online algorithms to run a DFA converted from a safety LTL formula. The main issue with their online algorithm is that the DFA running on the server must be revealed to the client, and the goal of *privacy of the server* is not satisfied.

Oblivious DFA evaluation (ODFA) [10, 24, 26, 38, 42, 44] is a technique to run a DFA on a server while keeping the DFA secret to the server and the monitored data secret to the client. Although the structure of the DFA is not revealed to the client, the client has to know the number of the states. Consequently, the goal *privacy of the server* is satisfied *only partially*. Moreover, to the best of our knowledge, none of the ODFA-based algorithms support online DFA execution. Therefore, the goal *online monitoring* is not satisfied.

Organization. The rest of the paper is organized as follows: In Section 2, we overview LTL monitoring (Section 2.1), the FHE scheme we use (Section 2.2), and the leveled homomorphic offline algorithm (Section 2.3). Then, in Section 3, we explain our fully homomorphic offline algorithm (OFFLINE) and two online algorithms (REVERSE and BLOCK). We describe the proposed protocol for oblivious online LTL monitoring in Section 4. After we discuss our experimental results in Section 5, we conclude our paper in Section 6.

2 Preliminaries

Notations. We denote the set of all nonnegative integers by \mathbb{N} , the set of all positive integers by \mathbb{N}^+ , and the set $\{0, 1\}$ by \mathbb{B} . Let X be a set. We write 2^X for the powerset of X . We write X^* for the set of finite sequences of X elements and X^ω for the set of infinite sequences of X elements. For $u \in X^\omega$, we write $u_i \in X$ for the i -th element (0-based) of u , $u_{i:j} \in X^*$ for the subsequence u_i, u_{i+1}, \dots, u_j of u , and $u_{i:} \in X^\omega$ for the suffix of u starting from its i -th element. For $u \in X^*$ and $v \in X^* \cup X^\omega$, we write $u \cdot v$ for the concatenation of u and v .

DFA. A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of alphabet, $\delta: Q \times \Sigma \rightarrow Q$ is a transition function, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. If the alphabet of a DFA is \mathbb{B} , we call it a *binary DFA*. For a state $q \in Q$ and a word $w = \sigma_1 \sigma_2 \dots \sigma_n$ we define $\delta(q, w) := \delta(\dots \delta(\delta(q, \sigma_1), \sigma_2), \dots, \sigma_n)$. For a

DFA M and a word w , we write $M(w) := 1$ if M accepts w ; otherwise, $M(w) := 0$. We also abuse the above notations for nondeterministic finite automata (NFAs).

2.1 LTL

We use *linear temporal logic (LTL)* [40] to specify the monitored properties. The following BNF defines the syntax of LTL formulae: $\phi, \psi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \psi \mid X\phi \mid \phi U\psi$, where ϕ and ψ range over LTL formulae and p ranges over a set AP of atomic propositions.

An LTL formula asserts a property of $u \in (2^{\text{AP}})^\omega$. The sequence u expresses an execution trace of a system; u_i is the set of the atomic propositions satisfied at the i -th time step. Intuitively, \top represents an always-true proposition; p asserts that u_0 contains p , and hence p holds at the 0-th step in u ; $\neg\phi$ is the negation of ϕ ; and $\phi \wedge \psi$ is the conjunction of ϕ and ψ . The temporal proposition $X\phi$ expresses that ϕ holds from the next step (i.e., u_1); $\phi U\psi$ expresses that ψ holds eventually and ϕ continues to hold until then. We write \perp for $\neg\top$; $\phi \vee \psi$ for $\neg(\neg\phi \wedge \neg\psi)$;

$\phi \implies \psi$ for $\neg\phi \vee \psi$; $F\phi$ for $\top U\phi$; $G\phi$ for $\neg(F\neg\phi)$; $G_{[n,m]}\phi$ for $\overbrace{X \dots X}^{n \text{ occurrences of } X} (\phi \wedge \overbrace{X \dots X}^{(m-n) \text{ occ. of } X})$ ($\phi \wedge \overbrace{X(\phi \wedge X(\dots \wedge X\phi))}^{(m-n) \text{ occ. of } X}$); and $F_{[n,m]}\phi$ for $\overbrace{X \dots X}^{n \text{ occ. of } X} (\phi \vee \overbrace{X(\phi \vee X(\dots \vee X\phi))}^{(m-n) \text{ occ. of } X})$.

We formally define the semantics of LTL below. Let $u \in (2^{\text{AP}})^\omega$, $i \in \mathbb{N}$, and ϕ be an LTL formula. We define the relation $u, i \models \phi$ as the least relation that satisfies the following:

$$\begin{aligned} u, i \models \top & \quad u, i \models p \stackrel{\text{def}}{\iff} p \in u(i) & \quad u, i \models \neg\phi \stackrel{\text{def}}{\iff} u, i \not\models \phi \\ u, i \models \phi \wedge \psi & \stackrel{\text{def}}{\iff} u, i \models \phi \text{ and } u, i \models \psi & \quad u, i \models X\phi \stackrel{\text{def}}{\iff} u, i+1 \models \phi \\ u, i \models \phi U\psi & \stackrel{\text{def}}{\iff} \text{there exists } j \geq i \text{ such that } u, j \models \psi \text{ and,} \\ & \text{for any } k, i \leq k \leq j \implies u, k \models \phi. \end{aligned}$$

We write $u \models \phi$ for $u, 0 \models \phi$ and say u *satisfies* ϕ .

In this paper, we focus on *safety* [32] (i.e., nothing bad happens) fragment of LTL properties. A finite sequence $w \in (2^{\text{AP}})^*$ is a *bad prefix* for an LTL formula ϕ if $w \cdot v \not\models \phi$ holds for any $v \in (2^{\text{AP}})^\omega$. For any bad prefix w , we cannot extend w to an infinite word that satisfies ϕ . An LTL formula ϕ is a *safety* LTL formula if for any $w \in (2^{\text{AP}})^\omega$ satisfying $w \not\models \phi$, w has a bad prefix for ϕ .

A *safety monitor* (or simply a *monitor*) is a procedure that takes $w \in (2^{\text{AP}})^\omega$ and a safety LTL formula ϕ and generates an alert if $w \not\models \phi$. From the definition of safety LTL, it suffices for a monitor to detect a bad prefix of ϕ . It is known that, for any safety LTL formula ϕ , we can construct a DFA M_ϕ recognizing the set of the bad prefixes of ϕ [43], which can be used as a monitor.

2.2 Torus Fully Homomorphic Encryption

Homomorphic encryption (HE) is a form of encryption that enables us to apply operations to encrypted values *without decrypting them*. In particular, a type

Table 2: Summary of TFHE ciphertexts, where N is a parameter of TFHE.

Ciphertext Kind	Notation in this paper	Plaintext Message	Conversion from TRLWE
TLWE	c	a Boolean value $b \in \mathbb{B}$	SAMPLEEXTRACT (fast)
TRLWE	\mathbf{c}	a Boolean vector $v \in \mathbb{B}^N$	—————
TRGSW	d	a Boolean value $b \in \mathbb{B}$	SAMPLEEXTRACT and CIRCUITBOOTSTRAPPING (slow)

of HE, called Fully HE (FHE), allows us to evaluate arbitrary functions over encrypted data [13, 23, 27, 28]. We use an instance of FHE called TFHE [16] in this work. We briefly summarize TFHE below; see [16] for a detailed exposition.

We are concerned with the following two-party secure computation, where the involved parties are a client (called Alice) and a server (called Bob): 1) Alice generates the keys used during computation; 2) Alice encrypts her plaintext messages into ciphertexts with her keys; 3) Alice sends the ciphertexts to Bob; 4) Bob conducts computation over the received ciphertexts and obtains the encrypted result *without decryption*; 5) Bob sends the encrypted results to Alice; 6) Alice decrypts the received results and obtains the results in plaintext.

Keys. There are three types of keys in TFHE: *secret key* SK, *public key* PK, and *bootstrapping key* BK. All of them are generated by Alice. PK is used to encrypt plaintext messages into ciphertexts, and SK is used to decrypt ciphertexts into plaintexts. Alice keeps SK private, i.e., the key is known only to herself but not to Bob. In contrast, PK is public and also known to Bob. BK is generated from SK and can be safely shared with Bob without revealing SK. BK allows Bob to evaluate the homomorphic operations (defined later) over the ciphertext.

Ciphertexts. Using the public key, Alice can generate three kinds of ciphertexts (Table 2): TLWE (Torus Learning With Errors), TRLWE (Torus Ring Learning With Errors), and TRGSW (Torus Ring Gentry-Sahai-Waters). Homomorphic operations provided by TFHE are defined over each of the specific ciphertexts. We note that different ciphertexts have different data structures, and their conversion can be time-consuming. Table 2 shows one such example.

In TFHE, different types of ciphertexts represent different plaintext messages. A TLWE ciphertext represents a Boolean value. In contrast, TRLWE represents a vector of Boolean values of length N , where N is a TFHE parameter. We can regard a TRLWE ciphertext as a vector of TLWE ciphertexts, and the conversion between a TRLWE ciphertext and a TLWE one is relatively easy. A TRGSW ciphertext also represents a Boolean value, but its data structure is quite different from TLWE, and the conversion from TLWE to TRGSW is slow.

TFHE provides different encryption and decryption functions for each type of ciphertext. We write $\text{Enc}(x)$ for a ciphertext of a plaintext x ; $\text{Dec}(c)$ for the plaintext message for the ciphertext c . We abuse these notations for all three types of ciphertexts.

Besides, TFHE supports *trivial samples* of TRLWE. A trivial sample of TRLWE has the same data structure as a TRLWE ciphertext but is *not* encrypted, i.e., anyone can tell the plaintext message represented by the trivial sample. We denote by $\text{TRIVIAL}(n)$ a trivial sample of TRLWE whose plaintext message is (b_1, b_2, \dots, b_N) , where each b_i is the i -th bit in the binary representation of n , i.e., $n = \sum_{i=1}^N b_i 2^{i-1}$.

Homomorphic Operations. TFHE provides *homomorphic operations*, i.e., operations over ciphertexts without decryption. Among the operators supported by TFHE [16], we use the following ones.

CMUX $(d, \mathbf{c}_{\text{true}}, \mathbf{c}_{\text{false}}) : \text{TRGSW} \times \text{TRLWE} \times \text{TRLWE} \rightarrow \text{TRLWE}$

Given a TRGSW ciphertext d and TRLWE ciphertexts $\mathbf{c}_{\text{true}}, \mathbf{c}_{\text{false}}$, CMUX outputs a TRLWE ciphertext $\mathbf{c}_{\text{result}}$ such that $\text{Dec}(\mathbf{c}_{\text{result}}) = \text{Dec}(\mathbf{c}_{\text{true}})$ if $\text{Dec}(d) = 1$, and otherwise, $\text{Dec}(\mathbf{c}_{\text{result}}) = \text{Dec}(\mathbf{c}_{\text{false}})$.

LOOKUP $(\{\mathbf{c}_i\}_{i=1}^{2^n}, \{d_i\}_{i=1}^n) : (\text{TRLWE})^{2^n} \times (\text{TRGSW})^n \rightarrow \text{TRLWE}$

Given TRLWE ciphertexts $\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_{2^n}$ and TRGSW ciphertexts d_1, d_2, \dots, d_n , LOOKUP outputs a TRLWE ciphertext \mathbf{c} such that $\text{Dec}(\mathbf{c}) = \text{Dec}(\mathbf{c}_k)$ and $k = \sum_{i=1}^n 2^{i-1} \times \text{Dec}(d_i)$.

SAMPLEEXTRACT $(k, \mathbf{c}) : \mathbb{N} \times \text{TRLWE} \rightarrow \text{TLWE}$

Let $\text{Dec}(\mathbf{c}) = (b_1, b_2, \dots, b_N)$. Given $k < N$ and a TRLWE ciphertext \mathbf{c} , SAMPLEEXTRACT outputs a TLWE ciphertext c where $\text{Dec}(c) = b_{k+1}$.

Intuitively, CMUX can be regarded as a multiplexer over TRLWE ciphertexts with TRGSW selector input. The operation LOOKUP regards $\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_{2^n}$ as encrypted entries composing a LookUp Table (LUT) of depth n and d_1, d_2, \dots, d_n as inputs to the LUT. Its output is the entry selected by the LUT. LOOKUP is constructed by $2^n - 1$ CMUX arranged in a tree of depth n . SAMPLEEXTRACT outputs the k -th element of \mathbf{c} as TLWE. Notice that all these operations work over ciphertexts without decrypting them.

Noise and Operations for Noise Reduction. In generating a TFHE ciphertext, we ensure its security by adding some random numbers called *noise*. An application of a TFHE operation adds noise to its output ciphertext; if the noise in a ciphertext becomes too large, the TFHE ciphertext cannot be correctly decrypted. There is a special type of operation called *bootstrapping*³ [27], which reduces the noise of a TFHE ciphertext.

BOOTSTRAPPING_{BK} $(c) : \text{TLWE} \rightarrow \text{TRLWE}$

Given a bootstrapping key BK and a TLWE ciphertext c , BOOTSTRAPPING outputs a TRLWE ciphertext \mathbf{c} where $\text{Dec}(\mathbf{c}) = (b_1, b_2, \dots, b_N)$ and $b_1 = \text{Dec}(c)$. Moreover, the noise of \mathbf{c} becomes a constant that is determined by the parameters of TFHE and is independent of c .

CIRCUITBOOTSTRAPPING_{BK} $(c) : \text{TLWE} \rightarrow \text{TRGSW}$

Given a bootstrapping key BK and a TLWE ciphertext c , CIRCUITBOOTSTRAPPING outputs a TRGSW ciphertext d where $\text{Dec}(d) = \text{Dec}(c)$. The

³ Note that bootstrapping here has nothing to do with bootstrapping in statistics.

Algorithm 1: The leveled homomorphic offline algorithm [16].

```

Input    : A binary DFA  $M = (Q, \Sigma = \mathbb{B}, \delta, q_0, F)$  and TRGSW monitored ciphertexts
               $d_1, d_2, \dots, d_n$ 
Output  : A TLWE ciphertext  $c$  satisfying  $\text{Dec}(c) = M(\text{Dec}(d_1)\text{Dec}(d_2) \dots \text{Dec}(d_n))$ 
1 for  $q \in Q$  do
2   |  $\mathbf{c}_{n,q} \leftarrow q \in F ? \text{TRIVIAL}(1) : \text{TRIVIAL}(0)$  // Initialize each  $\mathbf{c}_{n,q}$ 
3 for  $i = n, n-1, \dots, 1$  do
4   | for  $q \in Q$  such that  $q$  is reachable from  $q_0$  by  $(i-1)$  transitions do
5   |   |  $\mathbf{c}_{i-1,q} \leftarrow \text{CMUX}(d_i, \mathbf{c}_{i,\delta(q,1)}, \mathbf{c}_{i,\delta(q,0)})$ 
6  $c \leftarrow \text{SAMPLEEXTRACT}(0, \mathbf{c}_{0,q_0})$ 
7 return  $c$ 

```

noise of d becomes a constant that is determined by the parameters of TFHE and is independent of c .

These bootstrapping operations are used to keep the noise of a TFHE ciphertext small enough to be correctly decrypted. BOOTSTRAPPING and CIRCUITBOOTSTRAPPING are almost two and three orders of magnitude slower than CMUX, respectively [16].

Parameters for TFHE. There are many parameters for TFHE, such as the length N of the message of a TRLWE ciphertext and the standard deviation of the probability distribution from which a noise is taken. Certain properties of TFHE depend on these parameters. These properties include the security level of TFHE, the number of TFHE operations that can be applied without bootstrapping ensuring correct decryption, and the time and the space complexity of each operation. The complete list of TFHE parameters is presented in Appendix B.

We remark that we need to determine the TFHE parameters *before* performing any TFHE operation. Therefore, we need to know the number of applications of homomorphic operations without bootstrapping *in advance*, i.e., the homomorphic circuit depth must be determined *a priori*.

2.3 Leveled Homomorphic Offline Algorithm

Chillotti et al. [16] proposed an *offline* algorithm to evaluate a DFA over TFHE ciphertexts (Algorithm 1). Given a DFA M and TRGSW ciphertexts d_1, d_2, \dots, d_n , Algorithm 1 returns a TLWE ciphertext c satisfying $\text{Dec}(c) = M(\text{Dec}(d_1)\text{Dec}(d_2) \dots \text{Dec}(d_n))$. For simplicity, for a state q of M , we write $M^i(q)$ for $M(q, \text{Dec}(d_i)\text{Dec}(d_{i+1}) \dots \text{Dec}(d_n))$.

In Algorithm 1, we use a TRLWE ciphertext $\mathbf{c}_{i,q}$ whose first element represents $M^{i+1}(q)$, i.e., whether we reach a final state by reading $\text{Dec}(d_{i+1})\text{Dec}(d_{i+2}) \dots \text{Dec}(d_n)$ from q . We abuse this notation for $i = n$, i.e., the first element of $\mathbf{c}_{n,q}$ represents if $q \in F$. In Lines 1 and 2, we initialize $\mathbf{c}_{n,q}$; For each $q \in Q$, we let $\mathbf{c}_{n,q}$ be TRIVIAL(1) if $q \in F$; otherwise, we let $\mathbf{c}_{n,q}$ be TRIVIAL(0). In Lines 3–5, we construct $\mathbf{c}_{i-1,q}$ inductively by feeding each monitored ciphertext d_i to CMUX from tail to head. Here, $\mathbf{c}_{i-1,q}$ represents $M^i(q)$ because of $M^i(q) =$

$M^{i+1}(\delta(q, \text{Dec}(d_i)))$. We note that for the efficiency, we only construct $\mathbf{c}_{i-1,q}$ for the states reachable from q_0 by $i - 1$ transitions. In Line 6, we extract the first element of \mathbf{c}_{0,q_0} , which represents $M^1(q_0)$, i.e., $M(\text{Dec}(d_1)\text{Dec}(d_2)\dots\text{Dec}(d_n))$.

Theorem 1 (Correctness [16, Thm. 5.4]). *Given a binary DFA M and TRGSW ciphertexts d_1, d_2, \dots, d_n , if c in Algorithm 1 can be correctly decrypted, Algorithm 1 outputs c satisfying $\text{Dec}(c) = M(\text{Dec}(d_1)\text{Dec}(d_2)\dots\text{Dec}(d_n))$. \square*

Complexity Analysis. The time complexity of Algorithm 1 is determined by the number of applications of CMUX, which is $O(n|Q|)$. Its space complexity is $O(|Q|)$ because we can use two sets of $|Q|$ TRLWE ciphertexts alternately for $\mathbf{c}_{2j-1,q}$ and $\mathbf{c}_{2j,q}$ (for $j \in \mathbb{N}^+$).

Shortcomings of Algorithm 1. We cannot use Algorithm 1 under an *online* setting due to two reasons. Firstly, Algorithm 1 is a *leveled* homomorphic algorithm, i.e., the maximum length of the ciphertexts that Algorithm 1 can handle is determined by TFHE parameters. This is because Algorithm 1 does not use BOOTSTRAPPING, and if the monitored ciphertexts are too long, the result c cannot be correctly decrypted due to the noise. This is critical in an online setting because we do not know the length n of the monitored ciphertexts in advance, and we cannot determine such parameters appropriately.

Secondly, Algorithm 1 consumes the monitored ciphertext from back to front, i.e., the last ciphertext d_n is used in the beginning, and d_1 is used in the end. Thus, we cannot start Algorithm 1 before the last input is given.

3 Online Algorithms for Running DFA Obliviously

In this section, we propose two online algorithms that run a DFA obliviously. As a preparation for these online algorithms, we also introduce a fully homomorphic offline algorithm based on Algorithm 1.

3.1 Preparation: Fully Homomorphic Offline Algorithm (OFFLINE)

As preparation for introducing an algorithm that can run a DFA under an online setting, we enhance Algorithm 1 so that we can monitor a sequence of ciphertexts whose length is unknown *a priori*. Algorithm 2 shows our *fully homomorphic* offline algorithm (OFFLINE), which does not require TFHE parameters to depend on the length of the monitored ciphertexts. The key difference lies in Lines 6–9 (the red lines) of Algorithm 2. Here, for every I_{boot} consumption of the monitored ciphertexts, we reduce the noise by applying BOOTSTRAPPING to the ciphertext $\mathbf{c}_{i,j}$ representing a state of the DFA. Since the amount of the noise accumulated in $\mathbf{c}_{i,j}$ is determined only by the number of the processed ciphertexts, we can keep the noise levels of $\mathbf{c}_{i,j}$ low and ensure that the monitoring result c is correctly decrypted. Therefore, by using Algorithm 2, we can monitor an arbitrarily long sequence of ciphertexts as long as the interval I_{boot} is properly chosen according to the TFHE parameters. We note that we still cannot use Algorithm 2 for online monitoring because it consumes the monitored ciphertexts from back to front.

Algorithm 2: Our fully homomorphic offline algorithm (OFFLINE).

```

Input   : A binary DFA  $M = (Q, \Sigma = \mathbb{B}, \delta, q_0, F)$ , TRGSW monitored ciphertexts
             $d_1, d_2, \dots, d_n$ , a bootstrapping key BK, and  $I_{\text{boot}} \in \mathbb{N}^+$ 
Output : A TLWE ciphertext  $c$  satisfying  $\text{Dec}(c) = M(\text{Dec}(d_1)\text{Dec}(d_2) \dots \text{Dec}(d_n))$ 
1 for  $q \in Q$  do
2    $\mathbf{c}_{n,q} \leftarrow q \in F ? \text{TRIVIAL}(1) : \text{TRIVIAL}(0)$ 
3 for  $i = n, n-1, \dots, 1$  do
4   for  $q \in Q$  such that  $q$  is reachable from  $q_0$  by  $(i-1)$  transitions do
5      $\mathbf{c}_{i-1,q} \leftarrow \text{CMUX}(d_i, \mathbf{c}_{i,\delta(q,1)}, \mathbf{c}_{i,\delta(q,0)})$ 
6     if  $(n-i+1) \bmod I_{\text{boot}} = 0$  then
7       for  $q \in Q$  such that reachable from  $q_0$  by  $(i-1)$  transitions do
8          $\mathbf{c}_{i-1,q} \leftarrow \text{SAMPLEEXTRACT}(0, \mathbf{c}_{i-1,q})$ 
9          $\mathbf{c}_{i-1,q} \leftarrow \text{BOOTSTRAPPINGBK}(\mathbf{c}_{i-1,q})$ 
10   $c \leftarrow \text{SAMPLEEXTRACT}(0, \mathbf{c}_{0,q_0})$ 
11 return  $c$ 

```

Algorithm 3: Our first online algorithm (REVERSE).

```

Input   : A binary DFA  $M$ , TRGSW monitored ciphertexts  $d_1, d_2, d_3, \dots, d_n$ , a
            bootstrapping key BK, and  $I_{\text{boot}} \in \mathbb{N}^+$ 
Output : For every  $i \in \{1, 2, \dots, n\}$ , a TLWE ciphertext  $c_i$  satisfying
             $\text{Dec}(c_i) = M(\text{Dec}(d_1)\text{Dec}(d_2) \dots \text{Dec}(d_i))$ 
1 let  $M^R = (Q^R, \mathbb{B}, \delta^R, q_0^R, F^R)$  be the minimum reversed DFA of  $M$ 
2 for  $q^R \in Q^R$  do
3    $\mathbf{c}_{0,q^R} \leftarrow q^R \in F^R ? \text{TRIVIAL}(1) : \text{TRIVIAL}(0)$ 
4 for  $i = 1, 2, \dots, n$  do
5   for  $q^R \in Q^R$  do
6      $\mathbf{c}_{i,q^R} \leftarrow \text{CMUX}(d_i, \mathbf{c}_{i-1,\delta^R(q^R,1)}, \mathbf{c}_{i-1,\delta^R(q^R,0)})$ 
7     if  $i \bmod I_{\text{boot}} = 0$  then
8       for  $q^R \in Q^R$  do
9          $\mathbf{c}_{i,q^R} \leftarrow \text{SAMPLEEXTRACT}(0, \mathbf{c}_{i,q^R})$ 
10         $\mathbf{c}_{i,q^R} \leftarrow \text{BOOTSTRAPPINGBK}(\mathbf{c}_{i,q^R})$ 
11   $c_i \leftarrow \text{SAMPLEEXTRACT}(0, \mathbf{c}_{i,q_0^R})$ 
12 output  $c_i$ 

```

3.2 Online Algorithm 1: REVERSE

To run a DFA online, we modify OFFLINE so that the monitored ciphertexts are consumed from front to back. Our main idea is illustrated in Fig. 2b: we *reverse* the DFA M beforehand and feed the ciphertexts d_1, d_2, \dots, d_n to the reversed DFA M^R serially from d_1 to d_n .

Algorithm 3 shows the outline of our first online algorithm (REVERSE) based on the above idea. REVERSE takes the same inputs as OFFLINE: a DFA M , TRGSW ciphertexts d_1, d_2, \dots, d_n , a bootstrapping key BK, and a positive integer I_{boot} indicating the interval of bootstrapping. In Line 1, we construct the minimum DFA M^R that satisfies, for any $w = \sigma_1\sigma_2 \dots \sigma_k \in \mathbb{B}^*$, we have $M^R(w) = M(w^R)$, where $w^R = \sigma_k \dots \sigma_1$. We can construct such a DFA by reversing the transitions and by applying the powerset construction and the minimization algorithm [29].

In the loop from Lines 4–12, the reversed DFA M^R consumes each monitored ciphertext d_i , which corresponds to the loop from Lines 3–9 in Algorithm 2. The main difference lies in Lines 5 and 8: Algorithm 3 applies CMUX and BOOT-

Algorithm 4: Our second online algorithm (BLOCK).

```

Input   : A binary DFA  $M = (Q, \Sigma = \mathbb{B}, \delta, q_0, F)$ , TRGSW monitored ciphertexts
             $d_1, d_2, d_3, \dots, d_n$ , a bootstrapping key BK, and  $B \in \mathbb{N}^+$ 
Output : For every  $i \in \mathbb{N}^+$  ( $i \leq \lfloor n/B \rfloor$ ), a TLWE ciphertext  $c_i$  satisfying
             $\text{Dec}(c_i) = M(\text{Dec}(d_1)\text{Dec}(d_2) \dots \text{Dec}(d_{i \times B}))$ 
1  $S_1 \leftarrow \{q_0\}$  //  $S_i$ : the states reachable by  $(i-1) \times B$  transitions.
2 for  $i = 1, 2, \dots, \lfloor n/B \rfloor$  do
3    $S_{i+1} \leftarrow \{q \in Q \mid \exists s_i \in S_i. q \text{ is reachable from } s_i \text{ by } B \text{ transitions}\}$ 
   // We denote  $S_{i+1} = \{s_1^{i+1}, s_2^{i+1}, \dots, s_{|S_{i+1}|}^{i+1}\}$ 
4   for  $q \in Q$  do
5     if  $q \in S_{i+1}$  then
6        $j \leftarrow$  the index of  $S_{i+1}$  such that  $q = s_j^{i+1}$ 
7        $\mathbf{c}_{B,q}^{T_i} \leftarrow \text{TRIVIAL}((j-1) \times 2 + (q \in F ? 1 : 0))$ 
8     for  $k = B, B-1, \dots, 1$  do
9       for  $q \in Q$  such that  $q$  is reachable from a state in  $S_i$  by  $(k-1)$  transitions do
10       $\mathbf{c}_{k-1,q}^{T_i} \leftarrow \text{CMUX}(d_{(i-1)B+k}, \mathbf{c}_{k,\delta(q,1)}^{T_i}, \mathbf{c}_{k,\delta(q,0)}^{T_i})$ 
11    if  $|S_i| = 1$  then
12       $\mathbf{c}_{i+1}^{\text{cur}} \leftarrow \mathbf{c}_{0,q}^{T_i}$  where  $S_i = \{q\}$ 
13    else
14      for  $l = 1, 2, \dots, \lceil \log_2(|S_i|) \rceil$  do
15         $c_l \leftarrow \text{SAMPLEEXTRACT}(l, \mathbf{c}_i^{\text{cur}})$ 
16         $d'_l \leftarrow \text{CIRCUITBOOTSTRAPPINGBK}(c_l)$ 
17         $\mathbf{c}_{i+1}^{\text{cur}} \leftarrow \text{LOOKUP}(\{\mathbf{c}_{0,s_1^i}^{T_i}, \mathbf{c}_{0,s_2^i}^{T_i}, \dots, \mathbf{c}_{0,s_{|S_i|}^i}^{T_i}\}, \{d'_1, \dots, d'_{\lceil \log_2(|S_i|) \rceil}\})$ 
18     $c_i \leftarrow \text{SAMPLEEXTRACT}(0, \mathbf{c}_{i+1}^{\text{cur}})$ 
19  output  $c_i$ 

```

STRAPPING to all the states of M^R , while Algorithm 2 only considers the states reachable from the initial state. This is because in online monitoring, we monitor a stream of ciphertexts without knowing the number of the remaining ciphertexts, and all the states of the reversed DFA M^R are potentially reachable from the initial state q_0^R by the reversed remaining ciphertexts $d_n, d_{n-1}, \dots, d_{i+1}$ because of the minimality of M^R .

Theorem 2. *Given a binary DFA M , TRGSW ciphertexts d_1, d_2, \dots, d_n , a bootstrapping key BK, and a positive integer I_{boot} , for each $i \in \{1, 2, \dots, n\}$, if c_i in Algorithm 3 can be correctly decrypted, Algorithm 3 outputs c_i satisfying $\text{Dec}(c_i) = M(\text{Dec}(d_1)\text{Dec}(d_2) \dots \text{Dec}(d_i))$.*

Proof (sketch). SAMPLEEXTRACT and BOOTSTRAPPING in Lines 9 and 10 do not change the decrypted value of c_i . Therefore, $\text{Dec}(c_i) = M^R(\text{Dec}(d_i) \dots \text{Dec}(d_1))$ for $i \in \{1, 2, \dots, n\}$ by Theorem 1. As M^R is the reversed DFA of M , we have $\text{Dec}(c_i) = M^R(\text{Dec}(d_i) \dots \text{Dec}(d_1)) = M(\text{Dec}(d_1) \dots \text{Dec}(d_i))$. \square

3.3 Online Algorithm 2: BLOCK

A problem of REVERSE is that the number of the states of the reversed DFA can explode exponentially due to powerset construction (see Section 3.4 for the details). Another idea of an online algorithm without reversing a DFA is illustrated in Fig. 2c: we split the monitored ciphertexts into *blocks* of fixed size B

and process each block in the same way as Algorithm 2. Intuitively, for each block $d_{1+(i-1)\times B}, d_{2+(i-1)\times B}, \dots, d_{B+(i-1)\times B}$ of ciphertexts, we compute the function $T_i: Q \rightarrow Q$ satisfying $T_i(q) = \delta(q, d_{1+(i-1)\times B}, d_{2+(i-1)\times B}, \dots, d_{B+(i-1)\times B})$ by a variant of OFFLINE, and keep track of the current state of the DFA after reading the current prefix $d_1, d_2, \dots, d_{B+(i-1)\times B}$.

Algorithm 4 shows the outline of our second online algorithm (BLOCK) based on the above idea. Algorithm 4 takes a DFA M , TRGSW ciphertexts d_1, d_2, \dots, d_n , a bootstrapping key BK, and an integer B representing the interval of output. To simplify the presentation, we make the following assumptions, which are relaxed later: 1) B is small, and a trivial TRLWE sample can be correctly decrypted after B applications of CMUX; 2) the size $|Q|$ of the states of the DFA M is smaller than or equal to 2^{N-1} , where N is the length of TRLWE.

The main loop of the algorithm is sketched on Lines 2–19. In each iteration, we consume the i -th block consisting of B ciphertexts, i.e., $d_{(i-1)B+1}, \dots, d_{(i-1)B+B}$. In Line 3, we compute the set $S_{i+1} = \{s_1^{i+1}, s_2^{i+1}, \dots, s_{|S_{\text{next}}|}^{i+1}\}$ of the states reachable from q_0 by reading a word of length $i \times B$.

In Lines 4–10, for each $q \in Q$, we construct a ciphertext representing $T_i(q)$ by feeding the current block to a variant of OFFLINE. More precisely, we construct a ciphertext $\mathbf{c}_{0,q}^{T_i}$ representing the pair of the Boolean value showing if $T_i(q) \in F$ and the state $T_i(q) \in Q$. The encoding of such a pair in a TRLWE ciphertext is as follows: the first element shows if $T_i(q) \in F$ and the other elements are the binary representation of $j \in \mathbb{N}^+$, where j is such that $s_j^{i+1} = T_i(q)$.

In Lines 11–17, we construct the ciphertext $\mathbf{c}_{i+1}^{\text{cur}}$ representing the state of the DFA M after reading the current prefix $d_1, d_2, \dots, d_{B+(i-1)\times B}$. If $|S_i| = 1$, since the unique element q of S_i is the only possible state before consuming the current block, the state after reading it is $T(q)$. Therefore, we let $\mathbf{c}_{i+1}^{\text{cur}} = \mathbf{c}_{0,q}^{T_i}$.

Otherwise, we extract the ciphertext representing the state q before consuming the current block, and let $\mathbf{c}_{i+1}^{\text{cur}} = \mathbf{c}_{0,q}^{T_i}$. Since the $\mathbf{c}_i^{\text{cur}}$ (except for the first element) represents q (see Line 7), we extract them by applying SAMPLEEXTRACT (Line 15) and convert them to TRGSW by applying CIRCUITBOOTSTRAPPING (Line 16). Then, we choose $\mathbf{c}_{0,q}^{T_i}$ by applying LOOKUP and set it to $\mathbf{c}_{i+1}^{\text{cur}}$.

The output after consuming the current block, i.e., $M(\text{Dec}(d_1)\text{Dec}(d_2)\dots\text{Dec}(d_{(i-1)B+B}))$ is stored in the first element of the TRLWE ciphertext $\mathbf{c}_{i+1}^{\text{cur}}$. It is extracted by applying SAMPLEEXTRACT in Line 18 and output in Line 19.

Theorem 3. *Given a binary DFA M , TRGSW ciphertexts d_1, d_2, \dots, d_n , a bootstrapping key BK, and a positive integer B , for each $i \in \{1, 2, \dots, \lfloor n/B \rfloor\}$, if c_i in Algorithm 4 can be correctly decrypted, Algorithm 4 outputs a TLWE ciphertext c_i satisfying $\text{Dec}(c_i) = M(\text{Dec}(d_1)\text{Dec}(d_2)\dots\text{Dec}(d_{i \times B}))$.*

Proof (sketch). Let q^i be $\delta(q_0, \text{Dec}(d_1)\text{Dec}(d_2)\dots\text{Dec}(d_{i \times B}))$. It suffices to show that, for each iteration i in Line 2, $\text{Dec}(\mathbf{c}_{i+1}^{\text{cur}})$ represents a pair of the Boolean value showing if $q^i \in F$ and the state $q^i \in Q$ in the above encoding format. This is because c_i represents the first element of $\mathbf{c}_{i+1}^{\text{cur}}$. Algorithm 4 selects $\mathbf{c}_{i+1}^{\text{cur}}$ from $\{\mathbf{c}_{0,q}^{T_i}\}_{q \in S_i}$ in Line 12 or Line 17. By using a slight variant of Theorem 1 in

Table 3: Complexity of the proposed algorithms with respect to the number $|Q|$ of the states of the DFA and the size $|\phi|$ of the LTL formula. For BLOCK, we show the complexity *before* the relaxation.

Algorithm w.r.t.	Number of Applications				Space
	CMux	BOOTSTRAPPING	CIRCUITBOOTSTRAPPING	CIRCUITBOOTSTRAPPING	
OFFLINE	DFA	$O(n Q)$	$O(n Q /I_{\text{boot}})$	—	$O(Q)$
	LTL	$O(n2^{ \phi })$	$O(n2^{ \phi }/I_{\text{boot}})$	—	$O(2^{ \phi })$
REVERSE	DFA	$O(n2^{ \phi })$	$O(n2^{ \phi }/I_{\text{boot}})$	—	$O(2^{ \phi })$
	LTL	$O(n2^{ \phi })$	$O(n2^{ \phi }/I_{\text{boot}})$	—	$O(2^{ \phi })$
BLOCK	DFA	$O(n Q)$	—	$O((n \log Q)/B)$	$O(Q)$
	LTL	$O(n2^{ \phi })$	—	$O(n2^{ \phi }/B)$	$O(2^{ \phi })$

Lines 11–17, we can show that $\mathbf{c}_{0,q}^{T_i}$ represents if $T^i(q) \in F$ and the state $T^i(q)$. Therefore, the proof is completed by showing $\text{Dec}(\mathbf{c}_{i+1}^{\text{cur}}) = \text{Dec}(\mathbf{c}_{0,q^{i-1}}^{T_i})$.

We prove $\text{Dec}(\mathbf{c}_{i+1}^{\text{cur}}) = \text{Dec}(\mathbf{c}_{0,q^{i-1}}^{T_i})$ by induction on i . If $i = 1$, $|S_i| = 1$ holds, and by $q^{i-1} \in S_i$, we have $\text{Dec}(\mathbf{c}_{i+1}^{\text{cur}}) = \text{Dec}(\mathbf{c}_{0,q^{i-1}}^{T_i})$. If $i > 1$ and $|S_i| = 1$, $\text{Dec}(\mathbf{c}_{i+1}^{\text{cur}}) = \text{Dec}(\mathbf{c}_{0,q^{i-1}}^{T_i})$ holds similarly. If $i > 1$ and $|S_i| > 1$, by induction hypothesis, $\text{Dec}(\mathbf{c}_i^{\text{cur}})$ represents if $T_{i-1}(q^{i-2}) = q^{i-1} \in F$ and the state q^{i-1} . By construction in Line 16, $\text{Dec}(d'_i)$ is equal to the l -th bit of $(j-1)$, where j is such that $s_j^i = q^{i-1}$. Therefore, the result of the application of LOOKUP in Line 17 is equivalent to $\mathbf{c}_{0,s_j^i}^{T_i} (= \mathbf{c}_{0,q^{i-1}}^{T_i})$, and we have $\text{Dec}(\mathbf{c}_{i+1}^{\text{cur}}) = \text{Dec}(\mathbf{c}_{0,q^{i-1}}^{T_i})$. \square

We note that BLOCK generates output for every B monitored ciphertexts while REVERSE generates output for every monitored ciphertext.

We also remark that when $B = 1$, BLOCK consumes every monitored ciphertext from front to back. However, such a setting is slow due to a huge number of CIRCUITBOOTSTRAPPING operations, as pointed out in Section 3.4.

Relaxations of the Assumptions. When B is too large, $\mathbf{c}_{0,q}^{T_i}$ may not be correctly decrypted. We can relax this restriction by inserting BOOTSTRAPPING just after Line 10, which is much like Algorithm 2. When the size $|Q|$ of the states of the DFA M is larger than 2^{N-1} , we cannot store the index j of the state using one TRLWE ciphertext (Line 7). We can relax this restriction by using multiple TRLWE ciphertexts for $\mathbf{c}_{0,q}^{T_i}$ and $\mathbf{c}_{i+1}^{\text{cur}}$.

3.4 Complexity Analysis

Table 3 summarizes the complexity of our algorithms with respect to both the number $|Q|$ of the states of the DFA and the size $|\phi|$ of the LTL formula. We note that, for BLOCK, we do not relax the above assumptions for simplicity. Notice that the number of applications of the homomorphic operations is linear to the length n of the monitored ciphertext. Moreover, the space complexity is independent of n . This shows that our algorithms satisfy the properties essential

to good online monitoring; 1) they only store the minimum of data, and 2) they run quickly enough under a real-time setting [6].

The time and the space complexity of OFFLINE and BLOCK are linear to $|Q|$. Moreover, in these algorithms, when the i -th monitored ciphertext is consumed, only the states reachable by a word of length i are considered, which often makes the scalability even better. In contrast, the time and the space complexity of REVERSE is exponential to $|Q|$. This is because of the worst-case size of the reversed DFA due to the powerset construction. Since the size of the reversed DFA is usually reasonably small, the practical scalability of REVERSE is also much better, which is observed through the experiments in Section 5.

For OFFLINE and BLOCK, $|Q|$ is *doubly* exponential to $|\phi|$ because we first convert ϕ to an NFA (one exponential) and then construct a DFA from the NFA (second exponential). In contrast, for REVERSE, it is known that we can construct a reversed DFA for ϕ of the size of at most *singly* exponential to $|\phi|$ [18]. Note that, in a practical scenario exemplified in Section 5, the size of the DFA constructed from ϕ is expected to be much smaller than the worst one.

4 Oblivious Online LTL Monitoring

In this section, we formalize the scheme of oblivious online LTL monitoring. We consider a two-party setting with a client and a server and refer to the client and the server as Alice and Bob, respectively. Here, we assume that Alice has private data sequence $w = \sigma_1\sigma_2 \dots \sigma_n$ to be monitored where $\sigma_i \in 2^{\text{AP}}$ for each $i \geq 1$. Meanwhile, Bob has a private LTL formula ϕ . The purpose of oblivious online LTL monitoring is to let Alice know if $\sigma_1\sigma_2 \dots \sigma_i \models \phi$ for each $i \geq 1$, while keeping the privacy of Alice and Bob.

4.1 Threat Model

We assume that Alice is *malicious*, i.e., Alice can deviate arbitrarily from the protocol to try to learn ϕ . We also assume that Bob is *honest-but-curious*, i.e., Bob correctly follows the protocol, but he tries to learn w from the information he obtains from the protocol execution. We do not assume that Bob is malicious in the present paper; a protocol that is secure against malicious Bob requires more sophisticated primitives such as zero-knowledge proofs and is left as future work.

Public and Private Data. We assume that the TFHE parameters, the parameters of our proposed algorithms (e.g., I_{boot} and B), Alice’s public key PK, and Alice’s bootstrapping key BK are public to both parties. The input w and the monitoring result are private for Alice, and the LTL formula ϕ is private for Bob.

4.2 Protocol Flow

The protocol flow of oblivious online LTL monitoring is shown in Fig. 3. It takes $\sigma_1, \sigma_2, \dots, \sigma_n, \phi$, and $b \in \mathbb{B}$ as its parameters, where b is a flag that indicates the

	Input : Alice's private inputs $\sigma_1, \sigma_2, \dots, \sigma_n \in 2^{\text{AP}}$, Bob's private LTL formula ϕ , and $b \in \mathbb{B}$
	Output : For every $i \in \{1, 2, \dots, n\}$, Alice's private output representing $\sigma_1 \sigma_2 \dots \sigma_i \models \phi$
1	Alice generates her secret key SK.
2	Alice generates her public key PK and bootstrapping key BK from SK.
3	Alice sends PK and BK to Bob.
4	Bob converts ϕ to a binary DFA $M = (Q, \Sigma = \mathbb{B}, \delta, q_0, F)$.
5	for $i = 1, 2, \dots, n$ do
6	Alice encodes σ_i to a sequence $\sigma'_i := (\sigma_i^1, \sigma_i^2, \dots, \sigma_i^{ \text{AP} }) \in \mathbb{B}^{ \text{AP} }$.
7	Alice calculates $d_i := (\text{Enc}(\sigma_i^1), \text{Enc}(\sigma_i^2), \dots, \text{Enc}(\sigma_i^{ \text{AP} }))$.
8	Alice sends d_i to Bob.
9	Bob feeds the elements of d_i to REVERSE (if $b = 0$) or BLOCK (if $b = 1$).
10	// $\sigma'_1 \cdot \sigma'_2 \dots \sigma'_i$ refers $\sigma_1^1 \dots \sigma_1^{ \text{AP} } \sigma_2^1 \dots \sigma_2^{ \text{AP} } \sigma_3^1 \dots \sigma_i^{ \text{AP} }$. Bob obtains the output TLWE ciphertext c produced by the algorithm, where Dec(c) = $M(\sigma'_1 \cdot \sigma'_2 \dots \sigma'_i)$.
11	Bob randomizes c to obtain c' so that Dec(c) = Dec(c').
12	Bob sends c' to Alice.
13	Alice calculates Dec(c') to obtain the result in plaintext.

Fig. 3: Protocol of oblivious online LTL monitoring.

algorithm Bob uses: REVERSE ($b = 0$) or BLOCK ($b = 1$). After generating her secret key and sending the corresponding public and bootstrapping key to Bob (Lines 1–3), Alice encrypts her inputs into ciphertexts and sends the ciphertexts to Bob one by one (Lines 5–8). In contrast, Bob first converts his LTL formula ϕ to a binary DFA M (Line 4). Then, Bob serially feeds the received ciphertexts from Alice to REVERSE or BLOCK (Line 9) and returns the encrypted output of the algorithm to Alice (Lines 10–13).

Note that, although the alphabet of a DFA constructed from an LTL formula is 2^{AP} [43], our proposed algorithms require a binary DFA. Thus, in Line 4, we convert the DFA constructed from ϕ to a binary DFA M by inserting auxiliary states. Besides, in Line 6, we encode an observation $\sigma_i \in 2^{\text{AP}}$ by a sequence $\sigma'_i := (\sigma_i^1, \sigma_i^2, \dots, \sigma_i^{|\text{AP}|}) \in \mathbb{B}^{|\text{AP}|}$ such that $p_j \in \sigma_i$ if and only if σ_i^j is true, where $\text{AP} = \{p_1, \dots, p_{|\text{AP}|}\}$. We also note that, taking this encoding into account, we need to properly set the parameters for BLOCK to generate an output for each $|\text{AP}|$ -size block of Alice's inputs, i.e., B is taken to be equal to $|\text{AP}|$.

Here, we provide brief sketches of the correctness and security analysis of the proposed protocol. See Appendix A for detailed explanations and proofs.

Correctness. We can show that Alice obtains correct results in our protocol directly by Theorem 2 and Theorem 3.

Security. Intuitively, after the execution of the protocol described in Fig. 3, Alice should learn $M(\sigma'_1 \cdot \sigma'_2 \dots \sigma'_i)$ for every $i \in \{1, 2, \dots, n\}$ but nothing else. Besides, Bob should learn the input size n but nothing else.

Privacy for Alice. We observe that Bob only obtains $\text{Enc}(\sigma_i^j)$ from Alice for each $i \in \{1, 2, \dots, n\}$ and $j \in \{1, 2, \dots, |\text{AP}|\}$. Therefore, we need to show that Bob learns nothing from the ciphertexts generated by Alice. Since TFHE provides IND-CPA security [8], we can easily guarantee the client's privacy for Alice.

Privacy for Bob. The privacy guarantee for Bob is more complex than that for Alice. Here, Alice obtains $\sigma'_1, \sigma'_2, \dots, \sigma'_n$ and the results $M(\sigma'_1 \cdot \sigma'_2 \cdots \sigma'_i)$ for every $i \in \{1, 2, \dots, n\}$ in plaintext. In the protocol (Fig. 3), Alice does not obtain ϕ, M themselves or their sizes, and it is known that a finite number of checking $M(w)$ cannot uniquely identify M if any additional information (e.g., $|M|$) is not given [4, 39]. Thus, it is impossible for Alice to identify M (or ϕ) from the input/output pairs.

Nonetheless, to fully guarantee the model privacy of Bob, we also need to show that, when Alice inspects the result ciphertext c' , it is impossible for Alice to know Bob's specification, i.e., what homomorphic operations were applied by Bob to obtain c' . A TLWE ciphertext contains a random nonce and a noise term. By randomizing c properly in Line 11, we ensure that the random nonce of c' is not biased [41]. By assuming SRL security [12, 25] over TFHE, we can ensure that there is no information leakage regarding Bob's specifications through the noise bias. A more detailed discussion is in Appendix A.

5 Experiments

We experimentally evaluated the proposed algorithms (REVERSE and BLOCK) and protocol. We pose the following two research questions:

RQ1 Are the proposed algorithms scalable with respect to the size of the monitored ciphertexts and that of the DFA?

RQ2 Are the proposed algorithms fast enough in a realistic monitoring scenario?

RQ3 Does a standard IoT device have sufficient computational power acting as a client in the proposed protocol?

To answer RQ1, we conducted an experiment with our original benchmark where the length of the monitored ciphertexts and the size of the DFA are configurable (Section 5.1). To answer RQ2 and RQ3, we conducted a case study on blood glucose monitoring; we monitored blood glucose data obtained by `simglucose`⁴ against specifications taken from [14, 45] (Section 5.2). To answer RQ3, we measured the time spent on the encryption of plaintexts, which is the heaviest task for a client during the execution of the online protocol.

We implemented our algorithms in C++20. Our implementation is publicly available⁵. We used `Spot` [21] to convert a safety LTL formula to a DFA. We also used a `Spot`'s utility program `l1l1filt` to calculate the size of an LTL formula⁶. We used `TFHEpp` [37] as the TFHE library. We used $N = 1024$ as the size of the message represented by one TRLWE ciphertext, which is a parameter of TFHE. The complete TFHE parameters we used are shown in Appendix B.

For RQ1 and RQ2, we ran experiments on a workstation with Intel Xeon Silver 4216 (3.2GHz; 32 cores and 64 threads in total), 128GiB RAM, and Ubuntu

⁴ <https://github.com/jxx123/simglucose>

⁵ Our implementation is uploaded to <https://doi.org/10.5281/zenodo.6558657>.

⁶ We desugared a formula by `l1l1filt` with option `--unabbreviate="eFGiMRW"` and counted the number of the characters.

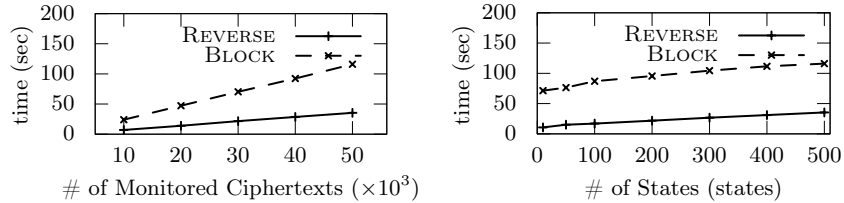


Fig. 4: Experimental results of M_m . The left figure shows runtimes when the number of states (i.e., m) is fixed to 500, while the right one is when the number of monitored ciphertexts (i.e., n) is fixed to 50000.

20.04.2 LTS. We ran each instance of the experiment setting five times and reported the average. We measured the time to consume all of the monitored ciphertexts in the main loop of each algorithm, i.e., in Lines 4–12 in REVERSE and in Lines 2–19 in BLOCK.

For RQ3, we ran experiments on two single-board computers with and without Advanced Encryption Standard (AES) [17] hardware accelerator. ROCK64⁷ has ARM Cortex A53 CPU cores (1.5GHz; 4 cores) with AES hardware accelerator and 4GiB RAM. Raspberry Pi 4⁸ has ARM Cortex A72 CPU cores (1.5GHz; 4 cores) without AES hardware accelerator and 4GiB RAM.

5.1 RQ1: Scalability

Experimental Setup. In the experiments to answer RQ1, we used a simple binary DFA M_m , which accepts a word w if and only if the number of the appearance of 1 in w is a multiple of m . The number of the states of M_m is m .

Our experiments are twofold. In the first experiment, we fixed the DFA size m to 500 and increased the size n of the input word w from 10000 to 50000. In the second experiment, we fixed $n = 50000$ and changed m from 10 to 500. The parameters we used are $I_{\text{boot}} = 30000$ and $B = 150$.

Results and Discussion. Fig. 4 shows the results of the experiments. In the left plot of Fig. 4, we observe that the runtimes of both algorithms are linear to the length of the monitored ciphertexts. This coincides with the complexity analysis in Section 3.4.

In the right plot of Fig. 4, we observe that the runtimes of both algorithms are at most linear to the number of the states. For BLOCK, this coincides with the complexity analysis in Section 3.4. In contrast, this is much more efficient than the exponential complexity of REVERSE with respect to $|Q|$. This is because the size of the reversed DFA does not increase.

In both plots of Fig. 4, we observe that REVERSE is faster than BLOCK. Moreover, in the left plot of Fig. 4, the curve of BLOCK is steeper than that of

⁷ <https://www.pine64.org/devices/single-board-computers/rock64/>

⁸ <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>

REVERSE. This is because 1) the reversed DFA M_m^R has the same size as M_m , 2) CIRCUITBOOTSTRAPPING is about ten times slower than BOOTSTRAPPING, and 3) I_{boot} is much larger than B .

Overall, our experiment results confirm the complexity analysis in Section 3.4. Moreover, the practical scalability of REVERSE with respect to the DFA size is much better than the worst case, at least for this benchmark. Therefore, we answer RQ1 affirmatively.

5.2 RQ2 and RQ3: Case Study on Blood Glucose Monitoring

Experimental Setup. To answer RQ2, we applied REVERSE and BLOCK to the monitoring of blood glucose levels. The monitored values are generated by simulation of type 1 diabetes patients. We used the LTL formulae in Table 4. These formulae are originally presented as signal temporal logic [34] formulae [14, 45], and we obtained the LTL formulae in Table 4 by discrete sampling.

To simulate blood glucose levels of type 1 diabetes patients, we adopted singlucose, which is a Python implementation of UVA/Padova Type 1 Diabetes Simulator [35]. We recorded the blood glucose levels every one minute⁹ and encoded each of them in nine bits. For ψ_1, ψ_2, ψ_4 , we used 720 minutes of the simulated values. For ϕ_1, ϕ_4, ϕ_5 , we used seven days of the values. The parameters we used are $I_{\text{boot}} = 30000$, $B = 9$.

To answer RQ3, we encrypted plaintexts into TRGSW ciphertexts 1000 times using two single-board computers (ROCK64 and Raspberry Pi 4) and reported the average runtime.

Results and Discussion (RQ2). The results of the experiments are shown in Table 5. The result for ψ_4 with REVERSE is missing because the reversed DFA for ψ_4 is too huge, and its construction was aborted due to the memory limit.

Although the size of the reversed DFA was large for ψ_1 and ψ_2 , in all the cases, we observe that both REVERSE and BLOCK took at most 24 seconds to process each blood glucose value on average. This is partly because $|Q|$ and $|Q^R|$ are not so large in comparison with the upper bound described in Section 3.4, i.e., doubly or singly exponential to $|\phi|$, respectively. Since each value is recorded every one minute, at least on average, both algorithms finished processing each value before the next measured value arrived, i.e., any congestion did not occur. Therefore, our experiment results confirm that, in a practical scenario of blood glucose monitoring, both of our proposed algorithms are fast enough to be used in the online setting, and we answer RQ2 affirmatively.

We also observe that average runtimes of ψ_1, ψ_2, ψ_4 and ϕ_1, ϕ_4, ϕ_5 with BLOCK are comparable, although the monitoring DFA of ψ_1, ψ_2, ψ_4 are significantly larger than those of ϕ_1, ϕ_4, ϕ_5 . This is because the numbers of the reachable states during execution are similar among these cases (from 1 up to 27 states). As we mentioned in Section 3.4, BLOCK only considers the states reachable by

⁹ Current continuous glucose monitors (e.g., Dexcom G4 PLATINUM) record blood glucose levels every few minutes, and our sampling interval is realistic.

Table 4: The safety LTL formulae used in our experiments. ψ_1, ψ_2, ψ_4 are originally from [14], and ϕ_1, ϕ_4 , and ϕ_5 are originally from [45].

	LTL formula
ψ_1	$G_{[100,700]}(p_8 \vee p_9 \vee (p_4 \wedge p_7) \vee (p_5 \wedge p_7) \vee (p_6 \wedge p_7) \vee (p_2 \wedge p_3 \wedge p_7))$
ψ_2	$G_{[100,700]}(\neg p_9 \vee (\neg p_7 \wedge \neg p_8) \vee (\neg p_5 \wedge \neg p_6 \wedge \neg p_8) \vee (\neg p_4 \wedge \neg p_6 \wedge \neg p_8) \vee (\neg p_3 \wedge \neg p_6 \wedge \neg p_8) \vee (\neg p_2 \wedge \neg p_6 \wedge \neg p_8) \vee (\neg p_1 \wedge \neg p_6 \wedge \neg p_8))$
ψ_4	$G_{[600,700]}((\neg p_8 \wedge \neg p_9) \vee (\neg p_7 \wedge \neg p_9) \vee (\neg p_4 \wedge \neg p_5 \wedge \neg p_6 \wedge \neg p_9) \vee (\neg p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge \neg p_5 \wedge \neg p_6 \wedge \neg p_9))$
ϕ_1	$G((\neg p_6 \wedge \neg p_7 \wedge p_8 \wedge \neg p_9) \vee (\neg p_5 \wedge \neg p_7 \wedge p_8 \wedge \neg p_9) \vee (\neg p_3 \wedge \neg p_4 \wedge \neg p_7 \wedge p_8 \wedge \neg p_9) \vee (p_4 \wedge p_7 \wedge \neg p_8 \wedge \neg p_9) \vee (p_5 \wedge p_7 \wedge \neg p_8 \wedge \neg p_9) \vee (p_6 \wedge p_7 \wedge \neg p_8 \wedge \neg p_9) \vee (p_1 \wedge p_2 \wedge p_3 \wedge p_7 \wedge \neg p_8 \wedge \neg p_9))$
ϕ_4	$G(\neg p_7 \wedge \neg p_8 \wedge \neg p_9) \implies F_{[0,25]}(p_7 \vee p_8 \vee p_9)$
ϕ_5	$G(p_9 \vee (p_3 \wedge p_7 \wedge p_8) \vee (p_4 \wedge p_7 \wedge p_8) \vee (p_5 \wedge p_7 \wedge p_8) \vee (p_6 \wedge p_7 \wedge p_8) \implies F_{[0,25]}((\neg p_8 \wedge \neg p_9) \vee (\neg p_7 \wedge \neg p_9) \vee (\neg p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge \neg p_6 \wedge \neg p_9)))$

Table 5: Experimental results of blood glucose monitoring, where Q is the state space of the monitoring DFA and Q^R is the state space of the reversed DFA.

Formula ϕ	$ \phi $	$ Q $	$ Q^R $	# of blood glucose values	Algorithm	Runtime (s)	Mean Runtime (ms/value)
ψ_1	40963	10524	2712974	721	REVERSE	16021.06	22220.62
					BLOCK	132.68	184.02
ψ_2	75220	11126	2885376	721	REVERSE	17035.05	23626.97
					BLOCK	131.53	182.43
ψ_4	10392	7026	—	721	REVERSE	—	—
					BLOCK	35.42	49.12
ϕ_1	195	21	20	10081	REVERSE	22.33	2.21
					BLOCK	1741.15	172.72
ϕ_4	494	237	237	10081	REVERSE	42.23	4.19
					BLOCK	2073.45	205.68
ϕ_5	1719	390	390	10081	REVERSE	54.87	5.44
					BLOCK	2084.50	206.78

a word of length i when the i -th monitored ciphertext is consumed, and thus, it ran much faster even if the monitoring DFA is large.

Results and Discussion (RQ3). It took 40.41 and 1470.33 ms on average to encrypt a value of blood glucose (i.e., nine bits) on ROCK64 and Raspberry Pi 4, respectively. Since each value is sampled every one minute, our experiment results confirm that both machines are fast enough to be used in an online setting. Therefore, we answer RQ3 affirmatively.

We also observe that encryption on ROCK64 is more than 35 times faster than that on Raspberry Pi 4. This is mainly because of the hardware accelerator for AES, which is used in TFHEpp to generate TRGSW ciphertexts.

6 Conclusion

We presented the first oblivious online LTL monitoring protocol up to our knowledge. Our protocol allows online LTL monitoring concealing 1) the client's monitored inputs from the server and 2) the server's LTL specification from the client. We proposed two online algorithms (REVERSE and BLOCK) using an

FHE scheme called TFHE. In addition to the complexity analysis, we experimentally confirmed the scalability and practicality of our algorithms with an artificial benchmark and a case study on blood glucose level monitoring.

Our immediate future work is to extend our approaches to LTL semantics with multiple values, e.g., LTL_3 [7] and rLTL [36]. Extension to monitoring continuous-time signals, e.g., against an STL [34] formula, is also future work. Another future direction is to conduct a more realistic case study of our framework with actual IoT devices.

Acknowledgements. This work was partially supported by JST ACT-X Grant No. JPMJAX200U, JSPS KAKENHI Grant No. 22K17873 and 19H04084, and JST CREST Grant No. JPMJCR19K5, JPMJCR2012, and JPMJCR21M3.

References

1. Abbas, H.: Private runtime verification: work-in-progress. In: EMSOFT 2019. p. 11. ACM (2019)
2. Albrecht, M.R., Curtis, B.R., Deo, A., Davidson, A., Player, R., Postlethwaite, E.W., Virdia, F., Wunderer, T.: Estimate all the $\{\text{lwe, ntru}\}$ schemes! In: Catalano, D., De Prisco, R. (eds.) Security and Cryptography for Networks. pp. 351–367. Springer International Publishing, Cham (2018)
3. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology* **9**(3), 169–203 (2015)
4. Angluin, D.: A note on the number of queries needed to identify regular languages. *Inf. Control.* **51**(1), 76–87 (1981)
5. Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. *Computer networks* **54**(15), 2787–2805 (2010)
6. Bartocci, E., Deshmukh, J.V., Donzé, A., Fainekos, G., Maler, O., Nickovic, D., Sankaranarayanan, S.: Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 135–175. Springer (2018)
7. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* **20**(4), 14:1–14:64 (2011)
8. Bellare, M., Rogaway, P.: Introduction to modern cryptography. *Ucsd Cse* **207**, 207 (2005)
9. Bing, K., Fu, L., Zhuo, Y., Yanlei, L.: Design of an internet of things-based smart home system. In: ICICIP 2011. vol. 2, pp. 921–924. IEEE (2011)
10. Blanton, M., Aliasgari, M.: Secure outsourcing of DNA searching via finite automata. In: Foresti, S., Jajodia, S. (eds.) Data and Applications Security and Privacy XXIV, 24th Annual IFIP WG 11.3 Working Conference, 2010. Proceedings. LNCS, vol. 6166, pp. 49–64. Springer (2010)
11. Botta, A., de Donato, W., Persico, V., Pescapè, A.: Integration of cloud computing and internet of things: A survey. *Future Gener. Comput. Syst.* **56**, 684–700 (2016)
12. Brakerski, Z., Döttling, N., Garg, S., Malavolta, G.: Factoring and pairings are not necessary for io: Circular-secure LWE suffices. *IACR Cryptol. ePrint Arch.* p. 1024 (2020)
13. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012. pp. 309–325. ACM (2012)
14. Cameron, F., Fainekos, G., Maahs, D.M., Sankaranarayanan, S.: Towards a verified artificial pancreas: Challenges and solutions for runtime verification. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 3–17. Springer (2015)
15. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Security estimates and parameter choices., Available: https://tfhe.github.io/tfhe/security_and_params.html. Accessed on: January 19th, 2022.
16. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol.* **33**(1), 34–91 (2020)
17. Daemen, J., Rijmen, V.: Aes proposal: Rijndael (1999)
18. De Giacomo, G., Stasio, A.D., Fuggitti, F., Rubin, S.: Pure-past linear temporal and dynamic logic on finite traces. In: Bessiere, C. (ed.) IJCAI 2020. pp. 4959–4965 (2020)

19. Dimitrov, D.V.: Medical internet of things and big data in healthcare. *Healthcare informatics research* **22**(3), 156–163 (2016)
20. Ducas, L., Stehlé, D.: Sanitization of FHE ciphertexts. In: Fischlin, M., Coron, J. (eds.) *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2016, Proceedings, Part I*. LNCS, vol. 9665, pp. 294–310. Springer (2016)
21. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A framework for LTL and ω -automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) *ATVA 2016*. LNCS, vol. 9938, pp. 122–129 (2016)
22. El-Hokayem, A., Falcone, Y.: Bringing runtime verification home. In: Colombo, C., Leucker, M. (eds.) *RV 2018*. LNCS, vol. 11237, pp. 222–240. Springer (2018)
23. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* p. 144 (2012)
24. Frikken, K.B.: Practical private DNA string searching and matching through efficient oblivious automata evaluation. In: Gudes, E., Vaidya, J. (eds.) *Data and Applications Security XXIII, 23rd Annual IFIP WG 11.3 Working Conference, 2009*. Proceedings. LNCS, vol. 5645, pp. 81–94. Springer (2009)
25. Gay, R., Pass, R.: Indistinguishability obfuscation from circular security. *IACR Cryptol. ePrint Arch.* p. 1010 (2020)
26. Gennaro, R., Hazay, C., Sorensen, J.S.: Text search protocols with simulation based security. In: Nguyen, P.Q., Pointcheval, D. (eds.) *PKC 2010*. LNCS, vol. 6056, pp. 332–350. Springer (2010)
27. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) *STOC 2009*. pp. 169–178. ACM (2009)
28. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti, R., Garay, J.A. (eds.) *CRYPTO 2013*. LNCS, vol. 8042, pp. 75–92. Springer (2013)
29. Hopcroft, J.E., Motwani, R., Ullman, J.D.: *Introduction to automata theory, languages, and computation*, 2nd Edition. Addison-Wesley series in computer science, Addison-Wesley-Longman (2001)
30. Ishai, Y., Paskin, A.: Evaluating branching programs on encrypted data. In: Vadhan, S.P. (ed.) *TCC 2007*. LNCS, vol. 4392, pp. 575–594. Springer (2007)
31. Joye, M.: Guide to fully homomorphic encryption over the [discretized] torus. *Cryptology ePrint Archive, Report 2021/1402* (2021), <https://ia.cr/2021/1402>
32. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. *Formal Methods Syst. Des.* **19**(3), 291–314 (2001)
33. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining (T). In: Cohen, M.B., Grunke, L., Whalen, M. (eds.) *ASE 2015*. pp. 81–92. IEEE Computer Society (2015)
34. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) *FORMATS 2004 and FTRTFT 2004*. LNCS, vol. 3253, pp. 152–166. Springer (2004)
35. Man, C.D., Micheletto, F., Lv, D., Breton, M., Kovatchev, B., Cobelli, C.: The uva/padova type 1 diabetes simulator: new features. *Journal of diabetes science and technology* **8**(1), 26–34 (2014)
36. Mascle, C., Neider, D., Schwenger, M., Tabuada, P., Weinert, A., Zimmermann, M.: From LTL to rtl monitoring: improved monitorability through robust semantics. In: Ames, A.D., Seshia, S.A., Deshmukh, J. (eds.) *HSCC 2020*. pp. 7:1–7:12. ACM (2020)

37. Matsuoka, K., Banno, R., Matsumoto, N., Sato, T., Bian, S.: Virtual secure platform: A five-stage pipeline processor over TFHE. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021. pp. 4007–4024. USENIX Association (2021)
38. Mohassel, P., Niksefat, S., Sadeghian, S.S., Sadeghiyan, B.: An efficient protocol for oblivious DFA evaluation and applications. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 398–415. Springer (2012)
39. Moore, E.F.: Gedanken-experiments on sequential machines. In: Automata Studies. (AM-34), pp. 129–154. Princeton University Press (Dec 1956)
40. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, 1977. pp. 46–57. IEEE Computer Society (1977)
41. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM **56**(6), 34:1–34:40 (2009)
42. Sasakawa, H., Harada, H., duVerle, D., Arimura, H., Tsuda, K., Sakuma, J.: Oblivious evaluation of non-deterministic finite automata with application to privacy-preserving virus genome detection. In: Ahn, G., Datta, A. (eds.) WPES 2014. pp. 21–30. ACM (2014)
43. Tabakov, D., Rozier, K.Y., Vardi, M.Y.: Optimized temporal monitors for systemc. Formal Methods Syst. Des. **41**(3), 236–268 (2012)
44. Troncoso-Pastoriza, J.R., Katzenbeisser, S., Celik, M.U.: Privacy preserving error resilient dna searching through oblivious automata. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) CCS 2007. pp. 519–528. ACM (2007)
45. Young, W., Corbett, J., Gerber, M.S., Patek, S., Feng, L.: DAMON: A data authenticity monitoring system for diabetes management. In: IoTDI 2018. pp. 25–36. IEEE Computer Society (2018)

A Correctness and Security of the Protocol

A.1 Correctness

The correctness of our protocol shown in Fig. 3 is formulated as follows:

Theorem 4. *Let ϕ be Bob’s LTL formula and M be the binary DFA converted from ϕ . Let $\sigma_1, \sigma_2, \dots, \sigma_n \in 2^{\text{AP}}$ be Alice’s inputs and $\sigma'_1, \sigma'_2, \dots, \sigma'_n \in \mathbb{B}^{|\text{AP}|}$ be the encoded Alice’s inputs. We assume 1) the protocol uses REVERSE (i.e., $b = 0$) and, for every $i \in \{1, 2, \dots, n\}$, c_i in Algorithm 3 can be correctly decrypted, or 2) the protocol uses BLOCK (i.e., $b = 1$) and, for every $i \in \{1, 2, \dots, n\}$, c_i in Algorithm 4 can be correctly decrypted. Then, by following the protocol described in Fig. 3, Alice obtains a Boolean value representing if $\sigma_1\sigma_2\dots\sigma_i \models \phi$ for every $i \in \{1, 2, \dots, n\}$.*

Proof (sketch). It suffices to show that, for every $i \in \{1, 2, \dots, n\}$, the decryption of the resulted ciphertext c' in Line 10 is equal to $M(\sigma'_1 \cdot \sigma'_2 \cdots \sigma'_i)$, which indicates $\sigma_1\sigma_2\dots\sigma_i \models \phi$. When executing Line 10 in Fig. 3, we can confirm that Bob has already fed the monitored ciphertexts d_1, d_2, \dots, d_i to Algorithm 3 or Algorithm 4. Therefore, by Theorem 2 and Theorem 3, the TLWE ciphertext c produced by these algorithms can be correctly decrypted, and we have $\text{Dec}(c) = M(\sigma'_1 \cdot \sigma'_2 \cdots \sigma'_i)$. Moreover, randomization (i.e., adding $\text{Enc}(0)$ to c) in Line 11 does not change its message, i.e., $\text{Dec}(c') = \text{Dec}(c)$ holds. \square

A.2 Security

In this subsection, we formally define the privacy of Alice and Bob, and based on these definitions, we prove the security of the protocol described in Fig. 3. We refer to [30] for the formal definitions of the privacy of the client (Alice) and the server (Bob). We note that the privacy of the server requires an additional assumption of TFHE called *shielded randomness leakage* (SRL) security [12,25].

Definition 1 (Representation model ([30, Definition 2])). A representation model is a polynomial-time computable function $U: \mathbb{B}^* \times \mathbb{B}^* \rightarrow \mathbb{B}^*$, where $U(P, x)$ is referred to as the value returned by a “program” P on the input x .

Definition 2 (Computing on encrypted data ([30, Definition 5])). Let $U: \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ be a polynomial-time computable function. A protocol for evaluating programs from U on encrypted data is defined by a tuple of algorithms $(\text{Gen}, \text{Enc}, \text{Eval}, \text{Dec})$ and proceeds as follows.

SETUP Given a security parameter k , the client computes $(\text{PK}, \text{SK}) \leftarrow \text{Gen}(1^k)$ and saves SK for a later use.

ENCRYPTION The client computes $c \leftarrow \text{Enc}(\text{PK}, x)$, where x is the input on which a program P should be evaluated.

EVALUATION Given the public key PK, the ciphertext c , and a program P , the server computes an encrypted output $c' \leftarrow \text{Eval}(1^k, \text{PK}, c, P)$.

DECRYPTION Given the encrypted output c' , the client outputs $y \leftarrow \text{Dec}(\text{SK}, c')$.

We require that if both parties act according to the above protocol, then for every input x , program P , and security parameter $k \in \mathbb{N}$, the output y of the final decryption phase is equal to $U(P, x)$ except, perhaps, with negligible probability in k .

Definition 3 (Client privacy ([30, Definition 6])). Let $\Pi = (\text{Gen}, \text{Enc}, \text{Eval}, \text{Dec})$ be a protocol for computing on encrypted data. We say that Π satisfies the client privacy requirement if the advantage of any probabilistic polynomial time (PPT) adversary Adv in the following game is negligible in the security parameter k :

- Adv is given 1^k and generates a pair $x_0, x_1 \in \{0, 1\}^*$ such that $|x_0| = |x_1|$.
- Let $b \xleftarrow{\mathbb{R}} \{0, 1\}$, $(\text{PK}, \text{SK}) \leftarrow \text{Gen}(1^k)$, and $c \leftarrow \text{Enc}(\text{PK}, x_b)$.
- Adv is given the challenge (PK, c) and outputs a guess b' .

The advantage of Adv is defined as $\Pr[b = b'] - 1/2$

Definition 4 (Size hiding server privacy: honest-but-curious model ([30, Definition 7 and Definition 8])). Let $\Pi = (\text{Gen}, \text{Enc}, \text{Eval}, \text{Dec})$ be a protocol for evaluating programs from a representation model U on encrypted data. We say that Π has computational server privacy in the honest-but-curious model if there exists a PPT algorithm Sim such that the following holds. For every polynomial-size circuit family D , there is a negligible function $\epsilon(\cdot)$ such that for every security parameter k , input $x \in \{0, 1\}^*$, pair (PK, c) that can be generated by Gen, Eval on inputs k, x , and program $P \in \{0, 1\}^*$, we have

$$\Pr[D(\text{Eval}(1^k, \text{PK}, c, P)) = 1] - \Pr[D(\text{Sim}(1^k, 1^{|x|}, \text{PK}, U(P, x))) = 1] \leq \epsilon(k).$$

Definition 5 (Size hiding server privacy: fully malicious model ([30, Definition 12 and Definition 13])). Let $\Pi = (\text{Gen}, \text{Enc}, \text{Eval}, \text{Dec})$ be a protocol for evaluating programs from a representation model U on encrypted data. We say that Π has computational server privacy in the fully malicious model if there exists a computationally unbounded, randomized algorithm Sim such that the following holds. For every polynomial-size circuit family D , there is a negligible function ϵ such that for every security parameter k , arbitrary public key PK , and arbitrary ciphertext c^* , there exists an “effective” input x^* such that for every program $P \in \{0, 1\}^*$, we have

$$\Pr[D(\text{Eval}(1^k, \text{PK}, c, P)) = 1] - \Pr[D(\text{Sim}(1^k, \text{PK}, c^*, U(P, x^*))) = 1] \leq \epsilon(k).$$

Theorem 5. The protocol described in Fig. 3 provides client privacy according to Definition 3.

Proof (sketch). Client privacy of the protocol readily follows from the fact that TFHE is an FHE scheme based on Learning-With-Errors problem [16, 41]. \square

Theorem 6. Assuming the SRL security of TFHE, the protocol described in Fig. 3 provides size hiding server privacy against an honest-but-curious client as defined in Definition 4.

Proof (sketch). First, we abstract our protocol after the i -th round of execution as $c_i = \text{Eval}(1^k, \text{PK}, \{d_j\}_{j=1}^i, M, i)$ where it holds that decryption of c_i is equal to $M(\sigma'_1 \cdot \sigma'_2 \cdots \sigma'_i)$.

On inputs $(1^k, 1^{|x|}, \text{PK}, U(P, x))$, we define a simulator Sim , which proceeds as follows:

- $c \leftarrow \text{Enc}_{\text{PK}}(U(P, x))$
- Return c

By the assumption of the SRL security of TFHE, it holds that

$$c \stackrel{c}{\equiv} c_i.$$

Therefore, for any PPT adversary \mathcal{A} , we have that

$$\begin{aligned} \Pr[\mathcal{A}(\text{Eval}(1^k, \text{PK}, \{d_j\}_{j=1}^i, M, i)) = 1] - \Pr[\mathcal{A}(\text{Sim}(1^k, 1^{|x|}, \text{PK}, U(P, x))) = 1] \\ = \epsilon(k) \end{aligned}$$

and the theorem follows. \square

Theorem 7. Assuming the SRL security of TFHE and the honest generation of the public key PK , the protocol described in Fig. 3 provides size hiding server privacy against a malicious client as defined in Definition 5.

As noted in [20], a malicious client may try to generate invalid ciphertexts or public keys to gain an advantage against the DFA held by the server. Fortunately, our protocol can easily achieve size hiding server privacy against a malicious client (i.e., Alice) by ensuring the honest generation of the public key PK in combined with the SRL security of TFHE [20], and we omit a formal proof for Theorem 7.

Table 6: Table of TFHE parameters

Parameter	Value in implementation	Meaning
q	2^{32}	The modulus for discretizing Torus for <code>lv10</code> and <code>lv11</code>
\bar{q}	2^{64}	The modulus for discretizing Torus for <code>lv12</code>
\bar{N}	635	The length of the <code>TLWElv0</code> ciphertext
α	2^{-15}	The standard deviation of the noise for the fresh <code>TLWElv0</code> ciphertext
N	2^{10}	The length of the <code>TLWElv1</code> ciphertext and the dimension of <code>TRLWElv1</code> ciphertext
α	2^{-25}	The standard deviation of the noise for the fresh <code>TLWElv1</code> , <code>TRLWElv1</code> and <code>TRGSWlv1</code> ciphertext
\bar{N}	2^{11}	The length of the <code>TLWElv2</code> ciphertext and the dimension of <code>TRLWElv2</code> ciphertext
$\bar{\alpha}$	2^{-44}	The standard deviation of the noise for the fresh <code>TLWElv2</code> , <code>TRLWElv2</code> and <code>TRGSWlv2</code> ciphertext
l	3	Half of the number of rows in <code>TRGSWlv1</code>
Bg	2^6	The base for <code>CMux</code> with <code>TRGSWlv1</code>
\bar{l}	4	Half of the number of rows in <code>TRGSWlv2</code>
$\bar{B}g$	2^9	The base for <code>CMux</code> with <code>TRGSWlv2</code>
t	7	The number of digits in <code>IDENTITYKEYSWITCHING</code>
$base$	2^2	The base for <code>IDENTITYKEYSWITCHING</code>
\bar{t}	10	The number of digits in <code>PRIVATEKEYSWITCHING</code>
\bar{base}	2^3	The base for <code>PRIVATEKEYSWITCHING</code>

B TFHE Parameters

The parameters for TFHE are the foundation of the security of our proposed protocols and greatly affect the performance. The security of the parameters is estimated by using lwe-estimator [3]. We use the default parameter provided by TFHEpp [37]. This parameter is selected to maximize the performance of Bootstrapping while achieving 128-bit security. In Table 6, we show all necessary parameters used in our implementation and briefly explain the meaning of each parameter. The parameters which directly affect the security guarantee are q , \bar{N} , α , N , α , \bar{N} , and $\bar{\alpha}$. The example of the estimation code for the security of TFHE using lwe-estimator is given at [15]. In the following subsections, we briefly explain some ideas which are omitted in the main text for simplicity but are necessary to understand the meaning of parameters.

B.1 Discretization of Torus

TFHE uses Torus, $\mathbb{T} = \mathbb{R}/\mathbb{Z}$, i.e., the set of real number modulo 1, as one of the most fundamental primitives, but lwe-estimator can treat only \mathbb{Z}_q , i.e., the set of integers modulo q . Therefore, in actual implementation, we discretized Torus into q parts (\mathbb{T}_q). By this discretization, we can reduce the security of `TLWE`, `TRLWE`, and `TRGSW` into standard `LWE`, `RLWE`, and `RGSW` [31]. Therefore, we can use lwe-estimator to estimate the security of TFHE. In addition to that, because floating-point operations are generally slower than integer operations, this discretization also gives a performance benefit. This is why we need the parameter q .

B.2 Levels in TFHE

In the main text, we introduced only one kind of TLWE ciphertexts, which can be converted from TRLWE by `SAMPLEEXTRACT`. In the real implementation of TFHE, there are three kinds of TLWE ciphertexts (`TLWElv0`, `TLWElv1`, and `TLWElv2`) and two kinds of TRLWE ciphertexts (`TRLWElv1` and `TRLWElv2`). `TLWElv1` and `TRLWElv1` are the ones introduced in the main text as TLWE and TRLWE, respectively. `TLWElv0` is more compact in ciphertext size than `TLWElv1` but needs more noise to establish 128-bit security. More noise means less capability for homomorphic computations. Therefore, `TLWElv0` only appears in `BOOTSTRAPPING` to reduce the complexity. `TLWElv2` and `TRLWElv2` are larger in ciphertext size than `TLWElv1` and `TRLWElv1`, respectively. Thus, they need less noise to establish 128-bit security and have more capability for homomorphic computations. `TLWElv2` and `TRLWElv2` are used in `CIRCUITBOOTSTRAPPING` as intermediate representations. They can be used in our protocols instead of `TLWElv1` and `TRLWElv1`, but it will cause performance degradation due to their ciphertext size.

Because we can convert `TRLWElv1` and `TRLWElv2` to `TLWElv1` and `TLWElv2`, respectively by `SAMPLEEXTRACT` and “due to the absence of known cryptanalytic techniques exploiting algebraic structure”, it is standard to assume the security of `TRLWElv1` and `TRLWElv2` are the same as `TLWElv1` and `TLWElv2` respectively [2].

B.3 IDENTITYKEYSWITCHING

`IDENTITYKEYSWITCHING` is one of the homomorphic operations in TFHE. This operation converts a `TLWElv1` ciphertext into a `TLWElv0` ciphertext which holds the same plaintext message. The parameters t and $base$ are used in this operation and are not related to security but the performance and the noise growth. Faster parameters generally mean more noise growth in this operation. Thus, the parameters are selected to balance the performance and the noise growth.

B.4 PRIVATEKEYSWITCHING

`PRIVATEKEYSWITCHING` is one of the homomorphic operations in TFHE. This operation converts a `TLWElv2` ciphertext into a `TRLWElv1` ciphertext which holds the result of applying the private Lipschitz (linear) function to the plaintext of the input `TLWElv2` ciphertext. In `CIRCUITBOOTSTRAPPING`, we need to apply the function which depends on the part of the secret key. Thus, `PRIVATEKEYSWITCHING` is used to hide the function to avoid leaking a part of the secret key. The parameters \bar{t} and \overline{base} are used in this operation and not related to security, but the performance and the noise growth.

B.5 Parameters for TRGSW

In the main text, we only introduced `TRGSWlv1` as TRGSW, but we also use `TRGSWlv2` in `CIRCUITBOOTSTRAPPING`. The parameter l and \bar{l} determine the

shape of TRGSWlv1 and TRGSWlv2 ciphertexts, respectively, and Bg and \overline{Bg} are used in CMUX with TRGSWlv1 and TRGSWlv2 ciphertexts, respectively. Therefore, they highly affect the performance of CMUX. A TRGSWlv1 and TRGSWlv2 ciphertext can be seen as the $2l$ and $2\bar{l}$ dimensional vector of TRLWElv1 and TRLWElv2 ciphertexts, respectively. Though each row encrypts the plaintext which is a multiple of the other row’s plaintext, there are no known cryptanalytic techniques exploiting this fact. Therefore, the security of TRGSWlv1 and TRLWElv2 ciphertexts are assumed to be the same as TRLWElv1 and TLWElv1, TRLWElv2 and TLWElv2, respectively. Increasing l and \bar{l} means increasing the number of polynomial multiplications in CMUX, which are the heaviest operation in CMUX, but exponentially reduce the noise growth in CMUX. Bg and \overline{Bg} are related to noise growth only, and there is the optimal value for fixed l and \bar{l} . Therefore, l, Bg, \bar{l} and \overline{Bg} are selected to balance between the performance and the noise growth.

C Detailed Experiment Results and Discussion

Table 7 shows the detailed results of M_m when the number of states is fixed. We observe that the memory usage is constant to the number of the monitored ciphertexts. We also observe that, in both algorithms, the runtimes of CMUX and CIRCUITBOOTSTRAPPING are linear to the length n of the monitored ciphertexts. These observations coincide with the complexity analysis in Section 3.4. In contrast, we observe that the runtimes of BOOTSTRAPPING do not change so much from $n = 30000$ to 50000 . This is because we set $I_{\text{boot}} = 30000$, and Lines 8–10 in Algorithm 3 are executed only once.

Table 8 shows the detailed results of M_m when the number of the monitored ciphertexts is fixed. The table shows that, in both algorithms, the runtimes of CMUX and BOOTSTRAPPING are linear to the number m of the states, and the runtimes of CIRCUITBOOTSTRAPPING are sublinear to m . Since, as described in Section 5, the number of states of the reversed DFA of M is equal to that of M , these observations coincide with the complexity analysis in Section 3.4.

In both Table 7 and Table 8, we observe that the memory usage of BLOCK is larger than that of REVERSE. This is because CIRCUITBOOTSTRAPPING needs a larger bootstrapping key than BOOTSTRAPPING, and we need to place the key on the memory when CIRCUITBOOTSTRAPPING is performed.

Table 9 shows the detailed results of blood glucose monitoring. We observe that, when we use REVERSE, the amounts of memory used for ψ_1 and ψ_2 are much larger than those for ϕ_1, ϕ_4, ϕ_5 . This is because the numbers of states of the reversed DFA of ψ_1, ψ_2 are much larger than those of ϕ_1, ϕ_4, ϕ_5 .

D Extended Protocol for General Output Interval

In this section, we extend our protocol (Fig. 3) to output the monitoring result after consuming every I_{out} Alice’s inputs, where I_{out} is the interval of the monitoring output. We present the extended protocol in Fig. 5. The main difference

Table 7: Experimental results of M_m when the number of the states (i.e., m) is fixed to 500.

Algorithm	# of Monitored Ciphertexts	Runtime (s)				Memoery Usage (GiB)
		CMUX	BOOTSTRAPPING	CIRCUITBOOTSTRAPPING	Total	
REVERSE	10000	6.94	—	—	6.98	0.34
	20000	13.90	—	—	13.97	0.34
	30000	20.79	0.75	—	21.65	0.34
	40000	27.64	0.83	—	28.63	0.34
	50000	34.55	0.71	—	35.44	0.34
BLOCK	10000	6.09	—	16.60	24.07	2.72
	20000	12.33	—	32.20	47.19	2.72
	30000	18.49	—	47.81	70.32	2.72
	40000	24.48	—	62.60	92.40	2.72
	50000	30.88	—	78.71	116.11	2.72

Table 8: Experimental results of M_m when the number of monitored ciphertexts (i.e., n) is fixed to 50000.

Algorithm	# of States	Runtime (s)				Memoery Usage (GiB)
		CMUX	BOOTSTRAPPING	CIRCUITBOOTSTRAPPING	Total	
REVERSE	10	10.52	0.13	—	10.70	0.33
	50	14.60	0.43	—	15.14	0.33
	100	16.36	0.52	—	17.04	0.33
	200	21.19	0.53	—	21.84	0.33
	300	25.95	0.64	—	26.72	0.33
	400	30.18	0.68	—	31.03	0.34
	500	34.55	0.71	—	35.44	0.34
	BLOCK	10	8.02	—	60.70	71.35
50		8.30	—	65.20	76.41	2.71
100		10.55	—	73.09	87.03	2.71
200		15.86	—	75.38	95.50	2.71
300		20.26	—	79.00	104.43	2.72
400		25.90	—	79.73	111.56	2.72
500		30.88	—	78.71	116.11	2.72

between Fig. 5 and Fig. 3 is that the output is generated every after I_{out} Alice’s inputs consumed (in Lines 10–14). Since we can prove the correctness and the security of the extended protocol in the same way as the original one, we omit the proof.

Notice that, in this setting, the block size B for BLOCK can be taken to be equal to $I_{\text{out}} \times |\text{AP}|$. As the complexity analysis in Section 3.4 implies, the larger B becomes the faster BLOCK works. Therefore, setting I_{out} large improves the performance of BLOCK.

Table 9: Experimental results of blood glucose monitoring, where Q is the state space of the monitoring DFA and Q^R is the state space of the reversed DFA.

Formula	$ Q $	$ Q^R $	# of blood glucose values	Algorithm	Runtime (s)				Average Runtime (ms/value)	Memory Usage (GiB)
					CMUX	BOOTSTRAPPING	CIRCUIT BOOTSTRAPPING	Total		
ψ_1	10524	2712974	721	REVERSE BLOCK	16005.55 1.33	— —	— 105.96	16021.06 132.68	22220.62 184.02	42.26 2.85
ψ_2	11126	2885376	721	REVERSE BLOCK	17019.28 1.35	— —	— 104.96	17035.05 131.53	23626.97 182.43	44.92 2.86
ψ_4	7026	—	721	REVERSE BLOCK	— 0.50	— —	— 20.64	— 35.42	— 49.12	— 2.80
ϕ_1	21	20	10081	REVERSE BLOCK	21.95 17.97	0.24 —	— 1679.74	22.33 1741.15	2.21 172.72	0.33 2.69
ϕ_4	237	237	10081	REVERSE BLOCK	41.43 32.84	0.58 —	— 1984.61	42.23 2073.45	4.19 205.68	0.33 2.70
ϕ_5	390	390	10081	REVERSE BLOCK	53.50 34.12	1.07 —	— 1988.20	54.87 2084.50	5.44 206.78	0.34 2.70

Input	: Alice's private inputs $\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_n \in 2^{AP}$, Bob's private LTL formula ϕ , $I_{out} \in \mathbb{N}^+$, and $b \in \mathbb{B}$
Output	: For every $i \in \mathbb{N}^+$ ($i \leq \lfloor n/I_{out} \rfloor$), Alice's private output which represents $\sigma_1 \sigma_2 \dots \sigma_{i \times I_{out}} \models \phi$
1	Alice generates her secret key SK.
2	Alice generates her public key PK and bootstrapping key BK from SK.
3	Alice sends PK and BK to Bob.
4	Bob converts ϕ to a binary DFA $M = (Q, \Sigma = \mathbb{B}, \delta, q_0, F)$.
5	for $i = 1, 2, \dots, n$ do
6	Alice encodes σ_i to a sequence $\sigma'_i := (\sigma_i^1, \sigma_i^2, \dots, \sigma_i^{ AP }) \in \mathbb{B}^{ AP }$.
7	Alice calculates $d_i := (\text{Enc}(\sigma_i^1), \text{Enc}(\sigma_i^2), \dots, \text{Enc}(\sigma_i^{ AP }))$.
8	Alice sends d_i to Bob.
9	Bob feeds the elements of d_i to REVERSE (if $b = 0$) or BLOCK (if $b = 1$).
10	if $i \bmod I_{out} = 0$ then
11	// $\sigma'_1 \cdot \sigma'_2 \dots \sigma'_i$ refers $\sigma_1^1 \dots \sigma_1^{ AP } \sigma_2^1 \dots \sigma_2^{ AP } \sigma_3^1 \dots \sigma_i^{ AP }$. Bob obtains the output TLWE ciphertext c produced by the algorithm, where Dec(c) = $M(\sigma'_1 \cdot \sigma'_2 \dots \sigma'_i)$.
12	Bob randomizes c to obtain c' so that Dec(c) = Dec(c').
13	Bob sends c' to Alice.
14	Alice calculates Dec(c') to obtain the result in plaintext.

Fig. 5: Protocol of oblivious online LTL monitoring (extended for the output interval I_{out}).