Authenticated Streamwise On-line Encryption[*]

Patrick P. Tsang[†], Rouslan V. Solomakhin and Sean W. Smith

Department of Computer Science
Dartmouth College
Hanover, NH 03755
USA

Dartmouth Computer Science Technical Report
TR2009-640

March 28, 2009

Abstract

In *Blockwise On-line Encryption*, encryption and decryption return an output block as soon as the next input block is received. In this paper, we introduce *Authenticated Streamwise On-line Encryption* ($\mathcal{ASOE}$), which operates on plaintexts and ciphertexts as streams of *arbitrary* length (as opposed to fixed-sized blocks), and thus significantly reduces message expansion and end-to-end latency. Also, $\mathcal{ASOE}$ provides data authenticity as an option. $\mathcal{ASOE}$ can therefore be used to efficiently secure resource-constrained communications with real-time requirements such as those in the electric power grid and wireless sensor networks.

We investigate and formalize $\mathcal{ASOE}$'s strongest achievable notion of security, and present a construction that is secure under that notion. An instantiation of our construction incurs *zero* end-to-end latency due to buffering and only *48 bytes* of message expansion, regardless of the plaintext-size.

Keywords: blockwise on-line encryption, authenticated encryption, ciphers, provable security, critical infrastructure protection

[*]This work was supported in part by the National Science Foundation, under grant CNS-0524695, the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, and the Institute for Security Technology Studies, under Grant number 2005-DD-BX-1091 awarded by the Bureau of Justice Assistance. The views and conclusions do not necessarily represent those of the sponsors.

[†]Corresponding author. Email him at patrick@cs.dartmouth.edu.

Contents

1 Introduction

1.1 On-line Encryption

In applications such as the Secure Shell (SSH), the symmetric encryption and decryption algorithms—or equivalently, the devices executing them—operate in an *"on-line"* manner: individual plaintext- and ciphertext-blocks are processed and output as soon as the next input block is received (i.e., without waiting until the receipt of the entire plaintext or ciphertext). On-line encryption and decryption are a desirable feature, as they significantly reduce the end-to-end latency due to buffering — as well as the storage needed — at the encryption and decryption devices (from depending on the size of the ciphertext/plaintext to depending on the size of only a single block[1]). In SSH, such feature is crucial as it permits the simulation of a real-time tunnel for interactive traffic.

The study of *on-line encryption* started in 2002 when Bellare et al. [4] published an attack on the provably CPA-secure (i.e., secure against *Chosen-Plaintext Attacks*) symmetric encryption used in SSH. The attack, and similar attacks on several other symmetric encryption identified soon after [12], were possible despite the provable security due to a discrepancy between the model under which the encryption is proven secure and how the encryption is actually used.

Specifically, it had traditionally been assumed [2, 16] that the encryption (resp. decryption) device in a symmetric encryption operates in an *"off-line"* manner: it takes a plaintext (resp. ciphertext) in its entirety as input, after which it outputs the corresponding ciphertext (resp. plaintext); an adversary can thus observe the corresponding ciphertext (resp. plaintext) only after he has submitted the plaintext (resp. ciphertext) in its entirety to the device. *On-line* encryption and decryption, however, allow the adversary to observe the encryption (resp. decryption) of individual plaintext- (resp. ciphertext-) blocks and craft the remaining blocks adaptively, thereby giving him more power than assumed in the model when launching an attack.

These attacks thus called for the study of *on-line encryption* [6], a symmetric encryption scheme which is secure even if *both* the encryption device and the decryption device are operated in an on-line manner. We note that some have referred to their construction as on-line encryption [9, 10], even though only encryption is on-line. As it turns out, it is relatively easy to construct a secure scheme when only encryption is on-line. Nonetheless, such scheme is not that useful: the end-to-end latency still depends on the size of the plaintext/ciphertext; it does not permit the simulation of a real-time tunnel, either. In this paper, we call a scheme *on-line encryption* only if *both* encryption and decryption are on-line.

1.2 (Authenticated) Blockwise On-line Encryption

Today, all existing on-line encryption constructions have been built from some secure block cipher, e.g., AES [13], and have been *blockwise*: their encryption device and decryption device take as input, operate on, and return as output fixed-sized blocks of data, where the size of each block is the same as the block-size of the underlying block cipher. These on-line encryption schemes are known as *Blockwise On-line Encryption* ($\mathcal{BOE}$).

Authenticated Blockwise On-line Encryption ($\mathcal{ABOE}$) is like $\mathcal{BOE}$, but additionally provides data authenticity, in a way analogous to how *Authenticated Encryption* [5] ($\mathcal{AE}$) adds data authenticity to conventional *Symmetric Encryption* ($\mathcal{SE}$). In particular, the decryption device additionally outputs

[1]The block-size equals that of the underlying block cipher, e.g., 128 bits for AES.

a boolean value at the end of a decryption, indicating whether the output plaintext, is authentic. (We will give a precise definition of data authenticity later in the paper).

Boldyreva and Taesombut [6] first formalized the strongest achievable notion of security, namely IND-BLK-CCA (more details on this later), for $\mathcal{BOE}$ and presented a $\mathcal{BOE}$ construction that is secure under that notion. In the same paper, they also presented an $\mathcal{ABOE}$ construction.

There are several other existing $(\mathcal{A})\mathcal{BOE}$-related constructions. Bard [1] studied the construction of $\mathcal{BOE}$ from various modes of operation, but the construction has only CPA-security. Fouque et. al proposed an "$\mathcal{ABOE}$" construction that is secure against chosen ciphertext attacks [9], but decryption is not on-line. Similarly, decryption is also not on-line in the work due to Fouque et al. [10].

1.2.1 The Challenge

$(\mathcal{A})\mathcal{BOE}$ processes and outputs plaintext-/ciphertext-blocks as soon as the next input block is received. A natural question — the answer to which has both theoretical and practical interests — to ask is then: "what if the input is smaller than a block?" More generally,

"What if the input is a stream of bits of arbitrary length?"

Such situations occur in practice, when the input plaintext/ciphertext is broken up into streams of varying lengths (as small as 1 bit) before it arrive at the encryption/decryption device, one stream at a time. In Section 8, we discuss two examples, one in the electric power grid and the other in wireless sensor networks (WSNs).

1.3 A New Cryptographic Primitive: Authenticated Streamwise On-line Encryption

To answer the challenge above, we introduce *Authenticated Streamwise On-line Encryption* ($\mathcal{ASOE}$), an authenticated encryption scheme with the following features:

- **Zero buffering latency** Plaintexts and ciphertexts may be fragmented into streams of arbitrary lengths. Upon receiving an input stream, the encryption and decryption devices return an output stream *immediately*, i.e., before the arrival of the next input stream.

  In particular, the fragmentation of the ciphertext when input to the decryption device can be different from its fragmentation when output by the encryption device. This may occur when the ciphertext is in transit due to, e.g., IP fragmentation in the Internet Protocol (IP).

- **Constant message expansion** The ciphertext resulted from encrypting a plaintext is of size larger than the plaintext-size by at most a fixed amount, independent of the plaintext-size. Consequently, the incurred latency due to message expansion is also independent of the plaintext-size.

- **Constant storage size** The encryption and decryption devices have a fixed amount of storage, independent of the maximum allowable size of the plaintexts and ciphertexts. Hence, even devices with limited storage such as sensor nodes can encrypt and authenticate long messages.

1.3.1 Paper Organization

The rest of the paper starts with a brief description of how our proposed $\mathcal{ASOE}$ encryption device and decryption device operate (Section 2). This provides us with a basis and the necessary vocabulary for explaining the challenges in constructing $\mathcal{ASOE}$ that meets the desired security, functionality and performance goals (Section 3). We then highlight the core ideas in our solution to overcome those challenges (Section 4). After that, we formalize the syntax and security of $\mathcal{ASOE}$ (Section 5), before we provide the details of our $\mathcal{ASOE}$ construction (Section 6) and evaluate its performance (Section 7). Finally, we discuss two application scenarios that can benefit from $\mathcal{ASOE}$ (Section 8), and conclude the paper (Section 9).

2 The $\mathcal{ASOE}$ Encryption and Decryption Devices

A pair of an $\mathcal{ASOE}$ encryption device $\mathcal{E}$ and its corresponding $\mathcal{ASOE}$ decryption device $\mathcal{D}$ are installed with a shared key k.

2.1 The encryption device

To encrypt a *plaintext* $m \in \{0,1\}^*$, $\mathcal{E}$ operates in three stages as follows.

1. *(Initialization.)* $\mathcal{E}$ is first initialized with an *initialization vector* (IV) v. As a result, $\mathcal{E}$ outputs a *ciphertext header* h of a fixed size. For security reasons, no IV should be reused to initialize $\mathcal{E}$ installed with the same key.

2. *(Stream encryption.)* The plaintext m may be arbitrarily fragmented into ℓ *plaintext streams* $m_1, m_2, \dots, m_\ell \in \{0,1\}^*$ such that $m = m_1||m_2||\dots||m_\ell$, which become available to $\mathcal{E}$ for encryption one stream at a time, in that order. When plaintext stream m_i becomes available, $\mathcal{E}$ encrypts m_i and outputs a corresponding *ciphertext stream* x_i immediately, i.e., before the next plaintext stream becomes available. The constant message expansion property requires that $|m_i| = |x_i|$.

3. *(Finalizing.)*. When $\mathcal{E}$ finished encrypting the last plaintext stream m_ℓ, it finalizes the encryption and outputs a *ciphertext trailer* t of a fixed size.

We call the in-order concatenation of all ciphertext streams the *ciphertext body* x, i.e., $x = x_1||x_2||\dots||x_\ell$. The entire ciphertext c as a result of $\mathcal{ASOE}$-encrypting plaintext m is hence the header-body-trailer triple (h, x, t).

2.2 The decryption device

To decrypt a ciphertext $c = (h, x, t)$, the corresponding $\mathcal{ASOE}$ decryption device $\mathcal{D}$ operates in three stage as follows.

1. *(Initialization.)* $\mathcal{D}$ is first initialized with the ciphertext header h. The initialization may or may not succeed. If it fails, $\mathcal{D}$ immediately terminates the decryption of c as failure, and hence refuses to decrypt c. As will become clear later, this step is the cornerstone of a secure $\mathcal{ASOE}$.

2. *(Stream decryption.)* The ciphertext body x may be arbitrarily fragmented into ℓ' ciphertext streams $x'_1, x'_2, \ldots, x'_\ell$ such that $x' = x'_1||x'_2||\ldots||x'_{\ell'}$, which become available to $\mathcal{D}$ for decryption one stream at a time, in that order. When ciphertext stream x'_i becomes available, $\mathcal{D}$ decrypts x'_i and outputs a corresponding plaintext stream m'_i immediately. Again we require that $|x'_i| = |m'_i|$.

3. *(Finalizing.)* When $\mathcal{D}$ finished decrypting the last ciphertext stream $x'_{\ell'}$, it finalizes the decryption and outputs either `true` or `false` as its decision on the authenticity of the recovered plaintext $m' = m'_1||m'_2||\ldots||m'_{\ell'}$.

Any $\mathcal{ASOE}$ construction should have *completeness*: in the absence of an adversary, (1) the recovered plaintext m' equals the original plaintext m, and (2) $\mathcal{D}$ outputs `true` at the finalizing stage, indicating that m' is authentic.

3 Security Goals

To construct a secure $\mathcal{ASOE}$, one must first understand what security means for $\mathcal{ASOE}$. As it turns out, the data authenticity requirement in $\mathcal{ASOE}$ is a pretty standard one. Below we thus first focus on the data privacy requirement.

3.1 Data Privacy

Boldyreva and Taesombut [6] first rigorously argued that it is *impossible* for $\mathcal{BOE}$ to have IND-CCA security[2], and formalized the strongest notion of data privacy achievable by $\mathcal{BOE}$ known as IND-BLK-CCA security[3]. Since $\mathcal{BOE}$ is an $\mathcal{ASOE}$ operated as a special case (when the plaintext-/ciphertext- streams have the same fixed size), it follows immediately that it is also *impossible* for $\mathcal{ASOE}$ to have IND-CCA security.

Below, we consider an attack that an IND-CCA-attacker can launch but no $\mathcal{ASOE}$ construction can defend against. The attack thus serves two purposes: (1) it proves that no $\mathcal{ASOE}$ can be IND-CCA-secure, and (2) it helps us define the strongest achieveable notion of data privacy.

3.1.1 A chosen-ciphertext attack

The attacker first initializes $\mathcal{E}$ with any IV v and gets in return a ciphertext header h. Next, the attacker chooses two 2-bit plaintext streams $m_{(0)} = \texttt{01}$ and $m_{(1)} = \texttt{11}$ and is given back (as part of the "challenge ciphertext") a 2-bit ciphertext stream $x_{(b)} = x_{(b)}[1]||x_{(b)}[2]$, which is the stream encryption of $m_{(b)}$, where $b = \texttt{0}$ or $\texttt{1}$ equally likely. In other words, the challenge ciphertext has the form $c = (h, x_{(b)}||\ldots, \cdot)$.

Now the attacker attempts to break IND-CCA-security by correctly guessing b[4] without asking $\mathcal{D}$ to decrypt c. To do so, he first initializes $\mathcal{D}$ with h. $\mathcal{ASOE}$'s completeness implies that such initialization of $\mathcal{D}$ will succeed. He then feeds $\mathcal{D}$ with the ciphertext stream $x_{(b)}[1]||\overline{x_{(b)}[2]}$ (i.e., $x_{(b)}$ with the second bit negated) and gets back the corresponding stream decryption $m'_{(b)} = m'_{(b)}[1]||m'_{(b)}[2]$.

[2]IND-CCA stands for *Ciphertext indistinguishability against adaptive chosen-ciphertext attacks*, a standard notion of data privacy for (non-on-line) symmetric encryption

[3]*Ciphertext Indistinguishability against Blockwise Chosen Ciphertext Attacks.*

[4]with probability non-negligibly better than random guessing

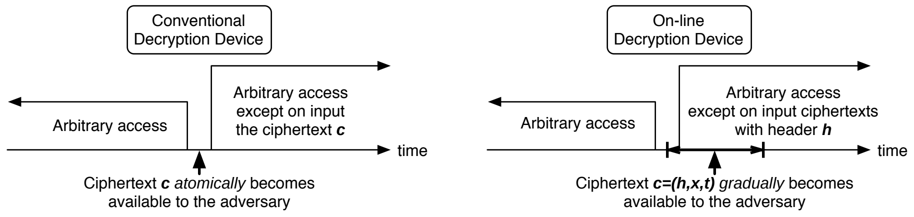
Conventional Decryption Device
Arbitrary access
Arbitrary access except on input the ciphertext c
time
Ciphertext c atomically becomes available to the adversary
On-line Decryption Device
Arbitrary access
Arbitrary access except on input ciphertexts with header h
time
Ciphertext c=(h,x,t) gradually becomes available to the adversary

Figure 1: To maintain the data privacy of a ciphertext, conventional symmetric encryption (left) and on-line encryption (right) impose different restrictions on the access to the decryption device.

$\mathcal{ASOE}$'s completeness implies that $m'_{(b)}[1] = m_{(b)}[1]$ always. Therefore, he can always make the correct guess of $b = m'_{(b)}[1]$.

In a nutshell, the attack above demonstrates that, for any $\mathcal{ASOE}$, once the header h of an $\mathcal{ASOE}$ ciphertext c becomes available to the attacker, he can always use $\mathcal{D}$ initialized with h to break the indistinguishability of c.

3.1.2 Strongest achievable data privacy in $\mathcal{ASOE}$

The strongest achievable data privacy in $\mathcal{ASOE}$ is hence IND-CCA-security, but with one restriction on the attacker's power:

> *The attacker may query $\mathcal{D}$ arbitrarily. Nevertheless, once the header h of the challenge ciphertext $c = (h, x, t)$ becomes known to the attacker, he is forbidden to use $\mathcal{D}$ to decrypt any ciphertext with the same header h.*

We call this notion of data privacy the *Ciphertext Indistinguishability against Streamwise Chosen-Ciphertext Attacks*, (IND-STR-CCA security). IND-STR-CCA security is strictly weaker than IND-CCA security: our own $\mathcal{ASOE}$ construction to be presented later in this paper is secure under IND-STR-CCA but not IND-CCA. Figure 1 constrasts the two security notions.

3.1.3 IND-STR-CCA v.s. IND-CCA

It might appear to some that IND-STR-CCA security provides much weaker data privacy than IND-CCA security does, and too quickly believe that an encryption scheme that is only IND-STR-CCA-secure — in particular, any $\mathcal{ASOE}$ — is of little use in practice. We refute such belief by closely examining the gap between the two notions below and, in Section 8, considering two real-world scenarios in which an IND-STR-CCA-secure encryption provides sufficient data privacy.

IND-STR-CCA security appears to be weaker because the attacker may manage to do the following with an IND-STR-CCA-secure encryption. Given a ciphertext $c = (h, x, t)$, he comes up with some other ciphertext c' with the same header h, i.e. $c' = (h, x', t')$ for some $(x', t') \neq (x, t)$, feeds the decryption device $\mathcal{D}$ with c', and somehow succeeds in cracking c.

Nevertheless, while it is true that the attacker may be able to crack c without feeding c into $\mathcal{D}$, the IND-STR-CCA security guarantees that to be able to crack c, the attacker must at least feed $\mathcal{D}$ with one ciphertext with the same header (i.e., c' in the above). As a result, IND-STR-CCA security offers a data-privacy guarantee that has little practical difference from what IND-CCA security offers. Specifically, to crack a ciphertext c, the attacker must still (temporarily) get hold

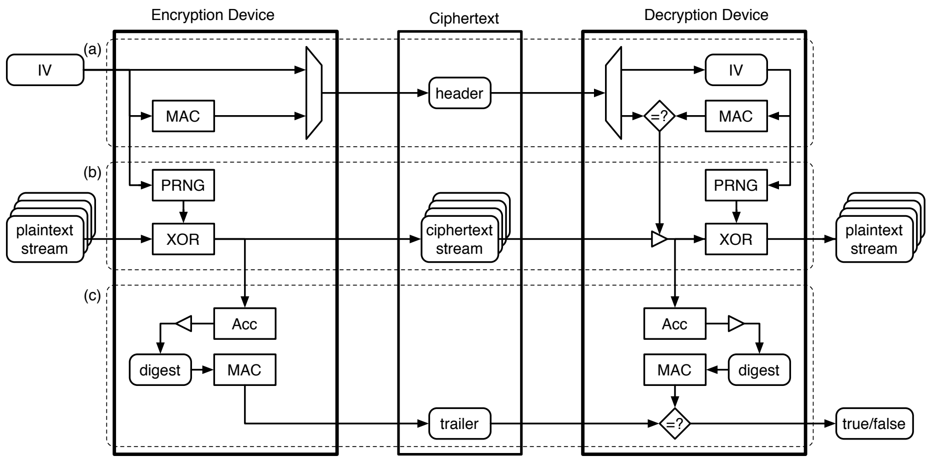

Figure 2: A block-diagram presentation of our $\mathcal{ASOE}$ construction. Encrypting a plaintext involves (a) an initialization step, (b) one or more iterations of encrypting a stream of the message, and (c) finalizing the encryption.

of $\mathcal{D}$ after c is given, and if $\mathcal{D}$ has a secure way of logging input ciphertexts, then we know exactly the set S of ciphertext(s) that may have been cracked — in case of IND-CCA security, S equals the set of all ciphertexts that the attacker has input to $\mathcal{D}$; in case of IND-STR-CCA security, S equals the set of all ciphertexts with the same header of any of the ciphertexts that the attacker has input to $\mathcal{D}$.

3.2 Data authenticity

We define the data authenticity requirement of $\mathcal{ASOE}$ as a variant of the notion of *"integrity of plaintexts"* (INT-PTXT security). INT-PTXT security was previously used to define data authenticity in (non-on-line) authenticated encryption [5]. It requires that if the decryption device $\mathcal{D}$ decides that the plaintext it outputs is authentic, then the plaintext must have been an input to the corresponding encryption device $\mathcal{E}$.

Intuitively, since $\mathcal{D}$ does not make the decision until it has received the entire ciphertext, the on-line nature of $\mathcal{ASOE}$ does not prevent $\mathcal{ASOE}$ from being INT-PTXT-secure. In fact, our $\mathcal{ASOE}$ construction to be presented in this paper does achieve INT-PTXT security. As we shall see, the real challenge has been how to achieve it along with other features, namely, zero buffering latency and constant message expansion and storage requirement.

4 An Overview of Our $\mathcal{ASOE}$ Construction

4.0.1 Achieving data privacy

As the skeleton of our $\mathcal{ASOE}$ construction, we use a secure stream cipher to provide some basic data privacy — our $\mathcal{ASOE}$ encryption device $\mathcal{E}$ uses the stream cipher to encrypt the plaintext to form the body of the ciphertext. The choice of a stream cipher is almost obvious: it readily gives us the streamwise on-line property, and does not violate the requirements of zero buffering latency and constant message expansion and storage.

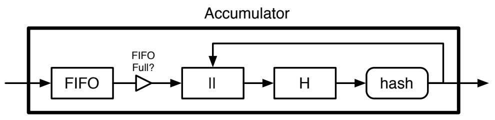

Figure 3: The accumulator fills the FIFO queue with input bits until it becomes full, and then concatenates its content with the current hash, updates the hash by hashing the concatenation, and finally empties the queue.

In our construction, the stream cipher is instantiated as XOR'ing the input plaintext stream with the key-stream output by a secure pseudorandom number generator.

See Figure 2(b).

A secure stream cipher in itself, however, is inadequate to provide IND-STR-CCA security, just like an CPA-secure symmetric encryption is inadequate to provide IND-CCA security. In particular, an attacker who gets hold of the corresponding $\mathcal{ASOE}$ decryption device $\mathcal{D}$ can learn the key-stream that will be used during an encryption *in the future*, and hence can break IND-STR-CCA security.

Our mechanism of using a ciphertext header fixes this problem. Specifically, the header h of a ciphertext $c = (h, x, t)$ consists of two components: the IV v used during the encryption, and a MAC a on v; $\mathcal{D}$ will decrypt c only if h is valid, where h is valid *if and only if* a is a valid MAC on v. The security of MAC assures that any valid h must have come from $\mathcal{E}$ without modification. Hence, $\mathcal{D}$ will reveal the key-stream for decrypting a ciphertext only if $\mathcal{E}$ has started outputting a ciphertext with the same header. Consequently, the attacker can't use $\mathcal{D}$ to learn a key-stream that will be used in the future.

See Figure 2(a).

4.0.2 Achieving data authenticity

To equip our $\mathcal{ASOE}$ construction with INT-PTXT security, we adopt the *"encrypt-then-mac"* approach [5], which was originally formalized as a generic method to construct authenticated encryption. The approach is simple: $\mathcal{E}$ symmetrically encrypts the plaintext m into ciphertext c and compute a MAC t on c; $\mathcal{D}$ decides that the plaintext into which c decrypts is authentic if t is a valid MAC on c.

There is, however, one issue to be resolved: $\mathcal{E}$ and $\mathcal{D}$ in $\mathcal{ASOE}$ only have a storage of constant size, i.e., independent of the longest possible plaintext/ciphertext, and hence can't store the entire ciphertext for computing its MAC. In our solution, instead of storing the incomplete ciphertext computed/received thus far, $\mathcal{E}$ and $\mathcal{D}$ store a constant-sized "digest" of it. When the next arbitrarily-sized ciphertext stream becomes available, $\mathcal{E}$ and $\mathcal{D}$ can incrementally update the digest. Finally, when the ciphertext is completely known, the MAC t is computed on its digest, instead of on the ciphertext itself.

We call the algorithm that produces a digest for a given (incomplete) ciphertext an *accumulator*. Obviously, if the accumulator is collision-resistant[5], such *"encrypt-then-accumulate-then-mac"* approach to data authentication remains INT-PTXT-secure.

See Figure 2(c).

It now suffices to find an accumulator for our $\mathcal{ASOE}$ construction. One possibility is this: given a ciphertext body x fragmented into $x = x_1||x_2||\ldots, x_\ell$, compute its digest as

[5]i.e., it is hard to find two inputs that get accumulated to the same digest

$H(\ldots H(H(x_1)||x_2)\ldots||x_\ell)$, where H is a collision-resistant hash function such as SHA-1. Computing the digest this way can obviously be done incrementally as new ciphertext streams become available.

Nonetheless, this method does not immediately work because the computed digest depends on how x is fragmented, which violates our zero buffering latency requirement. Therefore, in our construction, instead of inputting ciphertext streams to the hash function H directly, $\mathcal{E}$ and $\mathcal{D}$ first redirect them into a fixed-sized FIFO queue; every time the queue becomes full, $\mathcal{E}$ and $\mathcal{D}$ hash the content of the queue and then flush the queue. This way, $\mathcal{E}$ and $\mathcal{D}$ still use a constant amount of storage, and the digest of x is the same no matter how it is fragmented.

See Figure 6.

5 Formalizing $\mathcal{ASOE}$

We have already given a high-level description of the syntax and security of $\mathcal{ASOE}$ in Section 2 and 3 respectively. This section formalizes them.

5.1 Syntax

An *Authenticated Streamwise On-line Encryption* ($\mathcal{ASOE}$) scheme is a pair of an *encryption device* $\mathcal{E}$ and a *decryption device* $\mathcal{D}$. The encryption device $\mathcal{E}$ is a finite state machine that implements three algorithms: the *encryption initialization* algorithm $\mathcal{EI}$, the *stream encryption* algorithm $\mathcal{ES}$ and the *encryption finalizing* algorithm $\mathcal{EF}$. The decryption device $\mathcal{D}$ is a finite state machine that implements three other algorithms: the *decryption initialization* algorithm $\mathcal{DI}$, the *stream decryption* algorithm $\mathcal{DS}$ and the *decryption finalizing* algorithm $\mathcal{DF}$.

Let $\mathsf{Key} \doteq \{0,1\}^{\lambda_K}$, $\mathsf{IV} \doteq \{0,1\}^{\lambda_V}$, $\mathsf{State} \doteq \{0,1\}^{\lambda_S}$, $\mathsf{Header} \doteq \{0,1\}^{\lambda_H}$, $\mathsf{Trailer} \doteq \{0,1\}^{\lambda_T}$ and $\mathsf{Stream} \doteq \{0,1\}^*\backslash\{\varepsilon\}$, where λ_K, λ_V, λ_S, λ_H, $\lambda_T \in \mathbb{N}$ are security parameters. Below we describe the syntax of each algorithm.

- $\underline{\mathcal{EI} : \mathsf{Key} \times \mathsf{IV} \rightarrow \mathsf{State} \times \mathsf{Header}}$ On input a key $k \in \mathsf{Key}$ and an IV $v \in \mathsf{IV}$, the algorithm $\mathcal{EI}$ returns an initialized current state $s \in \mathsf{State}$ of $\mathcal{E}$, and a ciphertext header $h \in \mathsf{Header}$.

- $\underline{\mathcal{ES} : \mathsf{State} \times \mathsf{Stream} \rightarrow \mathsf{State} \times \mathsf{Stream}}$ On input $\mathcal{E}$'s current state $s \in \mathsf{State}$ and a plaintext stream m_i, the algorithm $\mathcal{ES}$ outputs $\mathcal{E}$'s new current state $s' \in \mathsf{State}$, and the corresponding ciphertext stream $x_i \in \mathsf{Stream}$.

- $\underline{\mathcal{EF} : \mathsf{State} \rightarrow \mathsf{Trailer}}$ On input the current state $s \in \mathsf{State}$ of $\mathcal{E}$, the algorithm $\mathcal{EF}$ outputs a ciphertext *trailer* $t \in \mathsf{Trailer}$.

- $\underline{\mathcal{DI} : \mathsf{Key} \times \mathsf{Header} \rightarrow \mathsf{State} \times \{\mathtt{true}, \mathtt{false}\}}$ On input a key $k \in \mathsf{Key}$ and a ciphertext header h, the algorithm $\mathcal{DI}$ returns an initialized current state $s \in \mathsf{State}$ of $\mathcal{D}$, and a boolean value $b \in \{\mathtt{true}, \mathtt{false}\}$.

- $\underline{\mathcal{DS} : \mathsf{State} \times \mathsf{Stream} \rightarrow \mathsf{State} \times \mathsf{Stream}}$ On input $\mathcal{D}$'s current state $s \in \mathsf{State}$ and an encrypted stream x_i, the algorithm $\mathcal{DS}$ outputs $\mathcal{D}$'s new current state $s' \in \mathsf{State}$, and the corresponding cipehertext stream $m_i \in \mathsf{Stream}$.

- $\underline{\mathcal{DF} : \mathsf{State} \times \mathsf{Trailer} \rightarrow \{0,1\}}$ On input the current state $s \in \mathsf{State}$ of $\mathcal{D}$ and a ciphertext trailer $t \in \mathsf{Trailer}$, the algorithm $\mathcal{DF}$ outputs a boolean value $b \in \{\mathtt{true}, \mathtt{false}\}$.

```
Oracle $\mathcal{O}_{\mathcal{EI}}(v)$
if $v \in \mathbf{V}$ then
  return $\perp$
else
  $\mathbf{V} \leftarrow \mathbf{V} \cup \{v\}$
  $(\mathbf{s}_{\mathcal{E}}, h) \leftarrow \mathcal{EI}(\mathbf{k}, v)$
  $\mathbf{f}_{\mathcal{E}} \leftarrow \text{true}$
  return $h$
```

```
Oracle $\mathcal{O}_{\mathcal{ES}}(m)$
if $\mathbf{f}_{\mathcal{E}} =$ false then
  return $\perp$
else
  $(\mathbf{s}_{\mathcal{E}}, c) \leftarrow \mathcal{ES}(\mathbf{s}_{\mathcal{E}}, m)$
  return $c$
```

```
Oracle $\mathcal{O}_{\mathcal{EF}}()$
if $\mathbf{f}_{\mathcal{E}} =$ false then
  return $\perp$
else
  $t \leftarrow \mathcal{EF}(\mathbf{s}_{\mathcal{E}})$
  $\mathbf{f}_{\mathcal{E}} \leftarrow$ false
  return $t$
```

```
Oracle $\mathcal{O}_{\mathcal{DI}}(h)$
$(\mathbf{s}_{\mathcal{D}}, \mathbf{f}_{\mathcal{D}}) \leftarrow \mathcal{DI}(\mathbf{k}, h)$
return $\mathbf{f}_{\mathcal{D}}$
```

```
Oracle $\mathcal{O}_{\mathcal{DS}}(c)$
if $\mathbf{f}_{\mathcal{D}} =$ false then
  return $\perp$
else
  $(\mathbf{s}_{\mathcal{D}}, m) \leftarrow \mathcal{DS}(\mathbf{s}_{\mathcal{D}}, c)$
  return $m$
```

```
Oracle $\mathcal{O}_{\mathcal{DF}}(t)$
if $\mathbf{f}_{\mathcal{D}} =$ false then
  return $\perp$
else
  $b \leftarrow \mathcal{DF}(\mathbf{s}_{\mathcal{D}}, t)$
  $\mathbf{f}_{\mathcal{D}} \leftarrow$ false
  return $b$
```

Figure 4: Various oracles modeling the adversary's capabilities when attempting to attack $\mathcal{ASOE}$. Variables in bold face are internal and persistent shared states.

Operation of the devices To $\mathcal{E}$, the very first operation and any operation immediately following an $\mathcal{EF}$ invocation must be an $\mathcal{EI}$ on invocation *on a new IV*, and any $\mathcal{EF}$ invocation must immediately follow an $\mathcal{ES}$ invocation. To $\mathcal{D}$, the very first operation and any operation immediately following a $\mathcal{DF}$ invocation must be a $\mathcal{DI}$ invocation during which $\mathcal{DI}$ output $(\cdot, \texttt{true})$, and any $\mathcal{DF}$ invocation must immediately follow a $\mathcal{DS}$ invocation.

Note that one may invoke $\mathcal{EI}$ (resp. $\mathcal{DI}$) at any time to abandon the current encryption (resp. decryption) and re-initialize the device for a new encryption (resp. decryption).

IV-explicit Encryption The encryption-related algorithms $\mathcal{EI}$, $\mathcal{ES}$ and $\mathcal{EF}$ and thus the entire process of encrypting a message in $\mathcal{ASOE}$ are a deterministic function on inputs the key, the IV and the plaintext. This is known as *IV-explicit encryption* in conventional symmetric encryption [16]. IV-explicit encryption such as $\mathcal{ASOE}$ has the advantage of not needing a cryptographic random number generator (RNG) in the encryption device.

5.2 Security

5.2.1 Adversarial Capabilities

The adversary has black-box access to a pair of an encryption device and a decryption device installed with the same key k. The six oracles $\mathcal{O}_{\mathcal{EI}}$, $\mathcal{O}_{\mathcal{ES}}$, $\mathcal{O}_{\mathcal{EF}}$, $\mathcal{O}_{\mathcal{DI}}$, $\mathcal{O}_{\mathcal{DS}}$ and $\mathcal{O}_{\mathcal{DF}}$ as defined in Figure 4 formally model the adversary's capability to operate the devices via the invocation of the algorithms $\mathcal{EI}$, $\mathcal{ES}$, $\mathcal{EF}$, $\mathcal{DI}$, $\mathcal{DS}$ and $\mathcal{DF}$ respectively.

We denote by $\{\mathcal{O}_{\mathcal{E}}\}$ the set of encryption-related oracles $\{\mathcal{O}_{\mathcal{EI}}(\cdot)$, $\mathcal{O}_{\mathcal{ES}}(\cdot)$, $\mathcal{O}_{\mathcal{EF}}()\}$, by $\{\mathcal{O}_{\mathcal{D}}\}$ the set of decryption-related oracles $\{\mathcal{O}_{\mathcal{DI}}(\cdot,\cdot)$, $\mathcal{O}_{\mathcal{DS}}(\cdot)$, $\mathcal{O}_{\mathcal{DF}}()\}$, and by $\{\mathcal{O}\}$ the set of all oracles $\{\mathcal{O}_{\mathcal{E}}\}$ $\cup$ $\{\mathcal{O}_{\mathcal{D}}\}$.

```
Experiment $\mathbf{Exp}^{\texttt{IND-STR-CCA}}_{\mathcal{ASOE}}(A)$
$\mathbf{k} \xleftarrow{\$} \mathsf{Key}$; $\mathbf{f}_{\mathcal{E}}, \mathbf{f}_{\mathcal{D}} \leftarrow \texttt{false}$; $\mathbf{s}_{\mathcal{E}}, \mathbf{s}_{\mathcal{D}} \leftarrow \varepsilon$
$\mathbf{V} \leftarrow \emptyset$; $\mathbf{s}_A \leftarrow \varepsilon$; $\mathbf{i} \leftarrow 0$
$b \xleftarrow{\$} \{0,1\}$
$(\hat{v}, \mathbf{s}_A) \leftarrow A^{\{\mathcal{O}\}}(\texttt{find}_0, \mathbf{s}_A)$; $\hat{h} \leftarrow \mathcal{O}_{\mathcal{EI}}(\hat{v})$
$(\hat{m}^{(0)}_1, \hat{m}^{(1)}_1, \mathbf{s}_A) \leftarrow$
            $A^{\{\mathcal{O}\}\setminus\{\mathcal{O}_{\mathcal{DI}}(\hat{h})\}}(\texttt{find}_1, \hat{h}, \mathbf{s}_A)$

repeat
  $\mathbf{i} \leftarrow \mathbf{i} + 1$
  if $|\hat{m}^{(0)}_{\mathbf{i}}| \neq |\hat{m}^{(1)}_{\mathbf{i}}|$ then
    return 0
  $\hat{c}_{\mathbf{i}} \leftarrow \mathcal{O}_{\mathcal{ES}}(\hat{m}^{(b)}_{\mathbf{i}})$
  $(\hat{m}^{(0)}_{i+1}, \hat{m}^{(1)}_{i+1}, \mathbf{s}_A, \mathbf{e}) \leftarrow$
            $A^{\{\mathcal{O}\}\setminus\{\mathcal{O}_{\mathcal{DI}}(\hat{h})\}}(\texttt{find}_{\mathbf{i}+1}, \hat{c}_{\mathbf{i}}, \mathbf{s}_A)$
until $\mathbf{e} = 1$

$\hat{t} \leftarrow \mathcal{O}_{\mathcal{EF}}()$
$\hat{b} \leftarrow A^{\{\mathcal{O}\}\setminus\{\mathcal{O}_{\mathcal{DI}}(\hat{h})\}}(\texttt{guess}, \hat{t}, \mathbf{s}_A)$
if $\hat{b} = b$ then
  return 1
else
  return 0
```

```
Experiment $\mathbf{Exp}^{\texttt{STR-INT-PTXT}}_{\mathcal{ASOE}}(A)$
$\mathbf{k} \xleftarrow{\$} \mathsf{Key}$; $\mathbf{f}_{\mathcal{E}}, \mathbf{f}_{\mathcal{D}} \leftarrow \texttt{false}$; $\mathbf{s}_{\mathcal{E}}, \mathbf{s}_{\mathcal{D}} \leftarrow \varepsilon$
$\mathbf{V} \leftarrow \emptyset$; $\mathbf{s}_A \leftarrow \varepsilon$; $\mathbf{i} \leftarrow 0$
$(\hat{h}, \hat{c}_1, \mathbf{s}_A) \leftarrow A^{\{\mathcal{O}\}}(\texttt{forge}_0, \mathbf{s}_A)$
$\hat{b}_0 \leftarrow \mathcal{O}_{\mathcal{DI}}(\hat{h})$
if $\hat{b}_0 =$ false then
  return 0

repeat
  $\mathbf{i} \leftarrow \mathbf{i} + 1$
  $\hat{m}_{\mathbf{i}} \leftarrow \mathcal{O}_{\mathcal{DS}}(\hat{c}_{\mathbf{i}})$
  $(\hat{c}_{\mathbf{i}+1}, \mathbf{s}_A, \mathbf{e}) \leftarrow A^{\{\mathcal{O}_{\mathcal{E}}\}}(\texttt{forge}_{\mathbf{i}}, \hat{m}_{\mathbf{i}}, \mathbf{s}_A)$
until $\mathbf{e} = 1$

$\hat{t} \leftarrow A^{\{\mathcal{O}_{\mathcal{E}}\}}(\texttt{forge}, \mathbf{s}_A)$; $\hat{b} \leftarrow \mathcal{O}_{\mathcal{DF}}(\hat{t})$
if $\hat{b} = 0$ then
  return 0

$\hat{m} \leftarrow \hat{m}_1 || \hat{m}_2 || \ldots || \hat{m}_{\mathbf{i}}$
denote the sequence of $\{\mathcal{O}_{\mathcal{E}}\}$
  queries by:
  $\Big(\mathcal{O}_{\mathcal{EI}}(\cdot), (\mathcal{O}_{\mathcal{ES}}(m_{ij}))_{j=1}^{L_i}, \mathcal{O}_{\mathcal{EF}}()\Big)_{i=1}^{L}$
$m_i \leftarrow m_{i1} || m_{i2} || \ldots || m_{iL_i}, \forall i \in [1, L]$
if $\forall i \in [1, L], m_i \neq \hat{m}$ then
  return 1
else
  return 0
```

Figure 5: Experiments defining $\mathcal{ASOE}$'s data privacy (left) and data authenticity (right).

5.2.2 Data Privacy

We define data privacy in $\mathcal{ASOE}$ as follows.

Definition 1 (IND-STR-CCA) *An $\mathcal{ASOE}$ construction has* ciphertext-indistinguishability against streamwise-adaptive-chosen-ciphertext attacks *(IND-STR-CCA security) if, for all PPT adversary A, the* advantage *of A defined as*

$$\mathbf{Adv}^{\textit{IND-STR-CCA}}_{\mathcal{ASOE}}(A) = 2 \cdot \Pr\left[1 \leftarrow \mathbf{Exp}^{\textit{IND-STR-CCA}}_{\mathcal{ASOE}}(A)\right] - 1$$

is a negligible function in the security parameter λ.

In the above, the experiment $\mathbf{Exp}^{\texttt{IND-STR-CCA}}_{\mathcal{ASOE}}(A)$ is listed in Figure 5. Very briefly, it proceeds as follows. The adversary first picks an IV of his choice to initialize the encryption device and gets back the resulting ciphertext header. The adversary then finds a pair of two equal-length streams

$\hat{m}_1^{(0)}$ and $\hat{m}_1^{(1)}$ of his choice and gets back from the encryption device the ciphertext stream $\hat{c}_1$ of $\hat{m}_1^{(b)}$, where $b = 0$ or 1 with equal probability. He may then keep finding another pair of two equal-length streams $\hat{m}_i^{(0)}$ and $\hat{m}_i^{(1)}$ of his choice, potentially adaptive to the value of the previous ciphertext streams, thereby getting back the ciphertext stream $\hat{c}_i$ of $\hat{m}_i^{(b)}$. (Note that b is the same throughout.) Eventually, the adversary declares that he has finished inputting the two plaintexts and gets back from the encryption device the ciphertext trailer. $\mathcal{ASOE}$ has data privacy if the adversary cannot correctly tell what b was better than random guessing.

5.2.3 Data Authenticity

We define data authenticity of $\mathcal{ASOE}$ as follows.

Definition 2 (STR-INT-PTXT) *An $\mathcal{ASOE}$ construction has* streamwise integrity of plaintexts *(STR-INT-PTXT security) if, for all PPT adversary A, the* advantage *of A defined as*

$$\mathbf{Adv}_{\mathcal{ASOE}}^{\textit{STR-INT-PTXT}}(A) = \Pr\left[1 \leftarrow \mathbf{Exp}_{\mathcal{ASOE}}^{\textit{STR-INT-PTXT}}(A)\right]$$

is a negligible function in the security parameter λ.

In the above, the experiment $\mathbf{Exp}_{\mathcal{ASOE}}^{\texttt{STR-INT--PTXT}}(A)$ is listed in Figure 5. Very briefly, it proceeds as follows. The adversary is given arbitrary access to the encryption and decryption devices until he decides to start forging a ciphertext by providing a ciphertext header and the first forged ciphertext stream. From this moment on, he is restricted to only access the encryption device. He fails immediately if the header is determined to be unauthentic by the decryption device when initialized with the IV and the header. Otherwise, the forged ciphertext stream is input the the decryption device and the corresponding output is given to the adversary. He may keep inputting a forged ciphertext stream to the decryption devices and gets back the corresponding decryption. The adversary finishes the forging by returning a forged trailer. The adversary fails immediately if the trailer is determined by the decryption device as invalid. Otherwise the forged ciphertext is authentic. Now the adversary has successfully attacked data authenticity of $\mathcal{ASOE}$ if the corresponding original message of the forged ciphertext as decrypted by the decryption device has *never* been input to the encryption device during the experiment.

5.2.4 Relations to Other Schemes

A secure $\mathcal{ASOE}$ construction trivially implies a secure $\mathcal{AE}$ (authenticated encryption) construction, as well as a secure $\mathcal{ABOE}$ (authenticated blockwise on-line encryption) construction. Specifically, if we require that there is at most one $\mathcal{ES}$ (resp. one $\mathcal{DS}$) immediately following an $\mathcal{EI}$ (resp. a $\mathcal{DI}$) in a secure $\mathcal{ASOE}$ construction, we obtain a secure $\mathcal{AE}$ construction. If we define Stream as $\{0,1\}^B$ instead of $\{0,1\}^*\backslash\{\varepsilon\}$, a secure $\mathcal{ASOE}$ construction becomes a secure $\mathcal{ABOE}$ construction, with a block-size of B bits.

Also, a secure $\mathcal{ASOE}$ construction becomes a secure $\mathcal{SOE}$ construction, if one ignores the output of $\mathcal{DF}$. By similar arguments, a secure $\mathcal{ASOE}$ construction also implies a secure symmetric encryption construction and a secure $\mathcal{BOE}$ construction.

```
Algorithm $\mathsf{Acc.Init}()$
1:  $\mathbf{buf}, \mathbf{md} := \epsilon$

Algorithm $\mathsf{Acc.Get}()$
2:  return $\mathsf{H}(\mathbf{md}||\mathbf{buf})$

Algorithm $\mathsf{Acc.Add}(b)$
3:  $\mathbf{buf} := \mathbf{buf}||b$
4:  if $|\mathbf{buf}| = B$ then
5:      $\mathbf{md} := \mathsf{H}(\mathbf{md}||\mathbf{buf})$
6:      $\mathbf{buf} := \epsilon$
```

Figure 6: The algorithms that constitute to our Acc construction.

6 Details of Our $\mathcal{ASOE}$ Construction

6.1 Building Blocks

Pseudorandom Number Generator A Pseudorandom Number Generator PRNG provides two algorithms PRNG.Init() and PRNG.Get. To initialize PRNG, one invokes PRNG.Init() with a key and a seed. Then, each invocation of PRNG.Get(ℓ) returns a string of ℓ pseudorandom bits. Our $\mathcal{ASOE}$ construction utilizes a secure PRNG [8, 15].

Message Authentication Scheme A message authentication scheme MA provides an algorithm MA.MAC, which computes an authentication tag on input a key and a message. Our $\mathcal{ASOE}$ construction uses a secure MA [3].

Accumulator As have explained in Section 4, our $\mathcal{ASOE}$ construction uses a secure accumulator Acc. Here we explain how to build one. Acc maintains two variables **buf** and **md** as persistent states. **buf** is a B-bit buffer that acts as a FIFO queue, and **md** is the digest of the bit-stream added into the accumulator so far. Acc provides three algorithms Acc.Init, Acc.Add and Acc.Get defined in Figure 6.

6.2 The Encryption and Decryption Devices

6.2.1 Key Generation

In our $\mathcal{ASOE}$ construction, the encryption/decryption key k is a triple of (k_H, k_E, k_T), where k_H and k_T are MAC keys, and k_E is a PRNG key. Thus, one generates an $\mathcal{ASOE}$ key by generating a PRNG key and two MAC keys, which can in turn be done by picking random binary strings of appropriate length.

6.2.2 The Encryption and Decryption Devices

In figure 7, we list the algorithms that together constitute to our $\mathcal{ASOE}$ construction. It is trivial to see that our construction has completeness, zero buffering latency, and constant message expansion and constant storage size.

6.3 Security

We state the following theorem regarding the security of our $\mathcal{ASOE}$ construction and sketch its proof in Appendix A.

```
Algorithm $\mathcal{EI}(k, v)$
1: $(k_H, k_E, k_T) := k$
2: $\mathsf{PRNG.Init}(k_E, v)$
3: $\mathsf{Acc.Init}()$
4: $a := \mathsf{MA.MAC}(k_H, v)$
5: return $h := (v, a)$
```

```
Algorithm $\mathcal{ES}(k, m)$
6: $(k_H, k_E, k_T) := k$
7: $s := \mathsf{PRNG.Get}(|m|)$
8: $x := s \oplus m$
9: for $j = 1$ to $|x|$ do
10:   $\mathsf{Acc.Add}(x[j])$
11: return $x$
```

```
Algorithm $\mathcal{EF}(s)$
12: $(k_H, k_E, k_T) := k$
13: $digest := \mathsf{Acc.Get}()$
14: $t := \mathsf{MA.MAC}(k_T, digest)$
15: return $t$
```

```
Algorithm $\mathcal{DI}(k, h)$
16: $(k_H, k_E, k_T) := k$; $(v, a) := h$
17: $\mathsf{PRNG.Init}(k_E, v)$
18: $\mathsf{Acc.Init}()$
19: $a' := \mathsf{MA.MAC}(k_H, v)$
20: return $a \overset{?}{=} a'$
```

```
Algorithm $\mathcal{DS}(k, x)$
21: $(k_H, k_E, k_T) := k$
22: $s := \mathsf{PRNG.Get}(|x|)$
23: $m := s \oplus x$
24: for $j = 1$ to $|x|$ do
25:   $\mathsf{Acc.Add}(x[j])$
26: return $m$
```

```
Algorithm $\mathcal{DF}(s, t)$
27: $(k_H, k_E, k_T) := k$
28: $digest := \mathsf{Acc.Get}()$
29: $t' := \mathsf{MA.MAC}(k_T, digest)$
30: return $t \overset{?}{=} t'$
```

Figure 7: The algorithms that constitute to our $\mathcal{ASOE}$ construction.

Theorem 1 (Security) *Our construction of the $\mathcal{ASOE}$ encryption scheme is IND-STR-CCA-secure and STR-INT-PTXT-secure if the underlying* PRNG*,* MA *and H are secure.*

7 Performance Evaluation

7.1 Instantiation

We instantiate PRNG with the AES block cipher with 128-bit keys (thus, $|k_E| = 128$). As a result, XOR'ing the plaintext/ciphertext streams with a PRNG output is precisely operating AES in counter mode. We instantiate MAC with HMAC-SHA-1 with 160-bit keys (thus, $|k_H| = |k_T| = 160$). Also HMAC-SHA-1 outputs are 160-bit long. We choose IVs to be 64-bit long, which should prevent the need of device re-keying due to the running out of fresh IVs.

We instantiate the hash function H in the accumulator Acc with SHA-1. We choose the buffer size B to be $512 - 160 = 352$ bits for the following reason. SHA-1 has a block-size of 512 bits and output-size of 160 bits. A buffer size of 352 bits makes sure that $\mathsf{H}(\mathbf{md}||\mathbf{buf})$ will always hash an input at most one block in size.

7.1.1 Hardware Costs

When implemented in hardware such as FPGA, each of the $\mathcal{ASOE}$ encryption and decryption devices have one AES core (a.k.a., coprocessor or engine) and one HMAC-SHA-1 core[6], in addition to some controller logic and registers. None of these components has size (in terms of, e.g., gate counts) dependent on the maximum allowable size of plaintext/ciphertext. Furthermore, no (cryptographic) RNG is needed.

7.2 Message Expansion

The ciphertext body has the same size as the plaintext. The message expansion is thus due solely to the ciphertext header and trailer. The header is $64 + 160 = 224$ bits, and the trailer is 160 bits. The message expansion is hence $224 + 160 = 384$ bits, or *48 bytes*.

7.3 End-to-end Latency

The total end-to-end latency (t_T) incurred by $\mathcal{ASOE}$ is measured by the latency due to buffering ($t_B^{(\mathcal{E})}$) and processing ($t_P^{(\mathcal{E})}$) at the encryption device, the latency due to buffering ($t_B^{(\mathcal{D})}$) and processing ($t_P^{(\mathcal{D})}$) at the decryption device, and the latency due to message expansion (t_E), i.e.,

$$t_T = t_B^{(\mathcal{E})} + t_P^{(\mathcal{E})} + t_B^{(\mathcal{D})} + t_P^{(\mathcal{D})} + t_E.$$

The latency due to message expansion t_E is *48 byte times*, where 1 byte time denotes the time it takes for the communication channel to send 1 byte. There is *zero* latency due to buffering at both devices, i.e., $t_B^{(\mathcal{E})} = t_B^{(\mathcal{D})} = 0$.

The latencies due to processing ($t_P^{(\mathcal{E})}$ and $t_P^{(\mathcal{D})}$) require a bit more analysis.

$\mathcal{ES}$ and $\mathcal{DS}$ involve (1) generating the key-stream, (2) XOR'ing two bit-streams and (3) adding input bits into the accumulator. The key-stream generation can be precomputed and is hence *not* on the critical path. As long as the generation rate can keep up with the rate at which plaintext/ciphertext streams are fed into the devices, key-stream generation incurs no non-hideable latency. For example, there are commercial AES cores for Gigabit Ethernet (and hence have throughput up to 1 Gbit/s). On the other hand, the XOR'ing operation is on the critical path, but only incurs a negligible latency. Finally, accumulating the input bits invokes one HMAC-SHA-1 operation per $B = 352$ bits. Again, using a core with sufficiently high throughput, one can hide the incurred latency.

Each of $\mathcal{EI}$, $\mathcal{DI}$, $\mathcal{EF}$ and $\mathcal{DF}$ essentially involves one non-precomputable invocation of HMAC-SHA-1 on an input no more than one HMAC-SHA-1 block in size, effectively incurring a non-hideable latency roughly equivalent to two SHA-1 operations, which is again insignificantly small with a fast HMAC-SHA-1 core.

Now that we have argued that $t_P^{(\mathcal{E})}$ and $t_P^{(\mathcal{D})}$ are insignificantly small, the total end-to-end latency t_T is thus dominated by the latency due to message expansion $t_E = 48$ byte times.

[6]One HMAC-SHA-1 is sufficient because the computation of the MAC in the header does not overlap in time with that in the trailer.

7.4 Further Optimizations

In certain applications such as those in which ciphertexts are transmitted in order with little loss, the decryption device often can correctly predict the IV used in an incoming ciphertext. One could thus reduce message expansion and hence also the latency it incurs by omitting sending the IVs (most of the time, and sending them only occasionally during idle times for synchronization).

Alternatively, if the communication protocol encapsulates each ciphertext into a packet and uses sequence numbers to enable out-of-order delivery of packets, one can use the sequence numbers as IVs and hence again omit sending the IVs, as long as the sequence numbers never repeat before device re-keying.

To further reduce message expansion, one could use shorter IVs such as 32 bits (potentially at the expense of more frequent device re-keying) and MAC with shorter outputs such as HMAC-SHA-1-96[7]. The message expansion is reduced from 48 bytes by 20 bytes to *28 bytes* instead. As a result, the end-to-end latency is also reduced by 20 byte times to roughly *28 byte times*.

8 Application Scenarios

In this section, we provides two application scenarios that can benefit from $\mathcal{ASOE}$. We then point out one caveat when using $\mathcal{ASOE}$.

8.1 Scenario I: Security Retrofit for Legacy SCADA Communications

Supervisory Control And Data Acquisition (SCADA) systems are process control systems used to monitor and control devices in critical infrastructures such as the electric power grid, of which many are legacy, use low-bandwidth links (e.g., 9600 baud/sec leased telephone lines) and are vulnerable to attacks such as traffic eavesdropping and tampering.

In the *"Bump-In-The-Wire" (BITW)* approach to secure the communications in such existing insecure legacy SCADA systems, two hardware BITW modules are inserted into the link connecting two communicating SCADA devices, one next to each device. Via encryption and data authentication, these modules retrofit security to the communications *transparently*, except that they incur end-to-end communication latency due to processing and buffering.

Buffering alone, however, can incur a prohibitively high latency in low-bandwidth links, rendering a BITW solution inapplicable to secure SCADA communications with stringent latency constraints [11]. In fact, most existing BITW solutions do not provide sufficient security within timing constraints. For example, the SCADA Cryptographic Module (SCM) [18] can be classified as an $\mathcal{ABOE}$, but has only 16-bit security under a threat model too weak to be realistic. YASIR [17] operates as an $\mathcal{ASOE}$, and is hence the most related work to our $\mathcal{ASOE}$ construction. YASIR does provide data authenticity with high security assurance, but its security is proved with assumptions about the specifics of the application and the communication protocols.

Therefore, BITW modules built with our $\mathcal{ASOE}$ can retrofit sufficient security to legacy SCADA links with a minimal impact on end-to-end communication latency. The BITW module at the recipient end — which encloses the $\mathcal{ASOE}$ decryption device — passes on the decrypted plaintext streams to the recipient SCADA device without buffering them, and informs the SCADA device immediately if the plaintext turns out to be unauthentic, in which case the SCADA device drops the plaintext and never acts on it.

[7]HMAC-SHA-1-96 is HMAC-SHA-1, but with outputs truncated to the first 96 bits [14].

8.2 Scenario II: Secure and Real-time Wireless Sensor Networks

A class of *Wireless Sensor Networks (WSNs)* demand not only security (data integrity, and/or data confidentiality), but also high timeliness (real-time or close to real-time) in data collection, such as those for fire monitoring [19], border surveillance, medical care, and highway traffic coordination [7].

In such applications, sensors often need to report readings as short as 4-byte integers or even 1-bit boolean values. When $(\mathcal{A})\mathcal{BOE}$ is used to secure the transmission of those readings, one would need to either pad a reading into a 128-bit AES block or aggregate multiple readings over time. The former approach wastes bandwidth and thus energy of the sensors; the latter one can violate the real-time requirements as readings are not immediately sent.

$\mathcal{ASOE}$ provides the security without padding or aggregation.

8.3 A Caveat on early availability of potentially unauthentic data

The decryption device in $\mathcal{ASOE}$ (and $\mathcal{ABOE}$) can only decide on the authenticity of the output plaintext *afterwards*, any application that exploits the on-line feature and make "early" use of the output plaintext streams must thus use caution. This also means $\mathcal{ASOE}/ABOE$ is not a more-efficient replacement of Æ, and some applications shouldn't use $\mathcal{ASOE}/ABOE$ for security reasons. We illustrate this with an example below.

Consider a server streaming the stock quotes of the NASDAQ stock market to its clients. Authenticated encryption can be used to provide the desired data privacy (only subscribers can read) and data authenticity (the quotes haven't been tampered during transit). One might, however, want to use $\mathcal{ASOE}$ to take advantage of its earlier data availability: as the individual stock quotes arrive, they can immediately be displayed to the client, without having to wait until the complete download of the entire list of quotes. Such use of $\mathcal{ASOE}$, however, could be dangerous because the authenticity of these "early-displayed" quotes can't be verified until later, before which the client may be tempted to quickly execute a trade based on the potentially false information.

9 Conclusions

We have introduced *Authenticated Streamwise On-line Encryption* ($\mathcal{ASOE}$), the first authenticated encryption scheme that has *zero* buffering latency, and *constant* message expansion and storage requirement, and can thus efficiently secure resource-constrained communications with stringent real-time requirements.

We have investigated and formalized the strongest achievable notion of security for $\mathcal{ASOE}$, provided an $\mathcal{ASOE}$ construction secure under such a notion. An instantiation using AES and HMAC-SHA-1 incurs only 48 bytes of message expansion.

References

[1] G. V. Bard. Blockwise-Adaptive Chosen-Plaintext Attack and Online Modes of Encryption. In *IMA Int. Conf.*, volume 4887 of *LNCS*, pages 129–151. Springer, 2007.

[2] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *FOCS*, pages 394–403, 1997.

[3] M. Bellare, J. Kilian, and P. Rogaway. The Security of Cipher Block Chaining. In *CRYPTO*, volume 839 of *LNCS*, pages 341–358. Springer, 1994.

[4] M. Bellare, T. Kohno, and C. Namprempre. Authenticated Encryption in SSH: Provably Fixing the SSH Binary Packet Protocol. In *ACM Conference on Computer and Communications Security*, pages 1–11. ACM, 2002.

[5] M. Bellare and C. Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In *ASIACRYPT*, volume 1976 of *LNCS*, pages 531–545. Springer, 2000.

[6] A. Boldyreva and N. Taesombut. Online Encryption Schemes: New Security Notions and Constructions. In *CT-RSA*, volume 2964 of *LNCS*, pages 1–14. Springer, 2004.

[7] P. Chen, S. Oh, M. Manzo, B. Sinopoli, C. Sharp, K. Whitehouse, O. Tolle, J. Jeong, P. Dutta, J. Hui, S. Schaffert, S. Kim, J. Taneja, B. Zhu, T. Roosta, M. Howard, D. Culler, and S. Sastry. Instrumenting Wireless Sensor Networks for Real-time Surveillance. *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 3128–3133, 0-0 0.

[8] A. Desai and S. K. Miner. Concrete Security Characterizations of PRFs and PRPs: Reductions and Applications. In *ASIACRYPT*, volume 1976 of *Lecture Notes in Computer Science*, pages 503–516. Springer, 2000.

[9] P.-A. Fouque, A. Joux, G. Martinet, and F. Valette. Authenticated On-Line Encryption. In *Selected Areas in Cryptography*, volume 3006 of *LNCS*, pages 145–159. Springer, 2003.

[10] P.-A. Fouque, G. Martinet, and G. Poupard. Practical Symmetric On-Line Encryption. In *FSE*, volume 2887 of *LNCS*, pages 362–375. Springer, 2003.

[11] IEEE Standard Communication Delivery Time Performance Requirements for Electric Power Substation Automation. IEEE Std 1646-2004, 2005.

[12] A. Joux, G. Martinet, and F. Valette. Blockwise-Adaptive Attackers: Revisiting the (In)Security of Some Provably Secure Encryption Models: CBC, GEM, IACBC. In *CRYPTO*, volume 2442 of *LNCS*, pages 17–30. Springer, 2002.

[13] NIST. FIPS 197: Announcing the ADVANCED ENCRYPTION STANDARD (AES). Technical report, National Institute of Standards and Technology (NIST), 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[14] NIST. FIPS 198: The Keyed-Hash Message Authentication Code (HMAC). Technical report, National Institute of Standards and Technology (NIST), 2002. http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf.

[15] D. H. Phan and D. Pointcheval. About the Security of Ciphers (Semantic Security and Pseudo-Random Permutations). In *Selected Areas in Cryptography*, volume 3357 of *LNCS*, pages 182–197. Springer, 2004.

[16] P. Rogaway. Nonce-Based Symmetric Encryption. In *FSE*, volume 3017 of *LNCS*, pages 348–359. Springer, 2004.

[17] P. P. Tsang and S. W. Smith. YASIR: A Low-Latency, High-Integrity Security Retrofit for Legacy SCADA Systems. In *23rd International Information Security Conference (IFIP SEC 2008)*, IFIP. Springer, 2008.

[18] A. K. Wright, J. A. Kinast, and J. McCarty. Low-Latency Cryptographic Protection for SCADA Communications. In *ACNS*, volume 3089 of *LNCS*, pages 263–277. Springer, 2004.

[19] L. Yu, N. Wang, and X. Meng. Real-time Forest Fire Detection with Wireless Sensor Networks. *Wireless Communications, Networking and Mobile Computing, 2005. Proceedings. 2005 International Conference on*, 2:1214–1217, 23-26 Sept. 2005.

A Proof of Theorem 1 (Sketch)

A.1 Data Privacy

Assume that there exists a probabilistic poly-time (PPT) adversary A such that the experiment $\mathbf{Exp}^{\texttt{IND-STR-CCA}}_{\mathcal{ASOE}}(A)$ returns 1 with non-negligible probability, we show how to construct an PPT simulator S that uses A to break the security of PRNG or MA with non-negligible probability.

Case I. In an attempt to break the security of MA, S obtains a ciphertext header from A and uses it to come up with a valid authentication tag on a string it has never queried. S first sets up the experiment. Note that the MA problem instance implicitly fixes k_H in the $\mathcal{ASOE}$ key and S does not know k_H. During the experiment, S simulates all oracles honestly, except those that involve invoking $\mathsf{MA.MAC}$, which S relays to the $\mathsf{MA.MAC}$ oracle. This way, if A ever queries $\mathcal{O}_{\mathcal{DI}}$ with a valid header $h = (v, a)$ such that v has never been an input to $\mathcal{O}_{\mathcal{EI}}$ (we call this *condition 1*), S can always use the pair (v, a) to solve the MA problem instance.

Case II. In an attempt to break the security of PRNG, S's goal is to distinguish the output of $\mathsf{PRNG.Get}(\ell)$ from a random ℓ-bit string. S first sets up the experiment. Note that the PRNG problem instance implicitly fixes k_E in the $\mathcal{ASOE}$ key and S does not know k_E. During the experiment, S simulates all oracles honestly for A, except those that involves invoking $\mathsf{PRNG.Get}(\cdot)$, which S relays to the $\mathsf{PRNG.Get}(\cdot)$ oracle. At some point during the experiment, A submits two non-null challenge plaintext message streams $m_1^{(0)}$ and $m_1^{(1)}$ of equal length ℓ_1^*. This occurs at least once. We consider the first occurrence. S challenges the PRNG problem instance and get back a string s, where s is either $\mathsf{PRNG.Get}(\ell)$ or a random ℓ-bit string with equal probability. S uses this as the key-stream for the encryption of the entire challenge ciphertext. In other words, the S XOR's s with $m_i^{(b)}$, where b is a fair coin flip. Note that S does not know at this point the length ℓ of the key stream that will be needed, but the simulation will work as long as S chooses ℓ to be no less the length of the challenge plaintext. If *condition 1* does *not* hold, then S has never query the $\mathsf{PRNG.Get}()$ oracle initialized with the same key and seed. If eventually A makes the right guess, then S guesses that s is indeed $\mathsf{PRNG.Get}(\ell)$. Otherwise, S guesses that s is $\mathsf{PRNG.Get}(\ell)$ with a probability of $1/2$.

The success probability of S is calculated as follows. If s is indeed $\mathsf{PRNG.Get}(\ell)$, then A has a non-negligible probability of guessing b right, and S has the same probability of guessing that s is $\mathsf{PRNG.Get}(\ell)$.

Combining Two Cases. Since S's simulation is correct in both cases, A can't correctly tell which cases it is during the experiment better than random guessing. Therefore, if *condition 1* holds, S breaks MA with non-negligible probability; if *condition 1* does not hold, S breaks PRNG with non-negligible probability.

A.2 Data Authenticity

Assume that there exists an PPT algorithm A such that the experiment $\mathbf{Exp}^{\texttt{STR-INT-PTXT}}_{\mathcal{ASOE}}(A)$ return 1 with non-negligible probability, we show how to construct an PPT simulator S that uses A to break the security of MA or H with non-negligible probability.

S sets up the experiment by generating only k_H and k_E in the $\mathcal{ASOE}$ key; S does not know k_T. S simulates all oracles honestly, except that it must relay all $\mathsf{MA.MAC}$ computation for ciphertext trailers to the $\mathsf{MA.MAC}$ oracle. If the experiment eventually returns with 1, then A must have given S a ciphertext $c = (h, x, t)$ that decrypts to m and such that y is a digest of accumulating x

and $\mathsf{MA.MAC}(y) = t$. S can use the pair (y, t) as an answer to the MA problem instance. Now it suffices to show that S had queried the $\mathsf{MA.MAC}$ oracle on input y with negligible probability.

Assume the contrary that S queried the $\mathsf{MA.MAC}$ oracle on input y with non-negligible probability, then it must have happened either when S was simulating $\mathcal{O}_{\mathcal{EF}}$ or $\mathcal{O}_{\mathcal{DF}}$. If it was during an $\mathcal{O}_{\mathcal{EF}}$ simulation, then A must have used the encryption oracle to encrypt a message m' such that y is the digest of $m' \oplus s$, where s is the corresponding key-stream. Due to the collision-resistance of Acc and the unforgeability of the MA used to compute ciphertext headers, $m \neq m'$ with negligible probability, which contradicts to the fact that A had won the game. On the other hand, if it was during an $\mathcal{O}_{\mathcal{DF}}$ simulation, one can show that the same had happened during an $\mathcal{O}_{\mathcal{DF}}$ simulation too, with non-negligible probability. The result follows.