9-24-2007

YASIR: A Low-Latency, High-Integrity Security Retrofit for Legacy SCADA Systems

Patrick P. Tsang
Dartmouth College

Sean W. Smith
Dartmouth College

YASIR: A Low-Latency, High-Integrity Security Retrofit for Legacy SCADA Systems*

Patrick P. Tsang† and Sean W. Smith‡

Department of Computer Science
Dartmouth College
NH 03755 USA

Dartmouth Computer Science Technical Report TR2007-603

September 24, 2007

Abstract

We construct a bump-in-the-wire (BITW) solution that retrofits security into time-critical communications over bandwidth-limited serial links between devices in Supervisory Control And Data Acquisition (SCADA) systems. Previous BITW solutions fail to provide the necessary security within timing constraints; the previous solution that does provide the necessary security is not BITW. At a comparable hardware cost, our BITW solution provides sufficient security, and yet incurs minimal end-to-end communication latency. A microcontroller prototype of our solution is under development.

*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.

†patrick@cs.dartmouth.edu

‡sws@cs.dartmouth.edu

Contents

1 Introduction

1.1 SCADA Systems

Supervisory Control And Data Acquisition (SCADA) systems are real-time process control systems that monitor and control local or geographically remote devices. They are widely used in modern industrial facilities and critical infrastructures, such as electric power generation and distribution systems, oil and gas refineries and transportation systems, allowing operators to ensure the proper functioning of these facilities and infrastructures.

Electric power utilities, for instance, were among the first to widely adopt remote monitoring and control systems. Their earliest SCADA systems provided simple monitoring through periodic sampling of analog data, but have evolved into more complex communication networks. In this paper, we focus on securing SCADA systems for electric power generation and distribution. However, our discussions and proposed solution are applicable to many other SCADA systems.

1.1.1 Devices and Communications

A SCADA system consists of physical devices and communication links that connect them together. Typical communications in a SCADA system include exchanging control and status information between master and slave devices. Master devices, most of which are PCs or *programmable logic controllers (PLCs)*, control the operation of slave devices. A slave device, such as a remote terminal unit (RTU), can be a sensor, an actuator, or both. Sensors read status or measurement data of field equipments such as voltage and current. Actuators send out commands or analog set-points such as opening or closing a switch or a valve.

Most SCADA systems have traditionally used low-bandwidth communication links, e.g., radio, direct serial and leased lines, with typical baud rates from 9600 to 115200. They are known as *serial-based SCADA systems*. Communication protocols used in these systems, or *serial-based SCADA protocols*, are very compact—messages are short, and slave devices send information only when polled. Among the many of these protocols, Modbus [12] and DNP3 [6] are the more popular ones.

1.1.2 Security Trouble

Many serial-based SCADA systems in operation today were implemented decades ago with availability and personnel safety as the primary concerns, rather than security. As with any complex protocols not designed to withstand adversarial action, these protocols are vulnerable to a variety of malicious attacks such as eavesdropping, tampering, injection and replying. The risks associated with such a lack of security in these systems are ever increasing, as an initial protection of "security through obscurity" breaks down. First, after initial dependence on proprietary elements, it is increasingly common to build SCADA systems using commercial off-the-shelf (COTS) hardware and software along with open communication protocols, the technical internals of which are often easily accessible. Second, many utilities have replaced, to various extent, their private networks by public ones such as the Internet. Their SCADA networks and corporate networks have also become highly inter-connected to achieve efficient information exchange—leading to increased risk of intentional or inadvertent exposure to the Internet. Finally, teams of sophisticated hackers are now employed by commercially motivated organizations or terrorists to attack and/or break into systems illegally.

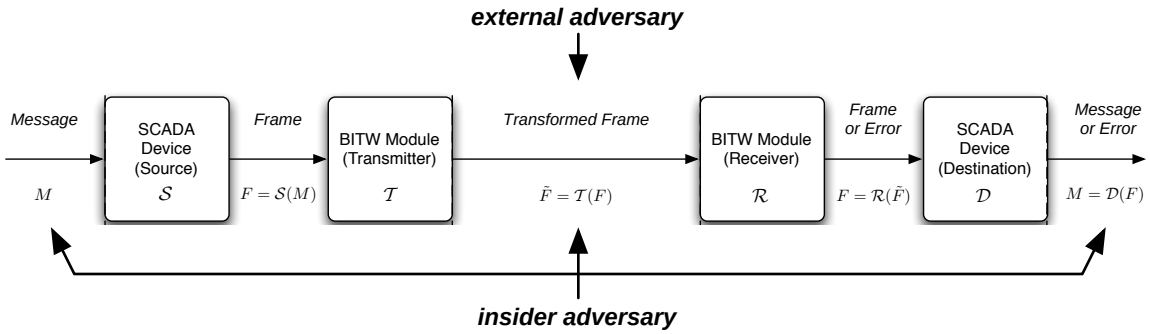

Figure 1: System and attack model for "bump-in-the-wire" approach.

In short, adversaries have increasing knowledge, increasing attack surface, and increased motivation.

1.1.3 Retrofitting Security

Considering the devastating consequences that failures of critical infrastructures could lead to,[1] it is paramount to secure SCADA systems against malicious attacks. In the long run, existing insecure SCADA systems may eventually be replaced by newer ones built with better technologies and with security as one of the primary goals—we will see devices that are computationally more powerful, communication links with higher bandwidths and protocols with built-in security. For example, new protocols such as DNPSec [11] and IEC 61850 [9] have been designed and employed in some new SCADA products.

However, for at least the next several decades (and possibly longer),[2] achieving security requires *non-intrusively retrofitting it* to existing insecure and legacy SCADA systems. It is economically infeasible, if not technically impossible, to simply throw away the existing infrastructures overnight. In such an effort, several *"bump-in-the-wire" (BITW)* solutions have been developed. In a BITW solution, two hardware modules are inserted into the communication link, one next to each of the communicating SCADA devices, as depicted in Figure 1. These BITW modules transparently augment the SCADA devices with the necessary security guarantees through mechanisms such as encryption and data authentication.

1.2 The Challenge

BITW solutions secure SCADA communications at the expense of incurring additional end-to-end communication latency, due to the need of processing and buffering in the BITW modules. Buffering can be prohibitively expensive in low-bandwidth communication links. As an example, a serial link with a baud-rate of 9600 has a byte-time (i.e. the time to send 8 bits of information) of roughly 1ms. If each of the two BITW modules buffers up a message of 20 bytes before processing

[1] As an example of cyber-attacks on critical infrastructures, in 2001, an Australian man hacked into a computerized sewage management system and dumped millions of liters of untreated waste into local parks and rivers [22]. For a fictional but not entirely unrealistic scenario, watch the movie *Live Free or Die Hard*.

[2] The expected lifetime of many SCADA equipments spans from 20 to 50 years.

it, then an overhead of 40ms is incurred, due to buffering alone. If the message has 250 bytes instead, the overhead becomes 0.5s.[3]

Such an overhead could be intolerable for serial-based SCADA systems that have timing constraints on communication latencies. For example, the exchange of event notification information for bus and transformer protection function between Intelligent Electronic Devices (IEDs) within a substation must be accomplished within 10ms, and the maximum delivery time for information such as response to data poll and phasor measurements is up to 0.2s [10].

As we will see when we review some of the existing solutions in Section 2.3, retrofitting data confidentiality to the communications in serial-based SCADA systems, even the time-critical ones, is a relatively trivial task; the real challenge lies in retrofitting *data integrity* (and thus data authenticity and freshness) to these systems in a timely manner, as the straightforward application of conventional data authentication techniques does not provide the required timing guarantee: the BITW module at the receiving end of the communication must *"hold back"* the message, i.e., it must wait until the receipt of the message and its authentication information in their entirety before relaying the message to the destination SCADA device, if the message is indeed authentic, intact and fresh. This incurs a latency dependent linearly on the length of the message being authenticated.

1.3 Our Contributions

In this paper, we propose *Yet Another SecurIty Retrofit*, or *YASIR*, which is a novel BITW solution for retrofitting security to time-critical communications in serial-based SCADA systems. To the authors' best knowledge, our solution is the first that achieves all of the following goals simultaneously:

- *High Security. YASIR* provides data integrity, and optionally data confidentiality, against not only eavesdroppers but stronger adversaries such as insiders, at a security level of almost 80 bits.[4]
- *Low Latency. YASIR* incurs an overhead of at most 18 byte-time, irrespective of the length of the message being authenticated, and can hence secure time-critical SCADA communications.
- *Comparable Cost. YASIR*'s BITW modules have hardware costs comparable to many existing solutions. Deploying *YASIR* is thus economically practical.
- *Standard and Patent-free Tools.* All cryptographic tools and techniques used in *YASIR*, such as AES and HMAC, are NIST-standardized [13] and patent-free.

Paper Organization In Section 2, we provide some background on SCADA protocols and cryptographic tools, and review several existing BITW solutions. Section 3 covers the preliminaries such as the threat model and security requirements of BITW solutions and the assumptions we have made about SCADA protocols. We give an overview to our solution in Section 4, highlighting the solution idea and some of the design choices. The actual construction of our solution is detailed in Section 5, and evaluated in Section 6. Section 7 concludes the paper.

[3] A typical SCADA message has a length of roughly 20 bytes. However, there are SCADA protocols that allow a maximum message length of 256 bytes or more.

[4] A security solution attains a security level of λ bits if brute-forcing a space of 2^{λ} possibilities is the most effective strategy for an adversary to break the solution's security.

2 Background

2.1 SCADA Protocols

The data link layer of a SCADA protocol specifies how control and data messages are encoded into bit-sequences known as *frames* for transmission over the communication link. Let $||$ denote the concatenation of (bit- or octet-) strings. A frame F has the form of $F = \mathtt{S}||H||P||\mathtt{E}$, where H is the header, P is the payload, and `S` and `E` are respectively the starting and ending symbol. Depending on the SCADA protocol, frames may or may not contain a header.

Headers, if present, may contain control information about the frame such as its type and length. Payloads contain messages in its encoded form and usually vary in length. `S` and `E` are bit-sequences distinct from any code symbols used in the rest of the frame so that a SCADA device can detect unambiguously the start and end of a frame. In many asynchronous protocols including Modbus/ASCII and DNP3-Serial, frame boundaries can be recognized within two byte-time. In Modbus/RTU, which is a synchronous protocol, a silence of 3.5 byte-time indicates the end of a frame.

Detecting Random Errors To detect the potential random errors introduced into the frames during transmission, error-detection codes such as Cyclic Redundancy Check (CRC) exist pervasively at the data link layer of SCADA protocols. Most commonly, the last few octets of the payload is a CRC on the rest of the frame (excluding `S` and `E`). This is the case for Modbus, for example. In some protocols such as DNP3, a CRC is appended to every chunk of certain length of the payload.

SCADA devices also utilize the length of a frame to detect random errors. If a frame has length longer than the specified maximum possible length, then an error must have occurred. In addition, for protocols in which the length of a frame can be inferred from its header, the SCADA devices know where to expect the ending symbol and can thus use this information to detect random errors.

The destination SCADA device drops the frame if it determines that the frame contains errors. For simplicity's sake, many serial-based SCADA protocols do not have a mechanism for retransmission at the data link layer. We stress that these techniques check for random errors only and provide no assurance on data integrity against malicious attacks.

2.2 Cryptographic Tools

We review some basic cryptographic tools that we use.

2.2.1 AES-CTR

Advanced Encryption Standard (AES) [15] is a block cipher with block size of 16 octets. A block cipher can operate in different *modes*, which specify ways of turning an operation on block-length values into an operation on longer-length messages [7]. In the *counter mode (CTR)*, a counter (incremented for each new block) is combined with a nonce and then encrypted under the block operation; the output is then XOR-d against the corresponding block of plaintext. The security of CTR mode is proven in [3].

Our solutions uses AES operating under the counter mode because, among its several other benefits, the "stream" nature of this mode means no buffering of plaintext is required. AES operating in CTR mode is NIST-standardized and patent-free. We denote AES operating in counter mode as AES-CTR, and denote AES-CTR with 128-bit keys as AES-CTR-128.

2.2.2 SHA and HMAC

SHA-1 and SHA-256 [14] are cryptographic hash functions that operate on messages of any length and produce 160-bit and 256-bit outputs respectively. A secure cryptographic hash function must possess properties such as preimage resistance and collision resistance. HMAC-SHA-1 and HMAC-SHA-256 [16] are keyed hash message authentication codes (HMACs) built from SHA-1 and SHA-256 respectively. HMAC-SHA-1-80 [16] is the same as HMAC-SHA-1, except that the 160-bit output is truncated to the first 80-bit. The above hash functions and HMACs have the same block size of 512-bit.

Our solution uses SHA-1 to compute the cryptographic digest of frame contents, and HMAC-SHA-1-80 for authenticating frames. They are all NIST-standardized and patent-free.

2.3 Existing Solutions

Encryption-only solutions are not under our consideration in this paper as retrofitting data confidentiality only does not provide sufficient security. Also, we do not consider a solution to be BITW if it requires replacing the communication link with one with a higher bandwidth, such as an Ethernet link, and/or upgrading the hardware and/or software of the SCADA devices. In the following, we review several BITW existing solutions, none of which is capable of securing time-critical communications in serial-based SCADA systems.

2.3.1 SEL's Serial Encrypting Transceiver

The SEL-3021 Serial Encrypting Transceiver [21] from Schweitzer Engineering Laboratories, Inc. (SEL) is a BITW module for securing EIA-232 serial links between SCADA devices against malicious attacks. Both available models, SEL-3021-1 and SEL-3032-2, support all standard SCADA protocols, including DNP3-Serial and Modbus/RTU, at data-rates up to 115200 bps.

The *SEL-3021-1* provides data confidentiality only, through encrypting SCADA messages using AES-CTR-128. It incurs a communication latency of only 5 byte-times [19], and hence can be used in systems that require time-critical communications. Unfortunately, SEL-3021-1 does not provide data integrity.

The *SEL-3021-2* provides data integrity with optional data confidentiality. It uses either HMAC-SHA-1 or HMAC-SHA-256. Unfortunately, SEL-3021-2 does not provide an upper-bound on the communication latency it may incur [20]. In fact, SEL suggests that SEL-3021-2 may not be suitable to secure links that require time-critical communications with low latency such as links for electrical tele-protection [20].

2.3.2 AGA's SCADA Cryptographic Module

The American Gas Association (AGA) [1] has been developing a set of standards for protecting SCADA communications. In particular, the AGA 12 Task Group designed the SCADA Cryptographic Module (SCM) [2] as a BITW solution that retrofits data integrity to SCADA communications while maintaining the performance requirements. The solution has an implementation [18] due to Andrew Wright at Critical Infrastructure Assurance Group (CIAG), Cisco Systems [5].

AGA's SCM provides several cipher-suites to choose from. The most secure ones use AES-CTR for data confidentiality and either HMAC-SHA-1 or HMAC-SHA-256 for data integrity. Messages must be held back by the receiving SCM. At an abstract level, AGA's SCMs, when employing one

of these cipher-suites, are no different from SEL-3021-2 in terms of cryptographic techniques used and performance delivered.

PE Mode of Operation In one of the cipher-suites provided by AGA's SCM, data authentication is achieved by operating AES in the Position-Embedded (PE) mode, designed by Wright et al. [23]. Using this cipher-suite, SCMs can provide data integrity with an overhead of only 32 byte-times, regardless of the message length. To the authors' best knowledge, it is the first and so far the only solution that provides data integrity without the need of holding back messages.

Unfortunately, SCMs employing this cipher-suite provide only λ-bit level of security, where λ is typically 8 or 16: with a probability of $2^{-\lambda}$, SCMs will accept arbitrarily crafted messages as authentic.[5] As a remedy, SCMs rely in addition on traditional HMAC for more secure data authentication. However, as pointed out in [11], even though unauthentic messages can eventually be detected, it is too late to prevent forwarding them to the SCADA devices. Moreover, AES in PE mode is proven secure only under known-plaintext attacks [23]. Consequently, this solution is not guaranteed to be secure against, for instance, attacks from an insider such as a compromised employee working in the control center.

2.3.3 PNNL's Secure SCADA Communications Protocol

A SCADA communications authenticator technology is under development by a group led by Mark Hadley at the Pacific Northwest National Laboratory (PNNL) [17]. In PNNL's solution, each SCADA message being protected is "wrapped" by an authenticator and potentially some other information such as a unique identifier. Their solution is effectively a protocol wrapper that converts an insecure SCADA protocol into their Secure SCADA Communications Protocol (SSCP).

PNNL's technology is being implemented both as a BITW solution and an embedded solution [8]. The BITW solution requires message hold-back. The embedded solution is fast but is not a BITW solution: it requires upgrading the hardware and/or software of the SCADA devices.

2.3.4 Summary

Thus, we see that previous BITW approaches—that can secure deployed legacy SCACA—all fall short in some critical property: they don't provide integrity against malicious attacks, or they don't provide security against a dedicated adversary, or they delay messages too long. The one approach that provides these properties is not bump-in-the-wire.

Table 1 summarizes this picture.

3 Preliminaries

3.1 Formalizing BITW Solutions

As Figure 1 showed, a *source* SCADA device $\mathcal{S}$ converts messages into frames. We overload $\mathcal{S}$ to denote the function that models the device, which takes a message M as input and outputs the corresponding frame F. Similarly, the *destination* SCADA device $\mathcal{D}$ is modeled as a function $\mathcal{D}$, which takes a frame F' as input and outputs one of two things:

[5] This is far below the generally accepted minimum of 80-bit level of security. Even in a few scenarios where 32-bit level of security is justifiable [16], $\lambda = 8$ or 16 is still too low.

Approach	*Bump-in-the-wire?*	*Confidentiality?*	*Integrity?*	*Security Level*	*Latency (byte-times)*
SEL 3021-1	Yes	Yes	***No*** ☹	High	Low (5)
SEL 3021-2	Yes	Yes (option)	Yes	High	***High*** ☹
AGA12/Cisco, PE-mode	Yes	Yes (option)	Yes	***Low*** ☹	Low (~32)
AGA12/Cisco, other modes	Yes	Yes (option)	Yes	High	***High*** ☹
PNNL SSCP BITW	Yes	Yes (option)	Yes	High	***High*** ☹
PNNL SSCP embedded	***No*** ☹	Yes (option)	Yes	High	Low (<10)
YASIR (our approach)	Yes ☺	Yes (option) ☺	Yes ☺	High ☺	Low (≤18) ☺

Table 1: Previous BITW solutions for securing communications in legacy SCADA systems all fall short in some critical property; the one previous approach that provides the critical property is not bump-in-the-wire. Our approach meets all the criteria.

- either an *error* if F' fails to pass certain conformance checks such as the random-error detection described in Section 2.1,
- or the corresponding original message M'.

If no error was introduced (randomly or maliciously) into F during its transmission (that is, if $F' = F$), then a correct pair of $\mathcal{S}$ and $\mathcal{D}$ always give $\mathcal{D}(F') = \mathcal{D}(F) = \mathcal{D}(\mathcal{S}(M)) = M$. If $F' \neq F$, then $\mathcal{D}$ may or may not return an error, depending on whether F' passes the conformance checks in $\mathcal{D}$. As discussed, most of the random errors can be caught by $\mathcal{D}$.

Now, any BITW solution injects two hardware modules into the link in the model, one next to $\mathcal{S}$ and the other next to $\mathcal{D}$, which we call the *Transmitter* $\mathcal{T}$ and the *Receiver* $\mathcal{R}$ respectively. Refer to Figure 1 again for a diagrammatic illustration. Again $\mathcal{T}$ is overloaded to denote the function that models the Transmitter, which takes each frame F output by $\mathcal{S}$ and outputs the corresponding transformed frame $\tilde{F}$ to be transmitted over the insecure channel. Similarly, the Receiver $\mathcal{R}$ is modeled as a function $\mathcal{R}$ that takes in a transformed frame $\tilde{F}'$ and outputs either an *error*, or the corresponding original frame F' to be given to $\mathcal{D}$. If no error was introduced (randomly or maliciously) into $\tilde{F}$, i.e., $\tilde{F}' = \tilde{F}$, a correct pair of $\mathcal{T}$ and $\mathcal{R}$ always give $\mathcal{R}(\tilde{F}') = \mathcal{R}(\tilde{F}) = \mathcal{R}(\mathcal{T}(F)) = F$. In most existing BITW solutions that provides data integrity, if for whatever reason $\tilde{F}' \neq \tilde{F}$, $\mathcal{R}$ outputs an error. Effectively, $\mathcal{R}$ acts as a guard and discards all malformed frames so that $\mathcal{D}$ won't even see them.

Finally, a SCADA device can be the source at one time and the destination at another. The BITW module attached to a SCADA device will thus play the role of a Transmitter and that of a Receiver accordingly.

3.2 Assumptions

There are more than a hundred SCADA protocols in use today, many of which are legacy and proprietary. An ideal BITW solution would make no assumption about the underlying SCADA protocol it is protecting, except perhaps how frame boundaries are indicated, and could thus be applied to, upon simple configuration, protect any protocol. In practice, a BITW solution should

make minimal assumptions about the protocols and be applicable to the majority of the more popular SCADA protocols. Our proposed solution to be presented does require certain assumptions to be made about the underlying protocols, but is otherwise designed so that those assumptions hold for the majority of SCADA protocols. Specifically, our solution assumes that a SCADA protocol belongs to Type-I or Type-II (or both), where:

- *A protocol is of Type-I if:* The end of a frame is indicated by the ending symbol, and, excluding `S` and `E`, the last $n \geq 1$ octets in the frame is a CRC of (a part of) the rest of the frame according to certain known CRC algorithm. A receiving SCADA devices drops the frame if the CRC is wrong. For example, in Modbus/ASCII, the last two octets is a CRC-16 on the rest of the payload; in DNP3, the last two octets is a CRC-16 on the previous 16 bytes.

- *A protocol is of Type-II if:* Frames have a header that contain information from which the length of the frame can be inferred. The receiving SCADA device drops the frame if the actual length of the frame does not match what is indicated in the header. For example, DNP3 frames contain in the header the size (in terms of the number of 16 octets) of the payload excluding the CRCs.

As one can see, the requirements for a protocol to be of one of the above two types are generic enough that it is highly likely that any legacy SCADA protocol belongs to at least one of them. Moreover, even for a closed and proprietary protocol, it is relatively easy to figure out if it belongs to one of the types, as well as the CRC algorithm used, by, e.g., examining several actual frames coming out of a real SCADA device.

3.3 Security Model

3.3.1 Security Goals

We would like to guarantee that both of the following security requirements are satisfied:

- *Data Integrity (and thus Data Authenticity and Freshness).* A destination SCADA device only accepts a frame if it was originated from a predetermined source SCADA device, hasn't been modified along its way, and is not a replayed frame.

- *Data Confidentiality.* No information about the corresponding original message can be inferred from a frame in transit, except perhaps its size and some insensitive control information.

However, there are scenarios when data confidentiality is not a concern,[6]. There are even scenarios when data confidentiality is undesirable, such as when a message has multiple recipients. One example is when the control center wants to broadcast the same control message to all RTUs. Also, one might want to install a logging device that audits all the messages leaving or entering a SCADA device. As will become clear, our solution provides both data confidentiality and data integrity by default, and yet can easily be modified to provide data integrity only and send frames in cleartext by skipping the encryption step.

[6]For example, it is fine for an IED to report the current temperature reading to another IED within the same substation over an unencrypted channel because an adversary who has broken into the substation might as well go to read off the temperature directly from the sensing IED instead of tapping into the serial link.

3.3.2 Attack Surface and Adversaries

In most SCADA systems, communication links travel through a long distance to connect the SCADA devices together. It is impossible to keep the adversaries away from the entirety of link. This is the case for private leased lines, and even more so for public networks such as the Internet. As Figure 1 showed, in our threat model, *communication links are insecure:* when trying to attack a SCADA system, an adversary may arbitrarily eavesdrop, inject, modify and replay communications in these links.

If there existed security boundaries around the substations and the control centers, then attacking the communication link would be all the adversaries could possibly do. Unfortunately, such security boundaries do not exist. For example, an adversary may physically break into an underguarded substation, compromise an employee working in the control center, or remotely hack into the computers auditing the SCADA devices. In our threat model, *SCADA devices are insecure:* when trying to attack a SCADA system, an adversary may feed source and destination SCADA devices with maliciously crafted inputs and learn the corresponding outputs. In other words, we allow adaptive chosen-plaintext and chosen-ciphertext attacks.

4 Solution Overview

4.1 The Idea

Recall that the BITW module at the receiving end $\mathcal{R}$ acts as a guard for the destination SCADA device $\mathcal{D}$ in most existing solutions. Since $\mathcal{R}$ can't decide if a frame is authentic and hence start relaying until the receipt of the whole frame and its authentication information, the latency grows linearly with the frame length. AES in PE mode used in AGA's SCM for authentication is a novel exception. $\mathcal{R}$ starts relaying the frame to $\mathcal{D}$ before the authenticity of the frame is known. However, $\mathcal{R}$ operates on the frame in such a way that, with probability close to 1, $\mathcal{D}$ will flag a CRC error and drop the frame if it is indeed maliciously tampered.[7]

In a sense, AGA's solution converts random-error detection, already built in to the legacy SCADA devices, into a mechanism for verifying data integrity against malicious attacks. In their solution, the conversion relies on the "real-or-random indistinguishability" property [4] of AES when used as a block cipher. However, this has three drawbacks: first, one 16-byte block of data must be buffered at each BITW module. Second, the receiving BITW module doesn't really know if the frame is authentic or not and can only hope that a maliciously tampered frame will result in a random block that leads to a wrong CRC. Third, this approach is proven secure only against known-plaintext attacks, but not stronger and yet still very realistic adversaries such as insiders.

Our solution shares the same idea of converting random-error detection to data integrity checking, but is different in how that conversion is done, which makes our solution better in a number of ways including lower timing overhead, higher level of security, and security guarantee against stronger adversaries. In particular, our solution uses the conventional mechanism of HMAC to provide data integrity to frames at a high security level, but in a novel way so that no message holdback is required: first, our BITW modules operate on a frame as a stream of bytes instead of waiting until the receipt of the whole frame before operating on it. Second, $\mathcal{R}$ in our solution knows for sure the integrity of a frame by verifying the HMAC and is hence capable of forcing $\mathcal{D}$ to drop a frame if it is unauthentic, at a security level of 80-bit. Finally, the use of HMAC for data

[7] However, as discussed, the probability is not close enough to 1 in AGA's solution.

integrity allows our solution to be secure even under a stronger attacking model such as insider attacks.

The Design At a high level, this is how our solution provides data integrity: for each frame F the BITW Transmitter $\mathcal{T}$ receives from the source SCADA device $\mathcal{S}$, $\mathcal{T}$ appends an HMAC-SHA-1-80 on F to F and sends it off to the insecure channel. This can done without the need of holding back the frame. At the other end, the BITW Receiver $\mathcal{R}$ relays every byte it gets from the insecure channel to the destination SCADA device $\mathcal{D}$, but with a delay of 10 byte-time. Since a HMAC-SHA-1-80 MAC is 80-bit in length, by the time $\mathcal{R}$ relayed the last byte in the payload, it has just received the whole HMAC and is able to verify the integrity of the received frame.

If the HMAC verifies, all $\mathcal{R}$ has to do is to finish up relaying the frame by sending the ending symbol. However, if the HMAC does not verify, $\mathcal{R}$ additionally sends several bytes to $\mathcal{D}$, in order to cause the conformance checks at $\mathcal{D}$ to fail.

4.2 Other Design Choices

4.2.1 Encryption

YASIR uses AES in counter mode for encrypting the content of a frame because of its low latency. First, it is effectively a stream cipher that can encrypt and decrypt on a per-byte basis. No buffering is thus necessary for both encryption and decryption. Second, the block cipher operation during both encryption and decryption can be easily taken away from the critical path; when the operation is pre-computed in the background, both encryption and decryption of a byte are just a byte-wise XOR, which takes virtually no time. Third, AES in counter mode requires no message padding and hence causes no message expansion.

4.2.2 Authentication

Instead of calculating a HMAC directly on the encrypted content of a frame, *YASIR* calculates a HMAC on the SHA-1 hash on the encrypted content of a frame. While this extra SHA-1 computation has no effect (positively or negatively) on *YASIR*'s security, it has efficiency benefits. First, the *YASIR* BITW module only needs to remember the last 64-octet of the encrypted frame content and the intermediate 20-octet hash value, rather than the whole message, as SHA-1 hashes a message by iteratively operating the message in 64-octet blocks. Second, the HMAC operation now takes constant time, since the input to HMAC is always less than one block, again regardless of the frame length.

4.2.3 Sequence Numbers

While HMAC provides data authenticity, it does not defend against replay attacks. A standard solution is to assign a unique sequence number to each piece of data being authenticated and then calculate the HMAC on the data along with the sequence number. Contrary to most protocols in which sequence numbers are contained in frame headers, *YASIR* puts the sequence number towards the end of a frame in order to reduce the amount data $\mathcal{R}$ has to receive before it has gained enough information to reconstruct a frame and decide on the integrity of the frame. Since *YASIR* uses a 4-octet sequence number, 4 byte-times of the end-to-end latency is saved.

Note that $\mathcal{R}$ does not know the actual sequence number of a frame by the time it has finished relayed the frame to $\mathcal{D}$. To properly decrypt and authenticate the incoming transformed frame, $\mathcal{R}$ predicts the sequence number of the frame. The sequence number in the frame is used for synchronizing the sequence number between $\mathcal{T}$ and $\mathcal{R}$ in case the prediction was wrong.

5 Our Proposed Solution

In this section, we present our *YASIR* construction.

5.1 Parameters

The BITW Transmitter $\mathcal{T}$ and Receiver $\mathcal{R}$ share a 128-bit AES key SK and a 160-bit HMAC-SHA-1-80 key HK. These keys are re-negotiated on a regular basis,[8] such as once every day. $\mathcal{T}$ and $\mathcal{R}$ keep sequence numbers SEQ_T and SEQ_R of octet-length $\ell_S = 4$ respectively, both of which are reset to zero every time keys are re-negotiated. [9] These sequence numbers are used as both a nonce for AES-CTR-128 and a sequence number for HMAC. SEQ_T is incremented every time $\mathcal{T}$ gets a frame from $\mathcal{S}$ so that it is never reused under the same keys.

In the following, we let Hash denote the cryptographic hash function SHA-1, which takes an octet-string of any length as input and outputs the corresponding 20-octet digest, and let HMAC denote the HMAC function HMAC-SHA-1-80, which takes a 160-bit key and a octet-string of any length as inputs and outputs the corresponding 10-octet HMAC. We also let $\mathsf{Encrypt}$ denote the encrypting (resp. decrypting)[10] function AES-CTR-128, which takes as inputs a 128-bit key, a 4-octet sequence number and a plaintext (resp. ciphertext) of any length, and outputs the corresponding ciphertext (resp. plaintext). Finally, in case of Type-I protocols, we let CRC denote the CRC algorithm that takes a frame and outputs a boolean answer of the validity of a frame, as described in Section 3.2.

5.2 Algorithms

We now present our *YASIR* construction in the form of algorithmic pseudo-code. Figure 2 shows the format of the inputs and outputs of $\mathcal{T}$ and $\mathcal{R}$.

5.2.1 *YASIR* Transmitter

On input an incoming frame $F = \mathtt{S}||H||P||\mathtt{E}$, the YASIR Transmitter $\mathcal{T}$ does the following:

1. Output the corresponding transformed frame $\tilde{F} = \mathtt{S}||CTXT||\mathtt{E}||\mathtt{mac}||\mathtt{seq}||\mathtt{E}$, where

$$\begin{aligned} CTXT &= \mathsf{Encrypt}_{SK}(SEQ_T, H||P), \\ \mathtt{mac} &= \mathsf{HMAC}_{HK}(\mathsf{Hash}(CTXT||SEQ_T), \text{ and} \\ \mathtt{seq} &= SEQ_T. \end{aligned}$$

[8] Key management is outside the scope of this paper. One can use the same techniques for key (re-)negotiation as some existing BITW solutions.

[9] We chose $\ell_S = 4$ so that there is no practical chance of exhausting a sequence number in any SCADA deployment.

[10] When AES is operating in counter mode, the encrypting and decrypting functions are equivalent.

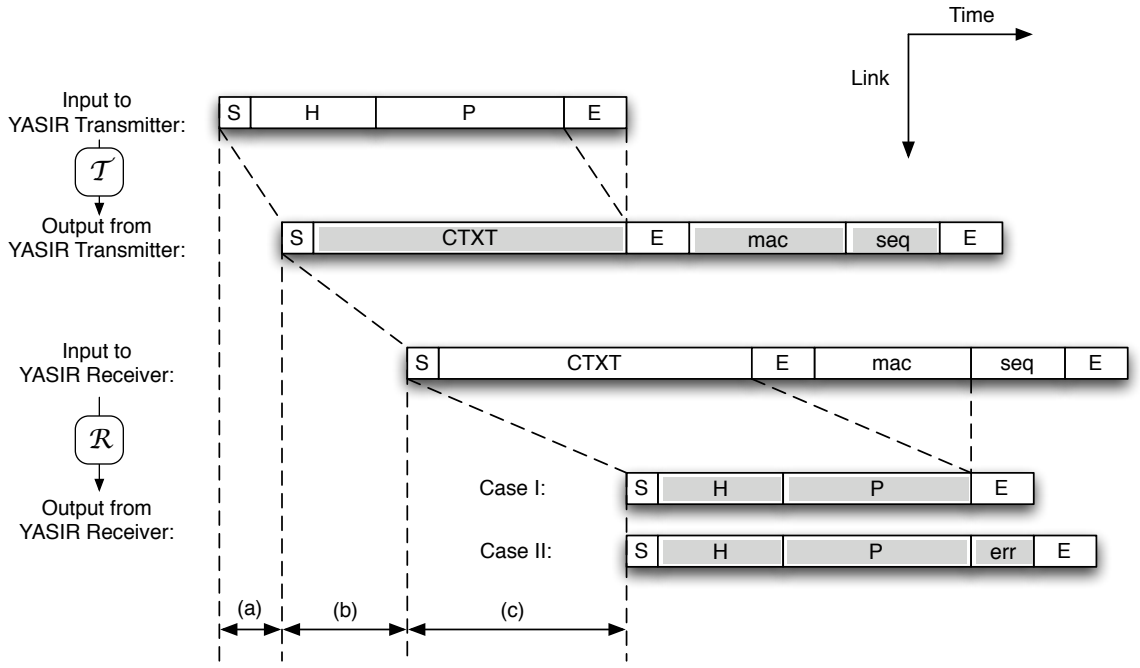

Figure 2: Latency incurred by (a) *YASIR* Transmitter, (b) the communication link itself, and (c) *YASIR* Receiver. Shaded boxes indicates values computed by the *YASIR* components.

2. Increment SEQ_T by 1.

In a nutshell, $\mathcal{T}$ transforms F by first encrypting its content, appending a HMAC on the ciphertext and the sequence number, and finally appending the sequence number itself. The additional ending symbol in the middle of $\tilde{F}$ provides an indication of the end of $CTXT$ and hence the start of `mac`.

5.2.2 *YASIR* Receiver

On input a transformed frame $\tilde{F}' = \texttt{S}||CTXT'||\texttt{E}||\texttt{mac}'||\texttt{seq}'||\texttt{E}$, the *YASIR* Receiver $\mathcal{R}$ does the following:

1. Compute

$$\begin{aligned} H'||P' &= \mathsf{Encrypt}_{SK}(SEQ_R, CTXT'), \text{ and} \\ \texttt{mac}'' &= \mathsf{HMAC}_{HK}(\mathsf{Hash}(CTXT')||SEQ_R). \end{aligned}$$

2. (Case I.) If $\texttt{mac}' = \texttt{mac}''$, output the corresponding original frame $F' = \texttt{S}||H'||P'||\texttt{E}$, and increment SEQ_R by 1.

3. (Case II.) Otherwise,

(a) Output a malformed frame $F'' = \mathtt{S}||H'||P'||\mathtt{err}||\mathtt{E}$, where `err` is any octet-string such that, in the case of Type-I protocols, $\mathsf{CRC}(F'')$ is invalid, or, in the case of Type-II protocols, the length of F'' is different from what is indicated in H'.

(b) If $\mathtt{seq}' > SEQ_R$ and $\mathtt{mac}' = \mathsf{HMAC}_{HK}(\mathsf{Hash}(CTXT')||\mathtt{seq}')$, assign $\mathtt{seq}' + 1$ to SEQ_R.

In other words, $\mathcal{R}$ reconstructs F' from $\tilde{F}'$ simply by decrypting $CTXT'$ if F' contains a valid HMAC. Otherwise, $\mathcal{R}$ injects `err` into F' during its reconstruction in such as way that F' will fail the conformance check in $\mathcal{D}$. For Type-I protocols, `err` is chosen to be any incorrect CRC (by first calculating the correct one). For Type-II protocols, the length of F' is inferred from H' and `err` is chosen to be any octet-string such that the length of F' differs from that indicated in H'. In either protocol-type, a desired `err` of almost 2 octets in length can be found efficiently.

Finally, in case when the HMAC is invalid, step (3b) provides a mechanism for synchronizing the sequence numbers in $\mathcal{T}$ and $\mathcal{R}$, which have potentially gone out of synchronization due to the random or malicious corruption of one or more frames recently sent. $\mathcal{R}$ does so by updating its own sequence number with the one contained in the frame, but only when the integrity of the frame can be verified using that sequence number.

5.2.3 Finite State Machines

We have included in the Appendix a presentation of our *YASIR* construction in the form of finite state machines (FSMs). They closer reflect how the algorithms above can be implemented in as a hardware module.

6 Evaluation

6.1 Efficiency Evaluation

6.1.1 End-to-end Latency

As discussed, all the *YASIR* Transmitter needs to do upon receiving an incoming byte is an byte-wise XOR. Other operations such as AES operation and computing the HMAC of the input are not on the critical path. As illustrated by Figure 2, *YASIR* Transmitter incurs no latency, except the time needed to recognize if the incoming code symbol is the ending symbol or not. There is at least 1 byte-time for the Transmitter to calculate the HMAC on the hash on the encrypted content, which involves only roughly two internal rounds of SHA-1, since the hash is being computed iteratively on the fly as discussed. Even for links with baud-rate as high as 115200, such computation can be easily done in 1 byte-time.

Regarding the latency incurred by the *YASIR* Receiver, the above arguments remain valid, as decryption goes through exactly the same procedure as encryption, and verifying a HMAC is done by computing a HMAC and comparing the results. Now, the overhead unique to the Receiver is a delay of 10 byte-time before relaying the decrypted frame content because the Receiver has 10 octets of HMAC to receive after the ciphertext. Now once the HMAC is received and verified, coming up with an `err` is easy in case it is needed. For Type-I protocols, there is a period of at least 10 byte-times to generate an incorrect CRC by calculating the correct CRC after the encrypted frame content is decrypted. CRCs are calculated iteratively as bytes come in and can be done in

virtually no time. For Type-II protocols, inferring the length of a frame from the header is just simple arithmetic.

In sum, if the time required to recognized the ending symbol is t_E, then the total end-to-end-latency incurred by *YASIR* is $10 + 2t_E$ byte-times. Since $t_E < 4$ in most protocols, our solution is at least twice as fast as AGA's SCM employing AES in PE mode, which has a latency of $32 + 2t_E$.

6.1.2 Hardware Costs

A *YASIR* Receiver requires an AES core and a SHA-1 core. HMAC-SHA-1-80 can be implemented by reusing the SHA-1 core with minimal additional circuitry. Some memory registers are needed to store the states, such as the keys, the sequence number, the intermediate value of the hash output, and a few counters.

A *YASIR* Transmitter requires virtually the same hardware as a Transmitter, except an additional buffer of 10-octets for delaying the relaying of frames by 10 byte-times.

Although a *YASIR* BITW module functions both as a Transmitter and a Receiver, it never needs to play both role at the same time and can thus reuse the hardware for both functionalities. Therefore, the hardware required to build a *YASIR* BITW module is the union of that required to build a Transmitter and that required to build a Receiver.

6.2 Security Evaluation

6.2.1 Data Confidentiality

Data confidentiality of *YASIR* is implied by the security of AES operating in counter mode and the security of HMAC-SHA-1-80. A transformed frame contains a ciphertext, an HMAC and a sequence number. Without the knowledge of the AES key, the security of AES in counter mode guarantees that no computationally bounded adversary can learn from any ciphertext any information about its corresponding plaintext except its size, even given the ability to launch chosen plaintext attacks, as along as a nonce is not reused, which is ensured in *YASIR* as the sequence number never overflows. *YASIR* allows chosen ciphertext attacks because the data integrity property possessed by *YASIR* (to be argued next) means $\mathcal{R}$ will act as a decryption oracle for an adversary only when the input is produced by $\mathcal{T}$. Also, the security of HMAC-SHA-1-80 guarantees that without the knowledge of the HMAC key, given an output of HMAC-SHA-1-80, any message is equally likely to be the corresponding input to a computationally bounded adversary, even if the adversary is given oracle access to the HMAC function. Finally, the sequence number provides the adversary with no additional information statistically correlated to the SCADA message.

6.2.2 Data Integrity

As long as both AES operating in counter mode and HMAC-SHA-1-80 are secure, *YASIR* has data integrity, and thus data authenticity and freshness. To see why, we first consider the case when *YASIR* is used to secure a Type-I protocol. Now, let us assume the contrary that there exists a computationally bounded adversary who is capable breaking *YASIR* data integrity. Then $\mathcal{D}$ must have accepted at least one frame that is unauthentic or replayed, or both. Since $\mathcal{D}$ thinks the CRC of that frame is correct, $\mathcal{R}$ must have determined that the contained HMAC is valid, which means the frame can't be unauthentic, given the security of HMAC-SHA-1-80 against any computationally bounded adversary with oracle access and the fact. Consequently, the frame must be a replayed

frame, which means it contains a sequence number smaller than the sequence number of some frame accepted by $\mathcal{R}$ as authentic. However, this is a contradiction because the sequence numbers contained in the frames accepted by $\mathcal{R}$ as authentic must be strictly increasing. The arugments for Type-II protocols are similar.

7 Conclusion

In this paper, we have proposed a BITW solution for retrofitting security to serial-based SCADA systems in which communications are time-critical, such as those for electric power generation and distribution. As Table 1 shows, our solution is the first to provide data integrity in a timely manner, at a high security level even against strong adversaries such as insiders.

We are actively developing a microcontroller prototype for *YASIR*. It is currently capble of speaking Modbus/ASCII and DNP3. In the final version of this paper, we plan to report empirical performance figures when *YASIR* is used to secure real SCADA devices.

References

[1] The American Gas Association. http://www.aga.org/.

[2] American Gas Association. Cryptographic Protection of SCADA Communications Part 1: Background, Policies and Test Plan. Technical Report AGA Report No. 12, American Gas Association, March 2006. http://www.gtiservices.org/security/AGA%2012%20Part%201%20Final%20Version.pdf.

[3] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *FOCS*, pages 394–403, 1997.

[4] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *FOCS*, pages 394–403, 1997.

[5] Critical Infrastructure Assurance Group (CIAG), Cisco Systems. http://www.cisco.com/web/about/security/security_services/ciag/.

[6] DNP Users Group. http://www.dnp.org.

[7] Morris Dworkin. Recommendation for block cipher modes of operations–methods and techniques. Technical report, National Institute of Standards and Technology (NIST), Dec 2001. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[8] Mark Hadley. Personal communication, Aug 2007.

[9] IEC 61850 Communication Networks and Systems in Substations. http://www.61850.com/.

[10] IEEE standard communication delivery time performance requirements for electric power substation automation. IEEE Std 1646-2004, 2005.

[11] Munir Majdalawieh, Francesco Parisi-Presicce, and Duminda Wijesekera. DNPSec: Distributed Network Protocol Version 3 (DNP3) Security Framework. In *Advances in Computer, Information, and Systems Sciences, and Engineering: Proceedings of IETA 2005, TeNe 2005, EIAE 2005*, pages 227–234. Springer, 2006.

[12] Modbus-IDA. http://www.modbus.org/.

[13] National institute of standards and technology. http://www.nist.gov/.

[14] NIST. FIPS 180-2: Secure hash standard (SHS). Technical report, National Institute of Standards and Technology (NIST), 2001. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

[15] 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.

[16] 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.

[17] Pacific Northwest National Laboratory. http://www.pnl.gov/.

[18] ScadaSafe. http://scadasafe.sourceforge.net/.

[19] Schweitzer Engineering Laboratories, Inc. SEL-3021-1 datasheet. http://www.selinc.com/datasheets/3021-1_DS_20060929.pdf.

[20] Schweitzer Engineering Laboratories, Inc. SEL-3021-2 datasheet. http://www.selinc.com/datasheets/3021-2_DS_20070109.pdf.

[21] Schweitzer Engineering Laboratories, Inc. Serial encrypting transceiver. http://www.selinc.com/sel-3021.htm.

[22] Tony Smith. Hacker jailed for revenge sewage attacks. *The Register*, Oct 31 2001. http://www.theregister.co.uk/2001/10/31/hacker_jailed_for_revenge_sewage/.

[23] Andrew K. Wright, John A. Kinast, and Joe McCarty. Low-latency cryptographic protection for scada communications. In *ACNS*, volume 3089 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2004.

A Finite State Machines

Finite State Machine 1 represents a hardware implemenation of *YASIR* Transmitter. Finite State Machine 2 represents a hardware implemenation of *YASIR* Receiver.

Notice that hashing a string $S = S_1||\ldots||S_n$ involves initializing the hash function through $\mathsf{Hash}.init()$, n invocation of $\mathsf{Hash}.update()$, each on S_i for $i = 1$ to n in order, and finally $\mathsf{Hash}.final()$.

Finate State Machine 1 YASIR Transmitter

Persistant States: Octet-strings MK, EK and of length $\ell_{MK} = 20$ and $\ell_{EK} = 16$ resp., and an integer $\mathtt{seq}_T$ of octet-length $\ell_S = 4$

Variables: Octet-strings `digest`, `mac`, `otp` and `msg` of length $\ell_H = 20$, $\ell_M = 10$, $\ell_E = 16$, $\ell_B = 64$ resp., and an integer `ctr` of octet-size $\ell_C = 4$.

INIT :
  On $\langle\mathtt{input}\rangle = \mathtt{S}$: goto START;
START :
  $\mathtt{seq}_T \leftarrow \mathtt{seq}_T + 1$; $\mathtt{ctr} \leftarrow 0$; $\mathtt{otp} \leftarrow \mathsf{AES}_{EK}(\mathtt{seq}_T||00\ldots0)$; $\mathsf{Hash}.init()$;
  $\langle\mathtt{output}\rangle \leftarrow \mathtt{S}$;
  goto MAIN;
MAIN :
  On $\langle\mathtt{input}\rangle = \mathtt{S}$: goto START;
  On $\langle\mathtt{input}\rangle = \mathtt{E}$:
    $\mathsf{Hash}.update(\mathtt{msg}[0\ldots(\mathtt{ctr} \bmod \ell_B)])$; $\mathtt{digest} \leftarrow \mathsf{Hash}.final()$;
    $\mathtt{mac} \leftarrow \mathsf{HMAC}_{MK}(\mathtt{seq}_T||\mathtt{digest})$;
    goto END;
  On $\langle\mathtt{input}\rangle \in \mathcal{D}$:
    $\langle\mathtt{output}\rangle \leftarrow \mathtt{otp}[\mathtt{ctr} \bmod \ell_E] \oplus \langle\mathtt{input}\rangle$;
    $\mathtt{msg}[\mathtt{ctr} \bmod \ell_B] \leftarrow \langle\mathtt{input}\rangle$;
    $\mathtt{ctr} \leftarrow \mathtt{ctr} + 1$;
    if $\mathtt{ctr} \bmod \ell_B = 0$ **then**
      $\mathsf{H}.update(\mathtt{msg})$;
    if $\mathtt{ctr} \bmod \ell_E = 0$ **then**
      $\mathtt{otp} \leftarrow \mathsf{AES}_{EK}(\mathtt{seq}_T||\mathtt{ctr}/\ell_E||00\ldots0)$;
    goto MAIN;
END :
  $\langle\mathtt{output}\rangle \leftarrow \mathtt{E}$;
  for $i = 0$ to $\ell_M - 1$:
    $\langle\mathtt{output}\rangle \leftarrow \mathtt{mac}[i]$;
  for $i = 0$ to $\ell_S - 1$:
    $\langle\mathtt{output}\rangle \leftarrow \mathtt{seq}_T[i]$;
  $\langle\mathtt{output}\rangle \leftarrow \mathtt{E}$;
  goto INIT;

Finate State Machine 2 YASIR Receiver

Persistant States: Octet-strings MK and EK of length $\ell_{MK} = 20$ and $\ell_{EK} = 16$ resp., and an integer $\texttt{seq}_R$ of octet-length $\ell_S = 4$.

Variables: Octet-strings `digest`, `mac`, `buf`, `otp`, `msg` and `err` of length $\ell_H = 20$, $\ell_M = 10$, ℓ_M, $\ell_E = 16$, $\ell_B = 64$ and $\ell_{err} = 2$ resp., and integers `ctr`, $\texttt{ctr}_M$ and $\texttt{ctr}_S$ of octet-length $\ell_C = 4$.

1: INIT :
2: On $\langle\texttt{input}\rangle = \texttt{S}$: goto START;
3: START :
4: $\texttt{ctr} \leftarrow 0$; $\texttt{ctr}_M \leftarrow 0$; $\texttt{ctr}_S \leftarrow 0$; $\texttt{otp} \leftarrow \mathsf{AES}_{SK}(\texttt{seq}_R||00\ldots0)$; $\mathsf{Hash}.init()$;
5: $\langle\texttt{output}\rangle \leftarrow \texttt{S}$;
6: goto MAIN;
7: MAIN :
8: On $\langle\texttt{input}\rangle = \texttt{S}$: goto START;
9: On $\langle\texttt{input}\rangle = \texttt{E}$:
10: $\mathsf{Hash}.update(\texttt{msg}[0\ldots(\texttt{ctr} \bmod \ell_B)])$; $\texttt{digest} \leftarrow \mathsf{Hash}.final()$;
11: $\texttt{mac} \leftarrow \mathsf{HMAC}_{MK}(\texttt{seq}_R||\texttt{digest})$;
12: goto END;
13: On $\langle\texttt{input}\rangle \in \mathcal{D}$:
14: **if** $\texttt{ctr} \geq \ell_M$ **then**
15: $\langle\texttt{output}\rangle \leftarrow \texttt{buf}[\texttt{ctr} \bmod \ell_M]$;
16: $\texttt{buf}[\texttt{ctr} \bmod \ell_M] \leftarrow \texttt{otp}[\texttt{ctr} \bmod \ell_E] \oplus \langle\texttt{input}\rangle$;
17: $\texttt{msg}[\texttt{ctr} \bmod \ell_B] \leftarrow \langle\texttt{input}\rangle$;
18: $\texttt{ctr} \leftarrow \texttt{ctr} + 1$;
19: **if** $\texttt{ctr} \bmod \ell_B = 0$ **then**
20: $\mathsf{H}.update(\texttt{msg})$;
21: **if** $\texttt{ctr} \bmod \ell_E = 0$ **then**
22: $\texttt{otp} \leftarrow \mathsf{AES}_{EK}(\texttt{seq}_R||\texttt{ctr}/\ell_E||00\ldots0)$;
23: goto MAIN;
24: END :
25: On $\langle\texttt{input}\rangle = \texttt{S}$: goto START;
26: On $\langle\texttt{input}\rangle \in \mathcal{D}$:
27: **if** $\texttt{ctr} + \texttt{ctr}_M \geq \ell_M$ **then**
28: $\langle\texttt{output}\rangle \leftarrow \texttt{buf}[\texttt{ctr} + \texttt{ctr}_M \bmod \ell_M]$;
29: $\texttt{buf}[\texttt{ctr} + \texttt{ctr}_M \bmod \ell_M] \leftarrow \langle\texttt{input}\rangle$;
30: $\texttt{ctr}_M \leftarrow \texttt{ctr}_M + 1$;
31: **if** $\texttt{ctr}_M \neq \ell_M$ **then**
32: goto END;
33: $\texttt{buf} \leftarrow \texttt{buf} \ll_R (\texttt{ctr} \bmod \ell_M)$;
34: **if** $\texttt{mac} = \texttt{buf}$ **then**
35: $\langle\texttt{output}\rangle \leftarrow \texttt{E}$;
36: $\texttt{seq}_R \leftarrow \texttt{seq}_R + 1$;
37: goto INIT;
38: **else**
39: $\langle\texttt{output}\rangle \leftarrow \texttt{err}[0]$; //Calculation of `err` is not shown for clarity
40: $\langle\texttt{output}\rangle \leftarrow \texttt{err}[1]$;
41: $\langle\texttt{output}\rangle \leftarrow \texttt{E}$;
42: goto SYNC;
43: SYNC :
44: On $\langle\texttt{input}\rangle = \texttt{S}$: goto START;
45: On $\langle\texttt{input}\rangle \in \mathcal{D}$:
46: $\texttt{seq}[\texttt{ctr}_S] \leftarrow \langle\texttt{input}\rangle$;
47: $\texttt{ctr}_S \leftarrow \texttt{ctr}_S + 1$;
48: **if** $\texttt{ctr}_S \neq \ell_S$ **then**
49: goto SYNC;
50: **if** $\texttt{seq} > \texttt{seq}_R$ and $\mathsf{HMAC}_{MK}(\texttt{seq}||\texttt{digest}) = \texttt{buf}$ **then**
51: $\texttt{seq}_R \leftarrow \texttt{seq} + 1$;
52: goto INIT;