PillarBox: Combating next-generation malware with fast forward-secure logging

Kevin D. Bowers
RSA Laboratories
Cambridge, USA

Catherine Hart
Bell Canada
Vancouver, Canada

Ari Juels
Cornell Tech
New York, USA

Nikos Triandopoulos
RSA Laboratories
Cambridge, USA

September 17, 2014

Abstract

Security analytics is a catchall term for vulnerability assessment in large organizations capturing a new emerging approach to intrusion detection. It leverages a combination of automated and manual analysis of security logs and alerts which originate from a wide and varying array of sources and are often aggregated into a massive data repository. Such log and alert sources include firewalls, VPNs, and endpoint instrumentation, such as intrusion-detection systems, syslog or other alerting host facilities, that we generically call *Security Analytics Sources (SASs)*.

Security analytics are only as good as the data being analyzed. Yet nearly all security analytics systems today suffer from a lack of even basic protections on data collection. By merely monitoring network traffic, an adversary can *eavesdrop* on SAS outputs to *discover* sensitive SAS instrumentation and security-alerting behaviors. Moreover, by using advance malware, an adversary can *undetectably suppress* or *tamper* with SAS messages to conceal attack evidence and disrupt intrusion detection.

We introduce *PillarBox*, a tool for securely relaying SAS data in a security analytics system. PillarBox enforces *integrity*: It secures SAS data against tampering, even when such data is buffered on a compromised host within an adversarially controlled network. Additionally, PillarBox (optionally) offers *stealth*: It can conceal SAS data, alert-generation activity, and potentially even alerting rules on a compromised host, thus hiding select SAS alerting actions from an adversary.

We present an implementation of PillarBox and show experimentally that it can secure messages against attacker suppression, tampering or discovery even in the most challenging environments where SASs generate real-time security alerts related to a host compromise directly targeting to diminish their alerting power. We also show, based on data from a large enterprise and on-host performance measurements, that PillarBox has minimal overhead and is practical for real-world security analytics systems.

1 Introduction

Big data security analytics is a popular term for the growing practice of organizations to gather and analyze massive amounts of security data to detect systemic vulnerabilities and intrusions, both in real-time and retrospectively. 44% of enterprise organizations today identify their security operations as including big data security analytics [33]. To obtain data for such systems, organizations instrument a variety of hosts with a range of *Security Analytics Sources* (SASs) (pronounced “sass”). By SAS here, we mean generically a system that generates messages or alerts and transmits them to a *trusted server* for analysis and action.

On a host, for instance, a SAS can be a Host-based Intrusion Detection System (HIDS), an anti-virus engine, any software facility that writes to syslog, or generally any eventing interface that reports events to a remote service, e.g., a Security and Information Event Monitoring (SIEM) system. Further afield, a SAS could be a dedicated Network Intrusion Detection System, or, in an embedded device, a feature that reports physical tampering. A SAS could also be the reporting facility in a firewall or proxy.

SASs play a central role in broad IT defense strategies based on security analytics, furnishing the data to detect systemic vulnerabilities and intrusions. But a big data security analytics system is only as good as the SAS data it relies on. Worryingly, current-generation SASs lack two key protections against a local attacker.

First, an attacker can *undetectably suppress or tamper with* SAS messages. Today's approach to securing SAS messages is to transmit them immediately to a trusted server. By disrupting such transmissions, an attacker can create false alarms or prevent real alarms from being received. Even a SAS with a secure host-to-server channel (such as SSL/TLS) is vulnerable: An attacker can undetectably blackhole or suppress transmissions until it fully compromises the host, and then break off SAS communications. (We demonstrate the feasibility of such an attack in this paper in Section 6.3.) And logged or buffered SAS messages are vulnerable to deletion or modification after host compromise.[1]

Consider, for instance, a rootkit Trojan that exploits a host vulnerability to achieve a privilege escalation on an enterprise host. A HIDS or anti-virus engine might immediately detect the suspicious privilege escalation and log an alert, "Privilege Escalation." An attacker can block transmission of this message and, once installed, the rootkit can modify or remove critical logs stored locally (as many rootkits do today, e.g., ZeroAccess [14, 15], Infostealer.Shiz [10], Android.Bmaster [1]).[2] Because any buffered alert can simply be deleted, and any transmission easily blocked, an enterprise server receiving the host's logs will fail to observe the alert and detect the rootkit.

A second problem with today's SASs is that an attacker can *discover* intelligence about their configuration and outputs. By observing host emissions on a network prior to compromise, an attacker can determine if and when a SAS is transmitting alerts and potentially infer alert-generation rules. After host compromise, an attacker can observe host instrumentation, e.g., HIDS rule sets, logs, buffered alerts, etc., to determine the likelihood that its activities have been observed and learn how to evade future detection.

For enterprises facing sophisticated adversaries, e.g., *Advanced Persistent Threats* (APTs) such as Operation Aurora [2, 3], the Stuxnet attack [5, 7, 13], the RSA breach [11, 12], and attacks related to malware Duqu [8], such shortcomings are critical. Threat-vector intelligence is widely known to play a key role in defense of such attacks, and its leakage to cause serious setbacks [30].

Thus an attacker's ability to suppress alerts undetectably and obtain leaked alert intelligence in today's SAS systems is a fundamental vulnerability in the host-to-server chain of custody and a considerable flaw in big data security analytics architectures.

Pillarbox. As a solution to these challenges, we introduce a tool called *PillarBox*.[3] PillarBox securely relays alerts from any SAS to a trusted analytics server. It creates a secure host-to-server chain of custody with two key properties:

1. **Integrity:** PillarBox protects a host's SAS messages against attacker tampering or suppression. It guarantees that the server receives all messages generated *prior* to host compromise (or detects a malicious system failure). PillarBox also aims to secure real-time alert messages *during* host compromise faster than the attacker can intercept them. *After* host compromise, PillarBox protects already generated SAS messages, even if an attacker can suppress new ones.

[1]To the best of our knowledge, we are unaware of any cryptographic integrity protection on messages at rest in today's SIEM products.

[2]These are just recent examples. Many rootkits remove or obfuscate logs by modifying the binary of the logging facility itself.

[3]A *pillar box* is a Royal Mail (U.K.) mailbox in the form of a red metal pillar. It provides a secure and stealthy chain of custody, with integrity (only postal workers can open it), message hiding (it's opaque), and delivery assurance (if you trust the Royal Mail).

2. **Stealth:** As an optional feature, PillarBox conceals when and whether a SAS has generated alerts, helping prevent leakage of intelligence about SAS instrumentation. It does so against an attacker that can see network traffic before compromise and learns all host state after compromise. Stealth can also involve making SAS alert-generation rules *vanish* (be erased) during compromise.

Counterintuitively, PillarBox *buffers SAS messages on the (vulnerable) host*. As we show, this strategy is better than pushing alerts instantly to the server for safekeeping: It's equally fast, more robust to message suppression, and important for stealth.

Challenges. While PillarBox is useful for any type of SAS, the most stringent case is that of *self-protection*, which means that *the SAS messages to be protected regard the very host producing the messages*, potentially while the host is being compromised (as with, e.g., a HIDS). Thus, integrity has two facets. First, a host's buffered alerts must receive ongoing integrity protection even *after host compromise*. Second, alerts must be secured *quickly*—before an attacker can suppress or tamper with them as it compromises the host. We show experimentally that even in the most challenging case of self-protection, PillarBox secures SAS alerts before a fast attacker can suppress them—*even if the attacker has full knowledge of and explicitly targets PillarBox*.

Stealth (optional in PillarBox) requires that the host's internal data structures be invariant to SAS message generation, so that they reveal no information to an attacker after host compromise. Message buffers must therefore be of fixed size, making the threat of overwriting by an attacker an important technical challenge. Additionally, to protect against an adversary that controls the network, stealth requires that PillarBox transmissions resist traffic analysis, e.g., do not reveal message logging times. A final challenge in achieving stealth is the fact that an attacker that compromises a host learns the host's current PillarBox encryption keys.

Contributions. In this paper we highlight and demonstrate the transmission vulnerability in security analytics systems and propose a solution, which we call PillarBox. In designing PillarBox, we also specify (and formally define) the properties of integrity and stealth, which are general and fundamental to the architecture of any security analytics system. We show how to combine standard forward-secure logging and activity-concealment techniques to simultaneously achieve both properties in the self-protection SAS mode of operation.

We present an architecture for PillarBox and a prototype end-to-end integration of the tool with syslog, a common SAS. We show experimentally that PillarBox can secure SAS messages in the challenging self-protection case before an attacker can suppress them by killing PillarBox processes. Since the majority of host compromises involve privilege escalation, we also show that for a common attack (the "Full-Nelson" privilege escalation attack), an alerter can be configured to detect the attack and the resulting SAS message can be secured before the attacker can shut down PillarBox. Additionally, we use alert-generation data from a large enterprise to confirm that PillarBox can be parameterized practically, with low performance overhead on hosts.

We emphasize that we do not address the design of SASs in this paper. That is, we neither propose new ways of generating intrusion detection alerts nor explore the direct efficacy of secure alerting in intrusion and malware detection. How SAS messages are generated or consumed and their exact content are outside the scope of this paper. Indeed, PillarBox is an alert relaying tool which is agnostic to the content of alerts and how they are generated. Instead it can be viewed as a black box[4] that can provide additional value to any currently deployed security analytics system by serving as a designated secure transmission channel between

[4]PillarBox is a black box both senses; firstly in that the SAS does not need to know how it works to use it, and secondly in the sense that data stored in the black box is assumed to be trustworthy and remain secret.

any existing (alert-generating) SAS and any existing corresponding (alert-collecting and consuming) SIEM that guarantees the properties of integrity and (optionally) stealth. Therefore, PillarBox is a practical, general tool to harden the host-to-server chain of custody for any SAS, providing a secure foundation for security analytics systems.

Organization. Section 2 introduces PillarBox's adversarial model and design principles, while Section 3 describes its architecture and integration with a SAS. Section 4 gives technical details on buffer construction and supporting protocols. Sections 5 and 6 present a prototype implementation and experimental evaluation of PillarBox. We review related work in Section 7 and conclude in Section 8. Cryptographic formalisms related to the security model and tools over which PillarBox's design is based are relegated to the Appendix.

2 Modeling and design principles

We now describe the adversarial model within which PillarBox operates. We then explain how host-side buffering serves to secure SAS alerts within this model and follow with details on the technical approaches in PillarBox to achieving integrity and stealth.

2.1 Threat model

Our threat model considers three entities, the *SAS* (which we refer to interchangeably as the *host*),the *attacker*, and the *server*. We model the attacker to be the strongest possible adversary, one attacking a host in the self-protecting setting. (Achieving security against this strong adversary ensures security against weaker ones, e.g., those attacking only the network or a firewall whose SAS only reports on network events.)

Recall that in the self-protecting case, a SAS reports alerts about the host itself: While the compromise is taking place, the SAS generates one or more alert messages relevant to the ongoing attack and attempts to relay them to the server.

The adversary controls the network in the standard Dolev-Yao sense [20]: That is, it can intercept, modify, and delay messages at will. When its intrusion is complete, the attacker achieves what we call a *complete compromise* of the host: It learns the host's complete state, including all memory contents—cryptographic keys, buffer messages, etc.—and fully controls the host's future behavior, including its SAS activity.

To violate integrity, the attacker's goal is to compromise the host: (1) Without any unmodified alerts reaching the server and (2) Without the server learning of any modification or suppression of alerts by the attacker.

The SAS can only start generating meaningful alerts, of course, once the intrusion is in progress. After the attacker has achieved complete compromise, it can shut down the SAS or tamper with its outputs. So a SAS produces valid and trustworthy alerts only *after* intrusion initiation but *prior* to complete compromise. We call the intervening time interval the *critical window* of an attack, as illustrated in Figure 1. This is the interval of time when intrusions are detectable and alerts can be secured (e.g. buffered in PillarBox) before the attacker intercepts them.

Conceptually, and in our experiments, we assume that the attacker has full knowledge of the workings of the SAS, including any mechanisms protecting alerts en route to the server, e.g., PillarBox. It fully exploits this knowledge to suppress or modify alerts. The attacker doesn't, however, know host state, e.g., cryptographic keys, prior to complete compromise nor does it know the detection rules (behavioral signatures) used by the SAS, i.e., the precise conditions leading to alert generation.

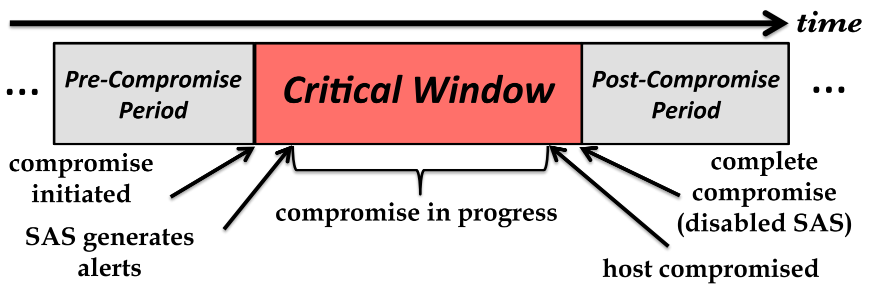

Figure 1: Event timeline of host compromise

To violate stealth, the attacker tries to learn information about SAS rules and actions, e.g., if the SAS has issued alerts during an attack, by making adaptive use of the network and of post-compromise host state, such as buffer state. SAS detection rules can also be used to infer behavior, but are outside the scope of PillarBox. However, as we discuss below, *vanishing rules*, that is, rules that are deleted if they ever trigger an alert, can be used to protect against adversarial rule discovery in the SAS.[5] By analogy with cryptographic indistinguishability definitions, a concise definition of stealth is possible: The stealth property holds if, for any SAS detection rule and any attack strategy, no attacker can distinguish between PillarBox instantiations with and without the rule (and, thus, between the associated observed SAS alerting activities).

We next describe the design principles used in PillarBox, gaining intuition behind the importance of alert buffering in building a secure alert relaying tool, as well as insight about the technical challenges introduced by alert buffering when trying to preserve integrity and stealth.

2.2 Secure alert relaying via buffering

A key design choice in PillarBox, as mentioned, is the use of host-side alert buffering. For brevity, we refer to the *PillarBox buffer* where alerts are secured as the *PBB*. The objective of PillarBox is to secure alerts in the PBB during the critical window, as shown in Figure 2. Once in the PBB, alert messages are protected in two senses: They are both integrity-protected and "invisible" to the attacker, i.e., they support systemic stealth. (Informally, the PBB serves as a "lockbox.") Also, as we explain, either alerts reliably reach the server, or the server learns of a delivery failure.

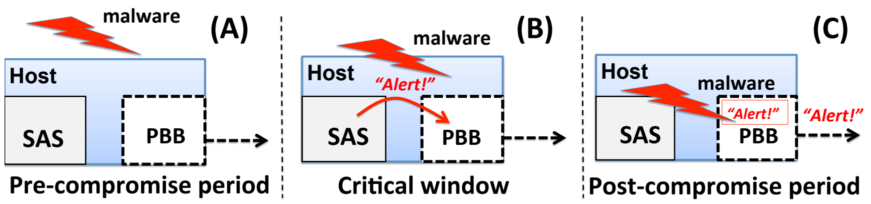

Figure 2: PillarBox across compromise phases: (A) The host has not yet been attacked. (B) The SAS detects in-progress compromise and places an alert in the PillarBox buffer PBB. (C) The host is under the attacker's full control, but PBB securely stores and transmits the alert.

We first explain why buffering is important to secure the SAS chain of custody in PillarBox and then how we address the technical challenges it introduces.

[5] Vanishing rules operate in the same adversarial model: The SAS tries to erase rules during the critical window, while the attacker tries to learn them before they are erased.

Why buffering is necessary. The approach of most SAS systems today, e.g., syslog and HIDSs, is to push alerts to a remote server in real time, and thus secure them at the server during the critical window. There are many important cases, though, both adversarial and benign, in which SAS messages *cannot be pushed reliably*, for two main reasons:

- *Imperfect connectivity:* Many host SAS systems lack continuous connectivity to the server. For instance, laptops that shuttle between an office and home have limited connection with corporate security servers but are open to infection in the home. Lightweight embedded devices often cannot ensure or even verify delivery of transmitted messages. For instance, wireless sensor networks[6] often experience transmission failures.

  Even fully connected systems can experience benign transport failures. For instance, syslog messages often travel over UDP, which doesn't guarantee packet delivery [32].

- *Network attacks*: An attacker can actively suppress on-the-fly SAS transmissions by causing malicious network failures. It can selectively disrupt network traffic, via, e.g., ARP stack smashing (see, e.g., [4, 6]), or flood target hosts to achieve denial-of-service (DoS) during a compromise, causing message delay or suppression. The result is complete occlusion of server visibility into the critical window—potentially appearing to be a benign network failure. We describe our own implementation of such an alert-suppression attack later in Section 6.3.

Even if reliable, immediate SAS message-pushing were generally feasible, it would *still* have an undesirable effect:

- *SAS intelligence leakage:* If a host pushes alerts *instantaneously*, then its outbound traffic reveals SAS activity to an attacker monitoring its output. An attacker can then probe a host to learn SAS detection rules and/or determine after the fact whether its intrusion into a host was detected. Note that encryption does not solve this problem: Traffic analysis alone can reveal SAS rule-triggering. (As noted above, PillarBox overcomes this problem via regular alert-buffer transmission.)

- *Delayed alert custody:* SASs today push alerts on to the wire as quickly as possible to remove them from a host under attack and secure them on a trusted server. But in Section 6.2, we show experimentally that it is generally *faster* to secure alerts in a buffer *on the host*. In fact, PillarBox secures alerts even faster than (unreliable and unprotected) UDP transmissions.

For these reasons, *message buffering*—as opposed to on-the-fly event-triggered transmission—is of key importance in a SAS chain of custody and the cornerstone of PillarBox. Buffering SAS messages, though, poses security challenges. If an attacker completely compromises a host, there is no way of course to prevent it from disabling a SAS or tampering with its *future* outputs. But there is a separate problem after host compromise: Inadequately protected buffered SAS messages are vulnerable to modification/suppression and intelligence leakage, as discussed before. We next elaborate on these problems and our solution to them in PillarBox.

[6]These systems do not have full-blown SASs but can have lightweight versions, e.g., the hardware tripwire proposed for authentication tokens [23].

2.3 Achieving integrity

A main challenge in creating a secure chain of custody in PillarBox is the need to secure alert messages *after* compromise, while they are still buffered and exposed to an attacker. Log-scrubbing malware can attempt to modify buffered alerts (e.g., replace the strong alert "Privilege Escalation" with the more benign "Port Scan Observed") or just purge alerts. *Post-compromise integrity protection* for buffered SAS messages is thus crucial in PillarBox—but at first glance, this might seem unachievable.

Indeed, a digital signature or message-authentication code (MAC) alone, as proposed, e.g., for syslog [26], does not protect against tampering: After host compromise an attacker learns the signing key and can forge messages. Message encryption similarly does not protect messages against deletion, nor does tagging them with sequence numbers, as an attacker with control of a host can forge its own sequence numbers.

Fortunately, post-compromise alert integrity is achievable using the well-known cryptographic technique of *forward-secure* integrity protection. The main idea is to generate new keys on the host after every alert generation and delete keys immediately after use. Forward-secure integrity protection is commonly used for forward-secure logging (e.g., [29, 35, 43, 44]), an application closely related to SAS protection.[7] Similarly, PillarBox uses forward-secure pseudorandom number generation (FS-PRNG) to create MAC keys. Each MAC key is used to secure a single message at the PillarBox host and is then deleted. An FS-PRNG has the property that past keys cannot be inferred from a current key, preventing tampering of messages that have already been secured. The server runs this FS-PRNG to compute the (same series of indexed) shared keys with the host, allowing it to detect tampering or erasure.

What's new in the use of forward security in PillarBox is primarily its application for self-protecting alerting: Indeed, recall that the main aspect of integrity here is securing alerts in the PBB as fast as possible during a compromise, i.e., in the critical window. Effectively, PillarBox engages in a race to secure alerts before the attacker intercepts them. Ensuring that PillarBox can win this race is not a matter of cryptography, but of the system design of PillarBox, including the design choice of host-side buffering! An important contribution of our work is an experimental validation that winning this race, and thus the whole PillarBox approach to securing alerts, is feasible. Our experiments in Section 6 show this to hold even against a fast, local, PillarBox-aware attacker that tries to kill PillarBox processes as quickly as possible.

2.4 Achieving stealth

Stealth, as we define it, requires concealment of the entire alerting behavior of a SAS, including detection rules, alert message contents, alert generation times, and alert message existence in compromised hosts. Stealth is a key defense against sophisticated attackers. (One example: Host contact with "hot" IP addresses can help flag an APT, but an attacker that learns these addresses can just avoid them [30].)

Achieving stealth is not possible with direct use of existing techniques; in particular, straightforward encryption alone does not achieve stealth. To protect against buffer captures, encryption of alerts alone (forward-secure or otherwise) cannot help: For example, if buffer alerts are encrypted on a host, an attacker can infer alert generation simply by counting buffer ciphertexts upon host compromise, thus potentially inferring whether an alert was triggered during compromise. Similarly, encrypted host-to-server traffic leaks information: An attacker that does not just compromise hosts, but also controls the network, can determine via traffic analysis when a host has triggered an alert. An attacker that has monitored past buffer transmissions can even perform a black-box probing attack against a host to discover SAS detection rules:

[7]Forward-secure logging systems, however, are designed mainly for forensic purposes rather than detection, e.g., they are designed to protect against administrator tampering after the fact. Because of this, some such systems "close" logs only periodically.

By inspecting the state of the buffer the attacker can determine when *some* SAS rule was triggered, and thus test specific attacks to infer which are or are not detectable. Instead, stealth in PillarBox requires a combination of several ideas.

In particular, PillarBox employs a buffer size T, and buffer transmission-time interval μ, that are *both fixed*, i.e., invariant. Each message is also of fixed size (or padded to that size). (We discuss the option to have variable-sized messages in Section 4.) When PillarBox transmits, it re-encrypts and sends the *entire* fixed-size buffer, not just fresh (encrypted) alerts. Such fixed-length transmissions prevent an attacker from determining when new alerts have accumulated in the host buffer, while its fixed communication patterns defeat traffic analysis. Otherwise, information about alert generation can be leaked to an attacker who harvests past host-to-server buffer transmissions before fully compromising the host: For instance, if only new alerts are transmitted, the transmission will itself leak information about alert-generation patterns, and if the buffer size is variable, whether the host compromise fired an alert will be inferred by examining the host's current buffer state in relation to older transmissions.

As the host buffer is of fixed size T, PillarBox writes messages to it in a round-robin fashion. Thus *messages persist in the buffer until overwritten*. (They may be transmitted multiple times. Such persistent transmission consumes bandwidth, but has a potentially useful side effect as it may allow temporarily suppressed messages to eventually reach the server.) This feature of fixed-size buffer in PillarBox creates a need for careful parameterization: We must ensure that T is large enough to hold all alert messages generated under benign conditions within a time interval μ; this condition also ensures that the buffer is also large enough so that if round-robin overwriting occurs, PillarBox implicitly signals to the server a "buffer-stuffing" attempt by an attacker. (Below, in Section 3, we develop a framework for parameterization of T and μ and then, in Section 6, we explore practical settings by analyzing real-world alert transmission patterns in a large enterprise.)

PillarBox generates encryption keys in a forward-secure way (using an FS-PRNG) to protect against decryption attacks after an attacker compromises a host's keys. To protect against an attacker that controls the network and eventually the host as well, encryption is applied in *two layers*: (1) To *buffered messages*, to ensure confidentiality after host compromise, and (2) to *host-to-server buffer transmissions*, to ensure against discovery of alert data from buffer ciphertext changes. Note that buffer encryption alone is insufficient: E.g., if identical buffer ciphertexts leave a host twice, the attacker learns that no new alert has been generated in between. Also note that semantically secure public-key encryption would enable use of just one layer, but with impractically high cost overheads for PillarBox.

Complete stealth in PillarBox carries an unavoidable cost: Periodic rather than immediate transmissions can delay server detection of intrusions. But we note that stealth is an optional feature in PillarBox. It can be removed or weakened for limited attackers.

Vanishing detection rules. Finally, we note that a key complement to stealth in PillarBox is concealment of detection rules in hosts: Indeed, encryption alone (in our work at two layers) ensures confidentiality in the buffer and over the network, but not in the SAS alerting engine itself. There, deleting a detection rule from the SAS as soon as it triggers an alert for the very first time offers an effective concealment method. In our experiments in Section 6, we show the viability of instrumenting the SAS with vanishing rules.

3 Architecture

We next describe PillarBox's general architecture, main software components and operating configuration, used to secure the host-to-server chain of custody in a SAS system.

As explained, PillarBox secures alerts through buffering: The PillarBox buffer, called PBB for short, is the cornerstone in our design that serves as a special "lockbox" which protects at rest or in transit alerts against tampering, suppression or leakage, even in post-compromise environments. PillarBox combines simple cryptographic protections (namely, forward secure MAC and encryption) with a data structure and buffer transmission protocol for the PBB that are invariant to the number of alert messages and their buffering history. In essence, this implements a low-level crypto-assisted reliable channel that, even under adversarial conditions, ensures tamper-evident and stealthy delivery of transmitted messages to a remote server. Details on the PBB design, its associated protocols as well as their properties are provided in Section 4.

PillarBox helps a SAS transmit security alerts to a remote, trusted service. This service can perform analysis and remediation that is impractical on hosts themselves. As it operates on a device other than the host, it is (generally) insulated from the intrusion. Additionally, the service can correlate events across multiple reporting hosts, enabling it to filter out false positives. It can also house sensitive threat intelligence that cannot be safely exposed on hosts and can trigger action by human system administrators or Security Operations Center (SOC) personnel.

3.1 Interface with SAS

Being agnostic to message content, PillarBox works with any SAS. It can serve as the main channel for SAS alerts or can deliver SAS alerts selectively and work in parallel with an existing transport layer. Exactly how SAS messages are produced at the host or consumed at the receiving server depends on SAS instrumentation and alert-consuming processes. (As such, it is outside the scope of our work.)

Similarly, our architecture abstracts away the communication path between the host and server, which can be complicated in practice. In modern enterprises, networks carry many SAS-based security controls that alert upon malfeasance. Typically, alerts are sent via unprotected TCP/IP transmission mechanisms, such as the syslog protocol (which actually uses UDP by default), the Simple Network Messaging Protocol (SNMP), or the Internet Control and Messaging Protocol (ICMP). These alerts are typically generated by endpoint software on host systems (such as anti-virus, anti-malware, or HIDS) or by networked security control devices. These devices are commonly managed by a SIEM systems, which may be monitored by human operators (e.g., by SOC staff) on a continuous basis. For the purposes of our architecture, though, we simply consider a generic SAS-instrumented host communicating with a server.

Alerter. We refer generically to the SAS component that generates alert messages as an *alerter* module.[8] This module monitors the host environment to identify events that match one of a set of specified alert *rules*. When an event triggers a rule, the alerter outputs a distinct alert message. An alert template may either be static (predefined at some setup time for the host) or dynamic (updated regularly or on-demand through communication with the server). Rules may take any form. They may test individual state variables (specified as what is generally called a *signature*) or they may correlate more than one event via a complicated predicate or classifier. As mentioned before, the SAS may tag select rules as "vanishing." When such a rule is triggered, it is erased from the current rule set to further enhance the stealth properties provided by PillarBox.

In our basic architecture, the alerter's interface with PillarBox is unidirectional. The alerter outputs alert messages, and PillarBox consumes them. Although many architectures are possible, given PillarBox's emphasis on critical alerts, in our canonical operational setting, the SAS may send only *high severity* messages

[8] Of course, a SAS (e.g., HIDS or anti-virus software) includes other components, e.g., a transport layer, update functionality, and so forth.

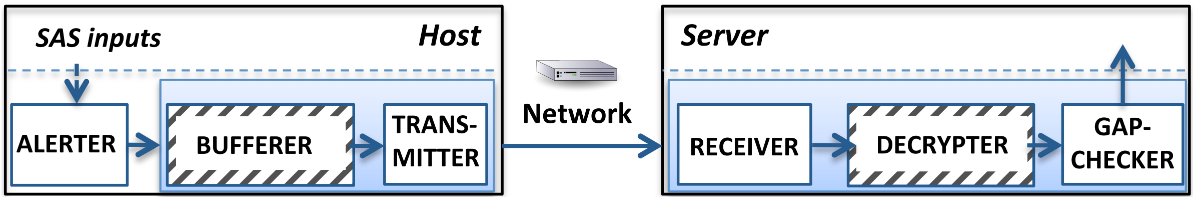

Figure 3: PillarBox architecture and data flow. Shaded areas show the PillarBox components (which exclude the alerter) and striped ones (the bufferer and decrypter) comprise those that make up PillarBox's crypto-assisted core reliable channel.

(e.g., those that seem to indicate impending compromise) to PillarBox, and relay regular logs through its ordinary low-priority transport layer.

3.2 PillarBox components

The general message flow in PillarBox is fairly simple. Most of the complexity is hidden by the PBB "lockbox." PillarBox consists of five modules, shown in Figure 3.

Bufferer. This module controls the core message buffer, the PBB (which is detailed in Section 4). It accepts two calls: A Write call from the alerter to insert a message into the PBB (in encrypted form) and a Wrap call from the transmitter—described below—requesting export of the current buffer contents (also in a securely encapsulated form). The bufferer responds to message-insertion requests by encrypting and storing the new message in the PBB and accordingly updating the secret PBB state, and to buffer-wrapping requests by further encrypting (as a whole) the current contents of the PBB and returning the corresponding ciphertext. This module is also responsible for maintaining the secret state of the PBB and updating cryptographic (MAC and encryption) keys, which are effectively used to securely label messages and buffers with sequence numbers. The bufferer does not discard messages from the buffer when they are transmitted: A message is encapsulated and transmitted until overwritten, offering the extra feature of persistence. This byproduct of stealth can be leveraged to accommodate lossy networks, as explained later.

Transmitter. This module schedules and executes buffer transmissions from the host to the server. Transmissions may be scheduled every μ seconds, like a "heartbeat." The module sends Wrap requests to the bufferer and transmits encapsulated buffers to the server over the network using any suitable protocol.

Receiver. This module receives encapsulated-buffer transmissions on the server from the host-based transmitter over the network. When it receives a transmission pushed from the host, it relays it with a Read instruction to the decrypter.

Decrypter. In response to a Read request from the receiver, the decrypter decrypts and processes an encapsulated buffer. It verifies the buffer's integrity and either outputs its constituent messages, or else outputs a $\perp$ symbol indicating a buffer corruption. It also labels the buffer and its messages with their corresponding (verified) sequence numbers.

Gap-checker. The gap-checker's main task is to look for lost messages in the SAS message stream, which cause it to output an alert that we call a *gap alert*. These may be caused by one of two things: (1) A flood of alerts on the host (typically signaling an intrusion) or (2) Overwriting of alerts in the buffer by malicious buffer-stuffing on the compromised host. (We detail these attacks in Section 4.) As messages are labeled with verified sequence numbers, gap checking requires verification that no sequence numbers go missing in the message stream. Because messages continue to be transmitted until overwritten, note that in normal operation sequence numbers will generally *overlap* between buffers. The gap-checker can optionally filter out redundant messages. To detect an attacker that suppresses buffer transmission completely, the gap-checker also issues an alert (namely, a transmission-failure alert) if buffers have stopped arriving for an extended period of time, as we discuss below.

3.3 Parameterizing PillarBox

The gap-checker always detects when a true gap occurs, i.e., there are no false-negatives in its gap-alert output. To ensure a low false-positive rate, i.e., to prevent spurious detection of maliciously created gaps, it is important to calibrate PillarBox appropriately.

The size T of the PBB dictates a tradeoff between the speed at which alerts can be written to the buffer and the rate at which they must be sent to the server. Let τ denote an estimate of the maximum number of alerts written by the host per second under normal (non-adversarial) conditions. Then provided that the encapsulation or heartbeat interval μ (the time between "snapshots" of buffers sent by the host) is at most T/τ seconds, a normal host will not trigger a false gap alert.

We characterize τ, the maximum SAS message-generation rate of normal hosts, in Section 6. Using a moderate buffer size T we are able to achieve extremely low false-positive gap-alert rate in most cases.

In networks vulnerable to message loss, the persistence feature of PillarBox can be useful: The larger T (or the smaller μ), the more repeated transmissions of every message. In particular, this feature is desirable to accommodate normal network disruptions as well as adversarial attempts to block network messages: If an inter-transmission time or heartbeat interval μ (which is less than T/τ) is used (thus, making the transmission rate $1/\mu$), then whenever the actual alerting rate drops to τ/k, for some $k > 0$, every message will be *transmitted at least* k *times*. Accordingly, if the server successfully receives encapsulated buffers at a rate higher than τ/T, it can *downsample for efficiency*.

Handling message disruptions. If an attacker suppresses buffer transmission completely, the gap-checker will cease to receive buffers. The gap-checker issues a *transmission-failure alert* if more than β seconds have elapsed without the receipt of a buffer, for parameter setting $\beta > T/\tau$. This is the task of the gap-checker, rather than the receiver or decrypter, as only the gap-checker can identify situations in which buffers arrive, but are replays, and thus a transmission-failure alert is appropriate. We note that a PillarBox deployment can in principle perform a more sophisticated check on buffer-transmission stoppage than the timeout β alone. For instance, if a device is displaying network activity, but is not transmitting alerts as expected, that is an additional evidence for a SAS failure, malicious or otherwise.

PillarBox cannot itself distinguish benign from adversarial transmission failures (although network liveness checks can help). While there are many possible policies for transmission-failure alerts, in reliable networks, PillarBox is best coupled with an access policy in which a host that triggers a transmission-failure alert after β seconds is *disconnected from network services other than PillarBox*. Any disconnected services are restored only when PillarBox's decrypter again receives a buffer from the host and can detect alerts. In a benign network outage, this policy will not adversely affect hosts: They will lack network service anyway. An adversary that suppresses PillarBox buffer transmission, though, will cut itself off from the network until PillarBox can analyze any relevant alerts. Such interfacing of PillarBox with network-access policies *limits the attackers ability to perform online actions while remaining undetected*, as it effectively caps the maximum possible interval of lost visibility for PillarBox.

4 PillarBox buffer and protocols

We now present the main component of PillarBox, the PBB, and its protocols (run by the bufferer and the decrypter modules), which realize a reliable messaging channel for PillarBox. We also discuss the functionality that the PBB exports to the alerter and the gap-checker modules to secure the SAS chain of custody. Overall, as we have mentioned earlier, PillarBox relies on a simple, novel, crypto-assisted "lockbox" which we now describe in detail. (The reader uninterested in the internals of these low-level functionality can skip this discussion without a loss of understanding of PillarBox.)

4.1 Ideal "lockbox" security model

Conceptually, in its *ideal* form, the PBB serves as a "lockbox" for message transport. We overview this simplified view of PBB as it helps better comprehend the core security properties of PillarBox. A formal description of this ideal lockbox model and a detailed discussion about its relation to PillarBox (as well as to existing forward-secure logging solutions) is given in Appendix A.

The PBB is a buffer consisting of T fixed-size slots that supports the following two basic operations:

1. `write`: The *sender* $\mathcal{S}$ (the client in PillarBox) inserts individual messages into the buffer via `write` in a *round-robin* fashion. Given currently *available* position $I \in \{0, \ldots, T-1\}$ (initially set at random), a new message is written in slot I (replacing the oldest message), and I is incremented by $1 \pmod{T}$.

2. `read`: The *receiver* $\mathcal{R}$ (the server in PillarBox) invokes `read`, which outputs the (monotonically increasing) sequence numbers j of the buffer and s_j of the last inserted message, along with the T messages in the buffer starting at position $s_j \bmod T$, with wraparound.

Messages buffered in this ideal "lockbox" can only be read via the `read` interface and can only be modified (authentically) via the `write` interface. When read by $\mathcal{R}$, a message m_i stored at slot i is guaranteed to be either the most recent message written to slot i (the empty message $\emptyset$ if no message was ever written), or a special corruption symbol $\perp$ that indelibly replaces all the buffer's contents if the buffer was tampered with or otherwise modified by `write`.

The goal of an attacker on compromising a host is to learn SAS actions and suppress alerts buffered during the critical window. The ideal `read` interface of the "lockbox" buffer protects against violations of stealth (the attacker cannot observe when $\mathcal{R}$ reads the buffer). Given the `write` interface, the attacker can only violate buffer integrity in the post-compromise period in one of four ways:

1. *Buffer modification/destruction:* The attacker can tamper with the contents of the buffer to suppress critical-window alerts. As noted above, this causes buffered messages to be replaced with a special symbol $\perp$.

2. *Buffer overwriting:* Although tampering with PBB contents will reliably produce a detectable message-integrity failure, the attacker can still write messages into the PBB buffer without destroying it: By writing T relatively benign messages, i.e., exploiting buffer wraparound to refill the PBB, it can *overwrite* and thereby destroy all messages generated during the critical window.

3. *Buffer dropping:* The attacker can simply drop buffers or delay their transmission.[9]

[9] The attacker can also potentially cause buffers to drop by means of a network attack during the critical window, but the effect is much the same as a post-compromise attack.

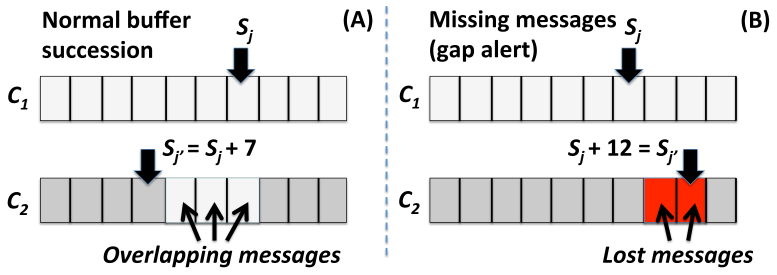

Figure 4: Gap rule example on successively received buffers C_1, C_2, indexed by sequence numbers j, j' and $T = 10$: (A) Normal message overlap between buffers; (B) A detectable gap: Messages with sequence numbers $s_j + 1$ and $s_j + 2$ have been lost.

4. *Transmission stoppage:* The attacker can break the PBB completely, causing no buffer transmission for an extended period of time, or indefinitely.

During the critical window, the attacker can alternatively try to attack so quickly that the critical window is nearly zero. In this case, there isn't sufficient time for PillarBox to take in a SAS alert message and put it in the PBB. Our experiments in Section 6 show in some settings of interest that this attack is unlikely.

Adversarial buffer modification or destruction, as explained above, is an easily detectable attack. It causes the server to receive a symbol $\perp$, indicating a cryptographic integrity-check failure. The gap-checker in PillarBox detects both buffer overwriting attacks and buffer dropping attacks by the same means: It looks for lost messages, as indicated by a gap in message sequence numbers. That is, a gap alert is issued when the sequence numbers s_j and $s_{j'}$ of (the last inserted messages in) two successively received buffers j and j' are such that $s_{j'} - s_j \geq T$. Figure 4 depicts a normal buffer transmission and one, ostensibly during an attack, in which messages have been lost to an alert flood or to buffer overwriting. A transmission stoppage is detectable simply when the server has received no buffers for an extended period of time, producing a transmission-failure alert, as noted above.

4.2 Security definitions

We now provide informal definitions for the security and system correctness properties that must hold in a real implementation of this ideal "lockbox." Formal definitions are given in Appendix B.

Our implementation of this ideal "lockbox" consists of the PBB and three operations: (i) The sender $\mathcal{S}$ runs Write to insert a message into the PBB and (ii) Wrap to encapsulate the PBB for transmission, and (iii) The receiver $\mathcal{R}$ runs Read to extract all messages from a received, encapsulated PBB. We denote by C the contents of the PBB after a series of Write operations by $\mathcal{S}$, and by $\hat{C}$ a cryptographic encapsulation transmitted to $\mathcal{R}$. In this operational setting, we require the security properties of *immutability* and *stealth*, as well as the non-cryptographic properties of *correctness* and *persistence*.

Informally, *correctness* dictates that under normal operation any sequence of messages of size at most T added to C by $\mathcal{S}$ can be correctly read by $\mathcal{R}$ in an order-preserving way; in particular, the T *most recent* messages of C and their *exact order* can be determined by $\mathcal{R}$. *Persistence* means that by encapsulating the buffer C repeatedly, it is possible to produce a given message in C *more than once*.

For our two cryptographic properties, we consider a powerful adaptive adversary $\mathcal{A}$ that operates in two phases: (1) Prior to compromise, $\mathcal{A}$ fully controls the network, and may arbitrarily modify, delete, inject, and re-order transmissions between $\mathcal{S}$ and $\mathcal{R}$; $\mathcal{A}$ may also determine what messages $\mathcal{S}$ writes to PBB and when $\mathcal{S}$ encapsulates and sends the PBB, and may also learn what pre-compromise messages $\mathcal{R}$ receives, and may finally choose its time of compromise; (2) On compromising $\mathcal{S}$, $\mathcal{A}$ corrupts $\mathcal{S}$, learns its secret state, and fully controls it from then on.

Immutability means, informally, that pre-compromise messages in C are either received unaltered by $\mathcal{R}$ in the order they were written, or are marked as invalid; i.e., even after compromising $\mathcal{S}$, $\mathcal{A}$ cannot undetectably drop, alter or re-order messages in C. *Stealth* means, informally, that $\mathcal{A}$ cannot learn any information about messages buffered prior to compromise (that were not explicitly determined by $\mathcal{A}$). It is stronger than confidentiality. Not only cannot $\mathcal{A}$ learn the contents of messages, it also cannot learn the number of buffered messages—or if any were buffered at all. This holds even after $\mathcal{A}$ has compromised $\mathcal{S}$.

We define stealth through a security experiment analogous to IND-CCA2 (indistinguishability under an adaptive chosen ciphertext attack): $\mathcal{A}$ chooses two challenge messages m_0, m_1; optionally, one may be blank, signifying "no message insertion." For a randomly selected bit b, message m_b is placed in PBB just prior to compromise; $\mathcal{A}$ breaks stealth if he can guess b with (non-negligible) probability greater than 1/2. (As noted above, $\mathcal{A}$ can learn messages received by $\mathcal{R}$ in the pre-compromised phase of the experiment.)

4.3 Intuition behind construction

Our lockbox construction uses standard forward-secure cryptographic primitives to secure buffered messages against capture of the PBB after a host's full compromise: Secret keys, used in tools providing integrity and confidentiality on transmitted messages, *evolve* over time so that no current key reveals past ones. Forward security is used widely in existing literature to protect the integrity and confidentiality of security logs (e.g., [29, 35, 43, 44]). Our key observation, though, is that forward security alone *cannot achieve stealth or persistence*. In particular, existing forward-secure logging solutions make no attempt to conceal the number of accumulated messages. These solutions employ an *append-only* buffer that is *read once* and can be *arbitrary long* in size. Indeed, authenticated and/or encrypted messages are typically organized in a linked list of arbitrary size, where any new message is added at the end of the list. This append-only technique, however, clearly does not achieve stealth: An adversary can, for instance, infer message-writing behavior from the length of buffer transmissions. Additionally, in such schemes, a Wrap operation typically sends to $\mathcal{R}$ all messages in the list, which are then deleted from the list, thus not achieving persistence, as individual messages can only be transmitted once.

To improve upon these limitations, we imbue our lockbox construction with the following simple but important design features which, in turn, provide stealth and persistence. To enable persistent message delivery, we avoid delete-on-read logging; instead, PillarBox keeps messages in the buffer until they are overwritten, and transmits them redundantly. To protect against inference of alerting activity by observing the *length* of message transmissions (as well as against truncation attacks), we avoid append-only logging; instead, PillarBox uses a fixed-capacity buffer. And to protect against inference of alerting activity by observing the *contents* of message transmissions, we employ two layers of authenticated encryption: In additional to the "low-layer" encryption that protects individual buffered messages after compromise, before any buffer transmission PillarBox applies a "high-layer" encryption to the entire content of the transmitted buffer. This completely conceals alerting activity: An adversary cannot observe the evolution of message ciphertexts as they are themselves freshly encrypted from buffer to buffer.

4.4 Detailed construction

Our construction employs a *forward-secure pseudorandom number generator* FS-PRNG (see, e.g., [24]) that exports two operations GenKey and Next to compute the next pseudorandom numbers, as well as an *authenticated encryption scheme* (see, e.g., [16]) that exports operations AEKeyGen, AuthEnc and AuthDec to encrypt messages m of size k to ciphertexts of size $g(k) \geq k$. We, here, assume basic familiarity with these primitives; formal definitions and posible instantiations of these primitives are presented Appendix C.

Data structure. The sender $\mathcal{S}$ maintains the following data structure:

1. a *secret key* σ (also kept by the receiver $\mathcal{R}$);
2. a *buffer* C, $C = (C[0], C[1], \ldots, C[T-1])$, initially filled with *random data*, that takes the form of an array of size $T+1$, where $C[i]$, $0 \leq i \leq T$, denotes the ith position in C; we set the size of each slot $C[i]$ to be $s = g(\ell)$, where ℓ is an appropriate given message length (defining message space $\{0,1\}^{\ell}$).[10]
3. a *current index* I, initialized at a *random* position in C, and itself stored at (special extra) slot $C[T]$.

Input: security parameter κ
Output: initial evolving secret keys $(\sigma_{w,0}, \sigma_{r,0})$, public key pk

```
1. $\sigma_{AE} \leftarrow \mathsf{AEKeyGen}(1^{\kappa})$
2. $s_0 \leftarrow \mathsf{GenKey}(1^{\kappa})$, $s'_0 \leftarrow \mathsf{GenKey}(1^{\kappa})$
3. $(r_0, s_1) \leftarrow \mathsf{Next}(s_0, 1)$, $(r'_0, s'_1) \leftarrow \mathsf{Next}(s'_0, 1)$
4. $\sigma_{w,0} = \sigma_{r,0} = (r_0, s_1, r'_0, s'_1, 0, 0)$, $pk = \emptyset$
5. return $((\sigma_{w,0}, \sigma_{r,0}), pk)$
```

Figure 5: Operation KGen.

Input: secret key (r_i, r'_j, i, j), step t, control b
Output: new secret key $(r_{i+t}, r'_j, i+t, j)$ or $(r_i, r'_{j+t}, i, j+t)$

```
1. if $t \leq 0$ then return $(r_i, r'_j, i, j)$.
2. if $b = $ low then $r_{i+t} \leftarrow \mathsf{Next}(r_i, t)$
   else $r'_{j+t} \leftarrow \mathsf{Next}(r'_j, t)$
3. delete $r_i$
4. if $b = $ high then return $(r_{i+t}, r'_j, i+t, j)$
   else return $(r_i, r'_{j+t}, i, j+t)$
```

Figure 6: Operation KEvolve.

Key management. Key generation and evolution operate as follows. Given a security parameter κ, algorithm KGen first initiates an authenticated encryption scheme as well as two FS-PRNGs, one *low-layer* to generate sequence $r_0, r_1, \ldots$ (for message encryption) and one *high-layer* to generate sequence $r'_0, r'_1, \ldots$ (for buffer encryption). It then initializes the secret states of $\mathcal{S}$ and $\mathcal{R}$, which take the (simplified) form (r_i, r'_j, i, j), denoting the *most recent forward-secure* pseudorandom numbers for the low and high layers,

[10]For instance, in our EAX encryption implementation: $\ell = 1004$ and $g(\ell) = 1024$.

along with their sequence numbers—see the algorithm in Figure 5. Also, given the current secret state (r_i, r'_j, i, j), an integer t and a control string $b \in \{\texttt{low}, \texttt{high}\}$, algorithm KEvolve creates the corresponding low- or high-layer t-th next forward-secure pseudorandom number—see the algorithm in Figure 6.

```
Input: secret key $(r_i, r'_j, i, j)$, message $m \in \{0,1\}^\ell$, buffer $C$
Output: new secret key $(r_{i+1}, r'_j, i+1, j)$, updated buffer $C$
  1. $c \leftarrow \mathsf{AuthEnc}_{r_i}(m)$
  2. $C[C[T]] = (c, i)$
  3. $C[T] = C[T] + 1 \bmod T$
  4. $(r_{i+1}, r'_j, i+1, j) \leftarrow \mathsf{KEvolve}(r_i, r'_j, i, j, 1, \texttt{low})$
  5. delete $r_i$
  6. return $[(r_{i+1}, r'_j, i+1, j), C]$
```

Figure 7: Operation Write.

```
Input: secret key $(r_i, r'_j, i, j)$, buffer $C$
Output: new secret key $(r_i, r'_{j+1}, i, j+1)$, encapsulated buffer $\hat{C}$
  1. $c' \leftarrow \mathsf{AuthEnc}_{r'_j}(C)$
  2. $\hat{C} = (c', j)$
  3. $(r_i, r'_{j+1}, i, j+1) \leftarrow \mathsf{KEvolve}(r_i, r'_j, i, j, 1, \texttt{high})$
  4. delete $r'_j$
  5. return $[(r_i, r'_{j+1}, i, j+1), \hat{C}]$
```

Figure 8: Operation Wrap.

Write, transmit and read operations. Then our main protocols operate as follows. First, given secret writing key (r_i, r'_j, i, j), message m and buffer C, Write securely encodes m, adds it in C and updates the secret key—see the algorithm in Figure 7. Then, given secret writing key (r_i, r'_j, i, j) and a buffer C, Wrap securely encapsulates C to $\hat{C}$ and updates the secret key—see the algorithm in Figure 8. Finally, given secret reading key (r_i, r'_j, i, j) and an encapsulated buffer $\hat{C} = (c', j')$, Read decrypts the buffer and all of its contents returning a set of T messages and updates the secret key—see the algorithm in Figure 9.

A security analysis of our "lockbox" construction appears in Appendix D.

Possible modifications. In certain implementation settings, the following simple modifications may lead to better efficiency. First, instead of applying high-layer encryption on the contents of the entire buffer, such encryption may alternatively be performed in a per-message manner (similar to low-layer encryption but for all messages in the buffer). Moreover, instead of having the two FS-PRNG sequences evolve asynchronously, where the secret keys r_i for low-layer encryption are computed independently of the keys r'_j for high-layer encryption, a combined, synchronous key generation process may be alternatively employed, where the two sequences r_i and r'_i evolve simultaneously using only one FS-PRNG, even if some of the produced keys are never used.

For simplicity, we here consider a fixed-size PBB that holds fixed-size messages (parameters T and ℓ respectively). We finish the section by explaining how PillarBox can be extended to handle variable-length messages and to dynamically enlarge the PBB buffer, as needed, in order to prevent loss of alert messages (due to overwriting) during prolonged PBB-transmission failures.

```
Input: secret key $(r_i, r'_j, i, j)$, encapsulated buffer $\hat{C}$
Output: new secret key $(r_l, r'_{j'}, l, j')$, $(m_0, \ldots, m_{T-1})$
  1. if $j' \leq j$ then return $[(r_i, r'_j, i, j), \perp]$
  2. $(r_i, r'_{j'}, i, j') \leftarrow \mathsf{KEvolve}(r_i, r'_j, i, j, j' - j, \texttt{high})$
  3. $C \leftarrow \mathsf{AuthDec}_{r'_{j'}}(c')$
  4. if $C = \perp$ then return $[(r_i, r'_j, i, j), \perp]$
  5. (View $C$ as an array $C[0], C[1], \ldots, C[T-1], C[T]$)
     $I = C[T]$; $x = 0$
  6. for $0 \leq k < T$ do
       (a) $(c, l) = C[k + I \mod T]$
       (b) if $k = 0 \wedge l \leq i$ then return $[(r_i, r'_j, i, j), \perp]$
       (c) if $k \neq 0 \wedge l \neq LAST + 1$ then return $[(r_i, r'_j, i, j), \perp]$
       (d) $(r_l, r'_{j'}, l, j') \leftarrow \mathsf{KEvolve}(r_i, r'_{j'}, i, j', l - i, \texttt{low})$
       (e) $m_k \leftarrow \mathsf{AuthDec}_{r_l}(c)$; $LAST = l$
  7. return $[(r_{l-T+1}, r'_{j'}, l - T + 1, j'), (m_0, \ldots, m_{T-1})]$
```

Figure 9: Operation Read.

Handling variable-length messages. Without loss of generality, we have considered messages of fixed size ℓ. PillarBox can easily relay messages of variable length using the following simple extension: A variable-length message m (of size ℓ') is padded (in a self-delimiting way) to size k such that $s|g(k)$ and the resulting ciphertext occupies $g(k)/s$ consecutive buffer slots—it is straightforward to extend the algorithms in Figures 7 and 9 accordingly.

Dynamically sizing the PBB. A fixed-size PBB, i.e., parameter T, is necessary to achieve stealth in PillarBox, as noted above. To ensure moderate communication and server-side processing costs, it is helpful to make T as small as possible for the system choice of inter-transmission time or heartbeat interval μ. During prolonged buffer transmission failures, however, even benign ones, a small buffer size T can result in the loss, i.e., overwriting, of alert messages. This problem cannot be solved by enlarging the PBB on demand as new alerts are generated, because T would then correlate with alerting history, and stealth would be lost. However, for the (common) case where the host-side PillarBox module can detect a loss of its network connection with the PillarBox server, it is possible to keep T small, yet preserve alert messages during periods of disconnection.

The idea is to have PillarBox perform *dynamic* enlargement of the PBB: T grows as a function of the time elapsed since the last successful connection.[11] For example, after every hour of lost network connection, PillarBox may add S extra slots to the PBB on the host: A special message m^* (encoding that the buffer adaptation policy is activated) is written in the PBB, to securely bind old and new slots and protect against truncation attacks, and new buffer slots are inserted at the position of the current index I, to ensure new slots are used before old ones are overwritten. When the PillarBox connection is reestablished, the PBB size may again be reduced to size T. (The most recently written T slots, behind pointer I, are retained.) With this approach, T is a function of connection history, *not* alerting history, and stealth is preserved. At the

[11] Note that detecting network connection failures does not require PillarBox server message acknowledgments. For instance, an enterprise device can simply observe whether it is connected to the enterprise's network.

same time, an adversary cannot simulate benign host disconnection and then overwrite alerts: The PillarBox server can record disconnection periods and adjust its expectation of PBB transmission sizes from the host accordingly.

5 Prototype implementation

We implemented a prototype of PillarBox in C++. As shown in Figure 3, PillarBox includes several components, both on the host and on the server. Host-based components depend on an alerter for detection of events indicating a compromise, but PillarBox is designed to be agnostic to the particular choice of alerter. In our implementation we instantiate the alerter with syslog.

The PBB data structure, described in Section 4, is implemented as a class which exports two methods, Write and Wrap. When the host-based application is started it creates a new buffer, which remains in memory, and interacts with the bufferer and transmitter. Its size is specified at creation time, and its starting position is randomly selected.

After creating the buffer, the host application launches two threads, one implementing the bufferer, the other implementing the transmitter. For independence from the alerter, the bufferer reads from a named pipe to which the alerter writes. Anytime the alerter writes to the pipe, it is received by the bufferer and then encrypted before being added to the buffer as described in the algorithm in Figure 7. The transmitter uses the Wrap functionality of the algorithm in Figure 8 to re-encrypt a copy of the buffer before sending it over the network to the server.

To implement the authenticated encryption necessary in our system we utilize an open-source version of EAX-mode encryption. We developed a custom forward-secure pseudorandom number (FS-PRN) generator that we use to generate the necessary encryption keys. We generate keys for both the message encryption before addition to the buffer, as well as the re-encryption of the buffer before transmission.

The server-based application of PillarBox implements three primary functions described in Figure 3. The receiver simply listens for incoming connections from the transmitter, passing them along to the decrypter as they arrive. The decrypter then decrypts the new messages using the Read method described in the algorithm of Figure 9. Once complete the messages are passed off to the gap-checker who raises an alert in the event that the current set of messages does not overlap the previous messages. The gap-checker also uses a second thread which wakes on regular intervals to ensure delivery of heartbeat messages. If, when the thread wakes, no messages have been received in the specified window, an alert is raised.

6 Experimental evaluation

We next experimentally validate the effectiveness of PillarBox in securing alerts during the critical window. We first demonstrate the merits of our alert-buffering approach via a generic attack against alert-pushing methods. We then show that PillarBox is fast enough to win the race condition against an attacker trying to disrupt the securing of alert messages. Surprisingly, even when an attacker already has the privilege necessary to kill PillarBox, the execution of the kill command itself can be secured in the PillarBox buffer before the application dies. Finally, we validate the feasibility of PillarBox as a practical alert-relaying tool.

6.1 Demonstrating direct-send vulnerability

We motivate the need for securing the chain of custody in SASs and justify our design choice of host-side buffering, rather than immediately putting alerts on the wire, by showing the feasibility of an attacker intercepting on-the-wire host alert transmissions silently (without sender or receiver detection) in a rather simple setting.

Using the Ettercap tool [9] we inserted an attack machine (attacker) as a man-in-the-middle between our client and server communicating over a switch. The attacker performed ARP spoofing against the switch, to which most non-military-grade hubs and switches are vulnerable. Because it attacked the switch, neither endpoint observed the attack. Once inserted between the two machines, our attacker was able to drop or rewrite undesired packets on the fly. Even if the client and server had been communicating over a secured channel (a rarity in current practice), alert messages could still easily have been dropped, preventing any indication of the attack from reaching the server.

If executed within a subnet, the attack described here would rarely be detected, even by a forensic tool performing network packet capture, as these tools are typically deployed to monitor only inbound/outbound traffic, or at best across subnets.

Given the ease with which we were able to not only prevent communication between a client and server, but moreover modify what the server received, without detection, it should be clear just how important chain-of-custody is in SASs. If the messages being transmitted are of any value, then they need to be protected. Otherwise an attacker can simply block or modify all SAS communication while attacking a host, after which he can turn off or otherwise modify what the SAS sends from the client side. Attacking in such a way makes it impossible for the server to detect anything has gone wrong and motivates our desire to provide a better way to secure log messages.

6.2 PillarBox vs. direct alert pushing

In addition to combatting the network-based attack demonstrated above, PillarBox can also provide integrity protection more quickly than current direct-send approaches where the transmission must reach the server before it is protected. Essentially we show that authenticated encryption and storage is faster than any network transmission. We avoid the overhead of socket connections by using UDP and measure only the time necessary to send a packet, which we then compare to the time needed to encrypt the same information and store it locally in the PBB. Distributions over 100 tests are presented in Figure 10.

As our experiments show, PillarBox achieves a clear advantage in the speed with which it secures alerts. These alerts must still be transmitted to the server, but PillarBox protects the integrity of the alert contents faster than most current on-the-wire pushing systems.

6.3 Race-condition experiments

We now show that it is feasible for a SAS combined with PillarBox to detect an attack in progress and secure an alert before an attacker can disrupt PillarBox operation (i.e., that the critical window is non-zero in size). PillarBox depends on both an alerter (in our case, syslog), and a named pipe used to communicate from the alerter to the bufferer. Both of these components, as well as PillarBox itself, can be attacked, creating a race condition with the attacker. If any of the components can be shut down fast enough during an attack, alerts may not be secured in the PBB. Surprisingly, we show that even an attacker with the necessary (root) privilege rarely wins this race ($\approx 1\%$ of the time).

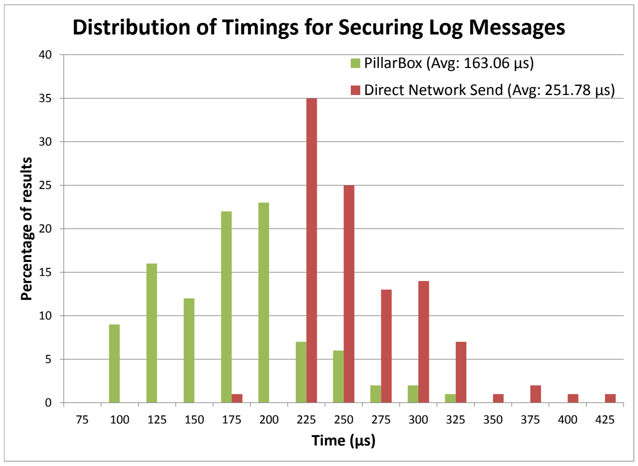

Figure 10: Distribution of timings for PillarBox cryptographically securing log messages locally vs. direct network transmission.

To demonstrate PillarBox winning the race condition, we now explore how quickly each of the necessary components can be shut down relative to the speed with which such events themselves can be logged. To bias our experiments in favor of an attacker, we assume the attacker has gained access to a privileged account that already has the necessary permissions to kill any of the components. We record time required for the attacker to issue a *single command* to kill the process and show that the command itself gets secured by PillarBox before the targeted component is terminated. Our tests were performed on an 2.5GHz Intel Core 2 Duo T9300 processor with 4 GB of memory and Ubuntu 12.04 as the operating system.

Killing PillarBox. PillarBox is a simple application that is easily terminated by an attacker, although it can be run as root to provide some protection. To be secured, alerts must be generated, routed by syslog to the named pipe, and then picked up by PillarBox, encrypted and added to the buffer. An attacker's best bet at disrupting the securing of alerts is to try and shutdown PillarBox itself. If run as root, PillarBox can be terminated by invoking root privilege and issuing a kill command.[12] Calling kill with the -9 signal immediately terminates any program, unless it's in the process of making a system call; it then terminates when the system call returns. Using sudo runs the command as root, but also generates an authentication alert message which syslog picks up. The full one-line command sudo kill -9 $<$ PillarBox_pid $>$ immediately terminates PillarBox, but usually not before a log event is created, routed by syslog through the named pipe, and secured.

As Table 1 shows, in the majority of runs the alert message is locked away in $\approx$ 4ms. Unlike our previous experiments, these timings include alert generation, routing by syslog through the named pipe, and securing by PillarBox. As Figure 10 shows, PillarBox accounts for a minuscule fraction of the total time observed. Alert messages are, on average, secured in PillarBox before it is killed with almost 3 ms. to spare. However, in about 1% of our experiments, PillarBox was killed before receiving the alert message and encrypting it. All of the commands in Table 1 were run 100 times with averages and standard deviations shown.

[12] kill or pkill could be used to terminate the process: pkill takes in the process name, while kill takes a process id; otherwise they operate the same.

Table 1: Average time from the start of a command until log is secured in PillarBox and total time for command completion.

	Secured	Std. Dev.	Disrupted	Std. Dev.	Command
Syslog (typical)	4.09ms	0.30ms	8.86ms	2.43ms	sudo kill -9 $<$ syslog_pid $>$
Syslog (worst)[14]	32.33ms	5.38ms	9.32ms	2.81ms	sudo kill -9 $<$ syslog_pid $>$
Named pipe	6.36ms	3.02ms	8.99ms	3.30ms	sudo rm named_pipe
PillarBox	4.01ms	0.19ms	6.95ms	0.37ms	sudo kill -9 $<$ PillarBox_pid $>$

Impacts of system load. To further test the ability of the attacker to beat PillarBox, we also ran tests under varying amounts of disk, memory, and CPU load. Disk load appeared to have little to no effect on either the success of PillarBox, or the timing measurements. As expected, load on the system memory slowed everything down—lengthening both the time to secure, but also the time until the kill completes—but did not appear to impact the success of PillarBox winning the race condition. For unexplained reasons, CPU load did seem to impact PillarBox on our test machine. Oddly, PillarBox did well (0% failure) at near 100% load, but relatively poorly ($< 4\%$ failure) at 20% load. These tests were run 1000 times to further reduce noise. Additionally, we re-ran our tests on a 2.13 GHz Intel Xeon E5506 Quad Core processor with 3GB of RAM running Red Hat Enterprise Linux WS v5.3 x86_64. On that machine we again noticed $\approx 1\%$ of tests failing, but did not find a correlation between load and failure rate. We expect CPU scheduling to be at fault but leave a more thorough investigation of the effects of load as well as the impact of virtualization or greater numbers of cores as future work. If CPU scheduling is indeed the cause, running PillarBox with higher priority should further lower the probability of an attacker winning the race condition.

Killing the named pipe. We also considered attacks against the other components (syslog and the named pipe). We use a named pipe to pass alerts from syslog to PillarBox. A named pipe is a permanent pipe created in the filesystem which can be read from and written to by any process. To destroy a named pipe created by root an attacker would need to run sudo rm named_pipe. Again, the invocation of sudo (or otherwise transitioning to root privilege) generates a log event. As Table 1 shows, the log messages created pass through the pipe before it is closed. There were no failures in these tests.

Killing syslog. The alerter (syslog) is the first to handle the log message, and can be shutdown or killed by running sudo kill -9 $<$ syslog_pid $>$.[13] Table 1 shows that the log message is sent by syslog before it is killed. However, presumably due to process scheduling, in several runs the kill command returns before the alert message is secured in the PBB. Because the message *always* arrives in the PBB (again, there were no failures), we assume these represent runs where the alert is passed to the named pipe before syslog terminates and then read from the pipe when the PillarBox process is later scheduled by the OS.[14] This issue is diminished in the tests against the named pipe and PillarBox, explaining their perceived lower average timings (and standard deviations).

Vanishing rules. When PillarBox provides stealth, it is best combined with vanishing SAS rules to prevent critical information leakage. Recall that if an attacker cannot prevent PillarBox from securing events in the critical window, the attacker benefits from at least learning how the system is instrumented and what alerts were likely to have been generated. In our test setup, the vanishing alerts generate an alert whenever

[13]The alerter could be more integrated into the kernel itself, making it even harder to intercept and/or kill. In our case, syslog channels log messages generated by the kernel and does not actually generate them itself.

[14]Due to presumed OS scheduling interruptions, in about 1/3 of the runs the kill command returns before the message is *successfully* secured in PillarBox. The "Syslog (worst)" results show the timings observed in those cases.

Table 2: Timeline of events related to the execution of the attacker's command `sudo cp /etc/rsyslog.d/vanish.conf /home/vanish.copy`.

Event	Start	Message Secured	Rule Deleted	Copy Fails
Avg. Time (ms)	0.00	4.00ms	4.04ms	7.21ms
Std. Dev.	N/A	0.44ms	0.44ms	0.81ms

a root user logs in. To test the race condition, we instrumented PillarBox to delete the vanishing alerts configuration file after securing the alert message. The attacker attempts to create a copy of the sensitive alerter configuration file. As it is shown by the relative timing of events over 100 test runs in Table 2, after securing the alert message, PillarBox always successfully deletes the configuration file at least 2.72 ms. before the attempted copy.

Privilege escalation. Having shown that PillarBox can win the race conditions related to securing alerts and causing them to vanish, even in the pessimistic case where the attacker *starts with the necessary permissions*, we now consider the issue of privilege escalation. The concern is that if the attacker exploits vulnerabilities the transition to root privilege may not get logged. We assume that most privilege escalations could be detected given the proper instrumentation and that disrupting any of the necessary components in our system (e.g. corrupting its memory address space) without root privilege is infeasible given current architectures (e.g., Address Space Randomization [37], etc.).

As an example of a common privilege escalation, we consider the "Full Nelson" attack, which exploits CVE-2010-4258, CVE-2010-3849, and CVE-2010-3850 to gain root access. We find that this attack generates kernel messages that syslog can pick up and pass through the named pipe and into the PBB before the exploit completes and the attacker terminates essential SAS or PillarBox components or reads the configuration file. In fact, the attack includes a necessary sleep command that further benefits timely securing of alerts in PillarBox. Even in the most pessimistic case, in which the exploit code uses the kill system call before ever launching a shell, and the sleep command is removed (causing the exploit to fail), the log messages are *still* locked away in PBB before the exploit program tries to disrupt PillarBox. Since the system must be restored after the privilege escalation, we were not able to run 100 instances, but we repeatedly demonstrated that the kernel log messages can be secured in PBB before being killed.

While the "Full Nelson" attack is representative of other local privilege escalation attacks, this by no means guarantees that faster or quieter privilege escalations don't exist. What it does demonstrate is that the event signaling the end of the critical window (the elevation of privilege giving the attacker full control) can itself often be detected and secured in PillarBox before such privilege enables disruption of the PillarBox tool.

Asynchronous logging. We have also tested PillarBox in a more asynchronous setting using Snort as our alert generator to detect a remote SSH exploit. Once the attacker has shell access it escalates privilege and then shuts down PillarBox. Table 3 shows that in the average case all of the defenders actions (detecting the attack, securing the log in PillarBox and deleting the stealthy rule) complete a whole second before the attacker even gains shell access. The high standard deviation (measured over 100 runs) indicates that the attacker may be able to learn detection rules on rare occasion (this was never observed in our experiments), but cannot prevent the securing of the log messages.

Table 3: Race condition timings (measured from the launch of the attack).

Defender Event	Attack detected	Log secured	Rule deleted
Average	1,645.441ms	1,645.609ms	1,645.772ms
Std. Dev.	1,069.843ms	1,069.842ms	1,069.840ms

Attacker Event	Remote shell	Privilege escalation	Rules copied	Log file deleted
Average	2,692.536ms	2,693.474ms	2,696.524ms	2,696.590ms
Std. Dev.	1,324.419ms	1,324.432ms	1,324.919ms	1,324.990ms

6.4 Observed alerting frequencies

We performed an analysis of a large enterprise (>50,000 users) dataset across a period of 7 hours. This dataset contains all collectable logs from this network, including servers, laptops, network devices, security appliances, and many more. The goal was to derive information about the typical alert frequency across a representative work day.

It is critical to note that only certain messages pertaining to, e.g., indicators of compromise, will be selected for inclusion in the PillarBox protected queue. As such, the data found here represents an overloaded maximum: It is unlikely that most networks will generate such volumes of alerts, and most alerts will not be applicable to PillarBox.

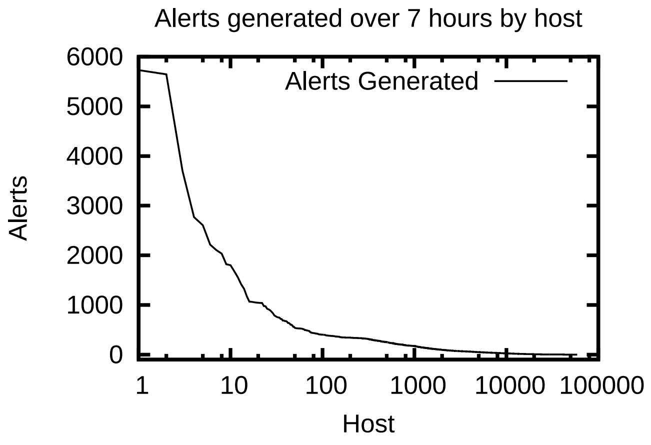

Figure 11: Host-generated alerts over 7h.

Figure 11 shows the distribution of alerts coming from hosts within the enterprise. The x-axis is in log scale, showing that the majority of machines send very few alert messages, while a small subset send the majority. Over a 7-hour window, the busiest machine generated 8603 alerts, but the average across all machines (59,034 in total) was only 18.3 alerts. Clearly, therefore, if we design the system to handle a throughput of one alert per second (3600 alerts an hour) our system will be able to handle even the busiest of alerters. The maximum observed rate in our dataset was 1707 alerts / hour.

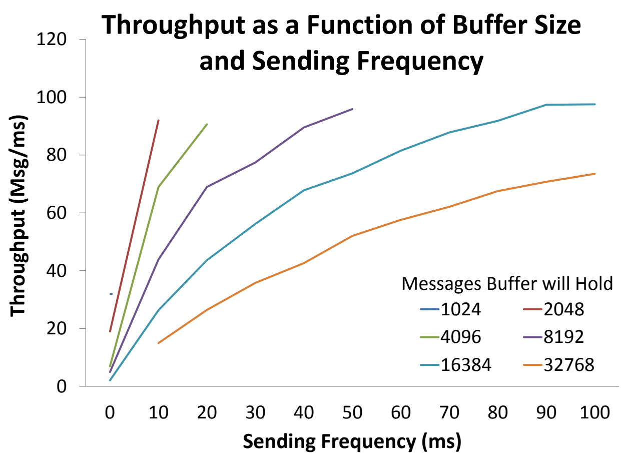

Figure 12: Throughput vs. $|\mathrm{PBB}|$, μ.

6.5 Throughput experiments

We now show that PillarBox can process events at a practical rate. Given a constant stream of events, the host-based application was able to process nearly 100,000 messages per second, higher than any rate recorded in our dataset. The speed with which PillarBox can encode messages naturally depends on a number of factors, e.g., message size, the cost of computing FS-PRNGs, PBB's size, and the frequency with which the buffer is re-encrypted and sent.

Obviously the larger the messages, the longer they take to encrypt. In our study, we used the standard log messages generated on our Linux system, typically a few hundred characters long. We also used a flat hash chain for our FS-PRNG, which only requires one computation per number, minimizing key-generation overhead.

Figure 12 explores tradeoffs between buffer size and send frequency in terms of their impact on maximum throughput. Some combinations of buffer size and send rate led to buffer overflows, and were removed. Performance seems to increase as buffer size increases and send frequency decreases, as expected. A large buffer that is rarely re-encrypted for sending can process events more quickly that a small, frequently sent buffer. As Figure 12 shows, throughput seems to top out just shy of 100 messages / ms, further evidence of the minimal overhead of PillarBox.

7 Related work

PillarBox uses host-side buffering to secure alerts for transmission to a remote server. An alternative is a trusted receiver within a protected environment on the host itself. Trusted hardware like a TPM can be leveraged to help secure alerts locally [18, 40]. Alternatively, a *hypervisor* (virtual machine monitor (VMM)), has higher privilege than a guest OS, isolating it from OS-level exploits. Thus PillarBox messages could be sent from a SAS to a same-host hypervisor. Hypervisor-based messaging can be blended with even stronger security functionality in which the hypervisor protects a SAS (or other monitoring software) itself against corruption as in, e.g., [38], and/or is itself protected by trusted hardware, as in Terra [21]. Using trusted hardware and/or hypervisor-based approaches, where available, offer an excellent alternative or complement to PillarBox.

Hypervisor-based approaches, however, have several notable limitations. Many hosts and devices today aren't virtualized and some, e.g., embedded devices, probably won't be for a long time. Operating constraints often limit security administrator access to hypervisors. For instance, IT administrators may be able to require that personal devices in the workplace (e.g., laptops, tablets, and smartphones) contain an enterprise-specific VM or application, but they are unlikely to obtain full privileges on such devices. Finally, hypervisors themselves are vulnerable to compromise: Some works have noted that the code sizes, privilege levels, and OS-independence of modern VMMs belie common assertions of superior security over traditional OSes [25, 42].

PillarBox builds in part on funkspiel schemes, introduced by Håstad et al. [23]. A funkspiel scheme creates a special host-to-server channel whose existence may be known to an adversary; but an adversary cannot tell if or when the channel has been used, a property similar to stealth in PillarBox. (By implication, an adversary cannot recover message information from the channel either.) As in our work, a funkspiel scheme resists adversaries that see all traffic on the channel and ultimately corrupt the sender.

Funkspiel schemes, though, are designed for a specific use case: Authentication tokens. The transmitter either uses its initialized authentication key or swaps in a new, random one to indicate an alert condition. A funkspiel scheme thus transmits only a single, one-bit message ("swap" or "no swap"), and is not practical for the arbitrarily long messages on high-bandwidth channels in PillarBox.

Another closely related technique is forward-secure logging (also called tamper-evident logging), which protects the integrity of log messages on a host after compromise by an adversary (see, e.g., [17, 19, 27, 29, 31, 35, 36, 41, 43, 44]). As already discussed, while these systems use forward-secure integrity protection like PillarBox, they are not designed for self-protecting settings like PillarBox. They aim instead for forensic protection, e.g., to protect against retroactive log modification by an administrator. Some schemes, e.g., [17, 27, 35, 36], are designed to "close" a log, i.e., create forward security for new events, only periodically, not continuously. Additionally, existing forward-secure logging systems do not aim, like PillarBox, to achieve stealth.

Finally, in a different context than ours, the Adeona system [34] uses forward-secure host-side buffering in order to achieve privacy-preserving location tracking of lost or stolen devices. Adeona uses cryptographic techniques much like those in PillarBox to cache and periodically upload location information to a peer-to-peer network. Adeona does not offer integrity protection like that in PillarBox, however, nor does it address the complications of high throughput, buffer wraparound, and transmission failures in our setting.

8 Conclusion

Today's big data security analytics systems rely on untrustworthy data: They collect and analyze messages from Security Analytics Sources (SASs) with inadequate integrity protection and are vulnerable to adversarial corruption. By compromising a host and its SAS, a strong attacker can suppress key SAS messages and alerts. An attacker can also gather intelligence about sensitive SAS instrumentation and actions (potentially even just via traffic analysis).

We have introduced PillarBox, a new tool that provides key, missing protections for security analytics systems by securing the messages generated by SASs. Using the approach of host-side buffering, PillarBox provides the two properties of *integrity* and *stealth*. PillarBox achieves integrity protection on alert messages even in the worst case: Hostile, self-protecting environments where a host records alerts about an attack in progress while an attacker tries to suppress them. Stealth, an optional property in PillarBox, ensures that at rest or in transit, a SAS message is invisible to even a strong adversary with network and eventually host control.

Our experiments with PillarBox validate its practicality and protective value. We show, e.g., that PillarBox can "win the race" against an adversary mounting a local privilege escalation attack and disabling PillarBox as fast as possible: PillarBox secures alert messages about the attack before the attacker can intervene. Our study of alerting rates in a large (50,000+ host) environment and of local host performance confirms the low overhead and real-world deployability of PillarBox. We posit that PillarBox can offer practical, strong protection for many big data security analytics systems in a world of ever bigger data and more sophisticated adversaries.

Acknowledgments. We thank Alina Oprea, Ting-Fang Yen and Todd S. Leetham for many useful discussions during the design and evaluation of PillarBox.

References

[1] Android.Bmaster. `http://www.symantec.com/security_response/writeup.jsp?docid=2012-020609-3003-99&tabid=2`.

[2] Aurora attack (source 1). `http://www.mcafee.com/us/threat-center/aurora-enterprise.aspx`.

[3] Aurora attack (source 2). `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0249`.

[4] Cve-2007-1531. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-1531`.

[5] Cve-2010-2772. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-2772`.

[6] Cve-2010-2979. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-2979`.

[7] Cve-2010-3888. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3888`.

[8] Duqu. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3402`.

[9] Ettercap. `http://ettercap.sourceforge.net/`.

[10] Infostealer.Shiz. `http://www.symantec.com/security_response/writeup.jsp?docid=2011-121202-4242-99&tabid=2`.

[11] RSA breach (source 1). `http://bits.blogs.nytimes.com/2011/04/02/the-rsa-hack-how-they-did-it/`.

[12] RSA breach (source 2). `http://blogs.rsa.com/rivner/anatomy-of-an-attack/`.

[13] Stuxnet. `http://www.symantec.com/connect/blogs/w32stuxnet-dossier`.

[14] ZeroAccess (source 1). `http://www.mcafee.com/threat-intelligence/malware/default.aspx?id=562354`.

[15] ZeroAccess, (source 2). `http://www.symantec.com/security_response/writeup.jsp?docid=2011-071314-0410-99&inid=us_sr_carousel_panel6_threat_spotlight`.

[16] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *J. Cryptol.*, 21:469–491, September 2008.

[17] Mihir Bellare and Bennet Yee. Forward-security in private-key cryptography. In *CT-RSA'03: Proceedings of the 2003 RSA conference on The cryptographers' track*, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag.

[18] CheunNgen Chong, Zhonghong Peng, and PieterH. Hartel. Secure audit logging with tamper-resistant hardware. In Dimitris Gritzalis, Sabrina De Capitani di Vimercati, Pierangela Samarati, and Sokratis Katsikas, editors, *Security and Privacy in the Age of Uncertainty*, volume 122 of *IFIP The International Federation for Information Processing*, pages 73–84. Springer US, 2003.

[19] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security*, pages 317–334, 2009.

[20] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[21] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 193–206, New York, NY, USA, 2003. ACM.

[22] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[23] Johan Håstad, Jakob Jonsson, Ari Juels, and Moti Yung. Funkspiel schemes: an alternative to conventional tamper resistance. In *CCS '00: Proceedings of the 7th ACM conference on Computer and communications security*, pages 125–133, New York, NY, USA, 2000. ACM.

[24] G. Itkis. *Handbook of Information Security*, chapter Forward Security: Adaptive Cryptography—Time Evolution. John Wiley and Sons, 2006.

[25] Paul A. Karger. Securing virtual machine monitors: what is needed? In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 1:1–1:2, New York, NY, USA, 2009. ACM.

[26] J. Kelsey, J. Callas, and A. Clemm. Rfc 5848: Signed syslog messages. http://tools.ietf.org/html/rfc5848, 2010.

[27] John Kelsey and Bruce Schneier. Minimizing bandwidth for remote access to cryptographically protected audit logs. In *Recent Advances in Intrusion Detection*, pages 9–9, 1999.

[28] Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In *ACM Conference on Computer and Communications Security*, pages 108–115, 2000.

[29] Di Ma and Gene Tsudik. A new approach to secure logging. *Trans. Storage*, 5:2:1–2:21, March 2009.

[30] Mandiant. M-trends: The advanced persistent threat. `www.mandiant.com`, 2010.

[31] Giorgia Azzurra Marson and Bertram Poettering. Practical secure logging: Seekable sequential key generators. In *Proceedings of 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings (ESORICS)*, volume 8134 of *Lecture Notes in Computer Science*, pages 111–128. Springer, 2013.

[32] A. Okmianski. Rfc 5426: Transmission of syslog messages over UDP. http://tools.ietf.org/html/rfc5848, 2009.

[33] J. Oltsik. Defining big data security analytics. *Networkworld*, 1 April 2013.

[34] Thomas Ristenpart, Gabriel Maganis, Arvind Krishnamurthy, and Tadayoshi Kohno. Privacy-preserving location tracking of lost or stolen devices: Cryptographic techniques and replacing trusted third parties with DHTs. In *Proceedings of the 17th USENIX Security Symposium*, pages 275–290. USENIX Association, 2008.

[35] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, pages 4–4, Berkeley, CA, USA, 1998. USENIX Association.

[36] Bruce Schneier and John Kelsey. Tamperproof audit logs as a forensics tool for intrusion detection systems. *Comp. Networks and ISDN Systems*, 1999.

[37] H. Shacham, M. Page, B. Pfaff, E.J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*, pages 298–307, 2004.

[38] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 477–487, New York, NY, USA, 2009. ACM.

[39] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. http://eprint.iacr.org/2004/332, 2004.

[40] Arunesh Sinha, Limin Jia, Paul England, and JacobR. Lorch. Continuous tamper-proof logging using tpm 2.0. In Thorsten Holz and Sotiris Ioannidis, editors, *Trust and Trustworthy Computing*, volume 8564 of *Lecture Notes in Computer Science*, pages 19–36. Springer International Publishing, 2014.

[41] Brent R. Waters, Dirk Balfanz, Glenn Durfee, and Diana K. Smetters. Building an encrypted and searchable audit log. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*. The Internet Society, 2004.

[42] Y. Chen Y. Chen, V. Paxson, , and R. Katz. What's new about cloud computing security? technical report what's new about cloud computing security? Technical Report UCB/EECS-2010-5, UC Berkeley, 2010.

[43] Attila Altay Yavuz and Peng Ning. Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, pages 219–228. IEEE Computer Society, 2009.

[44] Attila Altay Yavuz, Peng Ning, and Mike Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In *Proceedings of 2012 Financial Cryptography and Data Security (FC 2012)*, 2012. To appear.

A Ideal-model lockbox description

At the core of its functionality, PillarBox employs a cryptographic construction that implements a reliable message-trasmission channel that, even under adversarial conditions, ensures tamper-evident and stealthy delivery of messages to a remote server. Conceptually, this serves as a special "lockbox" and, thereby, we formally refer to our construction as realizing a *lockbox scheme*.

Here, to gain better understanding of the unique properties of a lockbox scheme, or simply lockbox, we cast its description in an ideal model where its functionality is expressed in a highly abstract setting that captures only necessary operational features. Accordingly, the security properties of a lockbox are defined in relation to some maximally restricted exported interface, and as such, they offer a clear description of which tasks an adversary can legitimately perform using the primitive and which are infeasible to perform. As we discuss later, such an idealized description of a lockbox has an additional benefit, as it helps distinguishing its functionality from that achieved by the weaker notion of *forward-secure logging*.

A.1 Lockbox as a special secure channel

We consider a *sender* $\mathcal{S}$ and a *receiver* $\mathcal{R}$ that communicate through a *channel* C which by design is used to transmit—from $\mathcal{S}$ to $\mathcal{R}$—messages that belong in a fixed, bounded-size language L. We view L as a subset of a universe U of all possible messages that are transmittable through C, where, in particular, $U = L \cup \{\emptyset, \perp\}$ for a special *null* message $\emptyset$ and a special *failure* message $\perp$. Communication through C is *asynchronous*, meaning that messages sent by $\mathcal{S}$ through C do not necessarily reach $\mathcal{R}$ instantaneously, if at all. Instead, we view C as a *fixed-capacity memory buffer* which can be accessed through a special write/read interface and which can simultaneously transmit up to T distinct messages in L. In this view, C consists of T cells or *slots*, each storing one message in U; we denote by $C[i]$ the content of the $(i+1)$th such slot, $0 \leq i \leq T-1$, and also assume the existence of an *index* $I \in \{0, \ldots, T-1\}$ that specifies the *current* slot which is scheduled to store the next new message that will be written in C. (We use the terms "current slot" or "current index" interchangeably.) For convenience, we consider C to be of size $T+1$, i.e., $C = (C[0], C[1], \ldots, C[T])$, where the current index I is stored at the $T+1$th slot $C[T]$. We use the terms channel and buffer interchangeably.

Then, the basic *read/write interface* exported by the channel is through the following two operations:

- **Operation** `write`: Given input message $m \in L$, the sender $\mathcal{S}$ can perform a $\texttt{write}(m)$ operation on C, which adds message m in C by setting $C[I] \leftarrow m$ and advancing $I \leftarrow I + 1 \mod T$. We assume that C is initiated to $C[i] = \emptyset$, $i \in \{0, \ldots, T-1\}$, with I *randomly chosen*, i.e., $I \overset{R}{\leftarrow} \{0, \ldots T-1\}$.

- **Operation** `read`: Analogously, the receiver $\mathcal{R}$ can perform a $\texttt{read}()$ operation, which returns pair (M, I) where M is a *sequence* $(m_1, \ldots m_T)$ of the messages contained in C, where $m_{i+1} = C[i] \in$

$L \cup \{\emptyset, \perp\}$, $i \in \{0, \ldots, T-1\}$ and I is the current index I of C. We say that this `read` operation *produces* messages $(m_1, \ldots m_T)$; we write $(m_1, \ldots m_T) \leftarrow \texttt{read}()$.

We note that with respect to the contents of channel C operation `write` is always *stateful*, as it changes the internal state of C. In contrast, an operation `read` may be *stateless*, i.e., it may not affect the internal state of C. Also, we note that since messages are stored in C in a FIFO manner, after the first T `write` operations the oldest message is evicted to write a new message.

A lockbox corresponds to a specific channel C that is controlled by an adversary: The adversary can get access to the read/write interface of the C in a certain restricted way, and can additionally get access to a special interface through which messages contained in C can be destroyed. More specifically:

Definition 1 (Lockbox scheme - Ideal Model.) *We say that an adversary $\mathcal{A}$* controls channel C*, if the following take place:*

- **Initialization:** *Initially, the channel is set to $C[i] = \emptyset$, $i \in \{0, \ldots, T-1\}$ with $I \overset{R}{\leftarrow} \{0, \ldots, T-1\}$, and the receiver maintains a sequence P of produced messages by* `read` *operations, set to $P = (\emptyset)$.*
- **Phase I:** *The sender $\mathcal{S}$ makes use of the channel C for the transmission of messages in L and the receiver $\mathcal{R}$ asynchronously performs a series of* `read` *operations on C.*
- **Attack:** *At some time t^* of its choice, the adversary $\mathcal{A}$* takes full control of the sender *$\mathcal{S}$ and learns the current index I of C. We call t^* the* time of compromise.
- **Phase II:** *After time t^*, the adversary $\mathcal{A}$ additionally* gets exclusively access *to:*
  1. *regular* `write` *operations, thus being able to insert into C messages from L of its choice; and*
  2. *a special operation* `corrupt` *which, given an index j as input, $0 \leq j \leq T-1$, sets the content of the $(j+1)$th slot of C to the failure message $\perp$, i.e.,*

$$\{C[j] \leftarrow \perp\} \leftarrow \texttt{corrupt}(j) .$$

A lockbox scheme is a channel C that can be controlled by an adversary $\mathcal{A}$ as above but where all `read` *operations are* stateless*, not altering the contents of C. That is, two or more consecutive* `read` *operations without any* `write` *operation in between produce exactly the same sequence of messages.*

A.2 Relation to PillarBox operation

We now discuss the security properties that a lockbox scheme adopts directly from Definition 1. Recall that (during normal operation) the channel C is used by the sender to asynchronously transmit to the receiver messages from an underlying language L. During a specific time window and depending on the rate at which messages are written in and read from C, a given written message may be produced by the receiver zero, one or more times. Considering an adversarial behavior over the channel, we observe that Definition 1 significantly restricts the way an adversary $\mathcal{A}$ can get access to C. In particular:

1. During phase I, $\mathcal{A}$ has no access to the messages transmitted through C.
2. At the point of compromise, $\mathcal{A}$ fully controls the sender and also learns the current index I of C.

3. During phase II, $\mathcal{A}$ additionally has full control over the `write` operations performed by the sender $\mathcal{S}$ and $\mathcal{A}$ may also perform the special `corrupt` operations. That is, from this point on only messages of $\mathcal{A}$'s choice are added in C and $\mathcal{A}$ may even selectively destroy the contents of slots of its choice in C.

We can associate the above three phases in our ideal model to corresponding phases of an attack against the operation of PillarBox as follows. Phase I corresponds to the *pre-compromise* state of the PillarBox system, where the adversary gets *no access or control* over the transmitted messages.[15] The attack phase corresponds to the complete compromise of the sender by the adversary, where, in particular, the internal current index is learned by the adversary. Phase II corresponds to the *post-compromise* state of the PillarBox system, where the adversary gets to choose which messages can be written into the channel or be destroyed.

Then, combined with Definition 1, this restricted access over the channel implies the following important security properties for any (real model implementation of a) lockbox scheme:

1. **Immutability:** A lockbox provides *immutability* of transmitted messages. That is, $\mathcal{A}$ cannot forge the messages that are contained in C at the time of compromise. Indeed, although $\mathcal{A}$ takes full control of $\mathcal{S}$, $\mathcal{A}$ may only make two uses of the channel:

   (a) $\mathcal{A}$ may either write in C *legitimate* messages in L, or

   (b) $\mathcal{A}$ may destroy the delivery of chosen messages in C but in a *detectable* way, since a failure message $\perp$ is substituted (and thus produced by $\mathcal{R}$).

   Consequently, $\mathcal{A}$ *cannot alter any message*: Any produced non-null message at $\mathcal{R}$ corresponds to a *valid* message in the lockbox that was written in C by $\mathcal{S}$ prior the time of compromise or by $\mathcal{A}$ after the time of compromise. Additionally, $\mathcal{A}$ *cannot change the order of the messages in the channel*: Only message deletions are permitted so that $\mathcal{A}$ can only destroy the delivery of chosen messages but cannot alter the order with which they are produced by $\mathcal{R}$.

2. **Stealth:** A lockbox provides *stealth* of transmitted messages. That is, $\mathcal{A}$ can learn *nothing* about the contents of C that is not explicitly added in it by $\mathcal{A}$ herself. Indeed, in $\mathcal{A}$'s view the channel is an opaque black box, and at all times $\mathcal{A}$ has limited access to its contents:

   (a) At pre-compromise state, $\mathcal{A}$ cannot learn anything about the contents of the channel:*at all times, C itself perfectly hides its contents*. Not only can messages in the channel be written such that the adversary never learns them, but also the adversary does not even learn of their existence.

   (b) At the point of compromise, $\mathcal{A}$ cannot learn the messages that are present in the channel or even learn if such messages exist in the channel. $\mathcal{A}$ can only learn the current slot in C, but since this index is initially randomized, it conveys no information about the usage of C thus far.

   (c) At post-compromise state, $\mathcal{A}$ of course completely controls all the messages that are added in the channel, as well as the positions of the messages that $\mathcal{R}$ will fail to produce.

   Overall, $\mathcal{A}$ *may only learn those messages explicitly written in C by itself in the post-compromise state and nothing else*.

[15]In this ideal model, we assume that the receiver privately consumes the received messages, thus leaking no information. In reality, depending on the application and the receiving server's policy some information about the transmitted messages may be leaked through the *publicly observable* actions of the receiver. In general, in a SAS system these actions may depend on the received messages (e.g., these public actions may be taken in response of an actual attack or intrusion). However, this type of leakage is inherent in any real-life SAS system, and is thus outside the scope of our work.

Finally, there is an additional important property provided by a lockbox. The following is rather an algorithmic or structural property but one that may have significant security implications for the higher-level application that employs a lockbox.

3. **Persistence:** A lockbox provides *persistence* of transmitted messages. That is, due to the stateless `read` operations, message transmissions can be configured so that *any message written once in C by $\mathcal{S}$ may be produced more than one once by $\mathcal{R}$.* (For instance, we can consider the configuration where `write` and `read` operations are tuned to occur at the same fixed rates).

A.3 Comparison with forward-secure logging

We can easily contrast the functionality of a lockbox with that of forward-secure (FS) logging schemes. In our setting, an FS logging scheme is a channel C of *unlimited capacity* where: (1) Reads are *destructive*, that is, they destroy the contents of C; and (2) An adversary $\mathcal{A}$ can control C and at all times also learn how many non-empty messages are contained in C. Specifically:

Definition 2 (FS Logging - Ideal Model.) *An FS logging scheme is a channel C of infinite size ($T = +\infty$) that can be controlled by an adversary $\mathcal{A}$ such that:*

- **Destructive reads:** *All* `read` *operations are* destructive*, i.e., they do alter the contents of the channel by deleting the produced messages from the channel . That is, two consecutive* $\texttt{read}_1$ *and* $\texttt{read}_2$ *operations on a non-empty channel C and with no* `write` *operation in between produce respectively the sequences of messages* $\texttt{read}_1() \rightarrow (m_1, \ldots, m_k) \neq \emptyset \leftarrow \texttt{read}_2()$*, where k is the number of non-empty messages in C, returned by* $\texttt{read}_1$ *operation.*
- **Count operations:** *At all times, $\mathcal{A}$ has access to a special operation* `count` *which takes no input and returns the number t of non-empty messages contained in C.*

We easily see that common practices that are based on FS logging schemes provide only immutability of messages, but neither stealth nor persistence. Indeed, the destructive nature of reads is inherent in the standard "read-once" models for logging technologies where logs are created, transmitted and consumed such that an individual log is either lost or else read only once. Note that just leaving the log entries in the sender's buffer for some amount of time could provide persistence (as reads would not be destructive in this case) but this is not common practice for current logging techniques.

B Formal security definitions

Here, we formally define our crypto-assisted protocols that realize the lockbox used in PillarBox. They can be viewed as a general-purpose cryptographic primitive for reliable, confidential and tamper-resistant message transmissions, which we call a *lockbox scheme*. We present the communication and threat models that we consider as well as the security properties that we desire our primitive to satisfy.

We consider a *sender* $\mathcal{S}$ and a *receiver* $\mathcal{R}$ that communicate through a *channel* C which by design is used to transmit—from $\mathcal{S}$ to $\mathcal{R}$—messages that belong in a fixed, bounded-size language L. We view L as a subset of a universe U of all possible messages that are transmittable through C, where, in particular, $U = L \cup \{\emptyset, \perp\}$ for a special *null* message $\emptyset$ and a special *failure* message $\perp$. Communication through C is *asynchronous*, meaning that messages in L sent by $\mathcal{S}$ through C do not necessarily reach $\mathcal{R}$ instantaneously, if at all. Instead, we view C as a *fixed-capacity memory buffer* which can be accessed through a special write/read interface and which can simultaneously send up to T distinct messages in L. In this view, the channel consists of T cells or *slots*, each used to store one message in U; we denote by $C[i]$ the content of the $(i+1)$th such slot. We also assume the existence of an index $I \in \{0, \dots, T-1\}$ specifying the *current* slot, the slot at which a new message written in the channel is to be stored. For convenience, we consider C to be of size $T+1$, i.e., $C = (C[0], C[1], \dots, C[T])$, where the current index I is stored at the last position $T+1$. We use the terms channel and buffer interchangeably.

Definition 3 (Lockbox scheme.) *A* lockbox scheme *LS comprises five probabilistic polynomial-time algorithms* $\{\mathsf{KGen}, \mathsf{KEvolve}, \mathsf{Write}, \mathsf{Wrap}, \mathsf{Read}\}$ *so that:*

- **Algorithm** KGen**:** *Key generation algorithm* KGen *takes as input security parameter κ and returns a pair $(\sigma_{w,0}, \sigma_{r,0})$ of initial* evolving *secret keys, where $\sigma_{w,0}$ (resp. $\sigma_{r,0}$) is is a secret* writing (*resp.* reading) key*, and a public key pk (to be used by all algorithms). We write* $((\sigma_{w,0}, \sigma_{r,0}), pk) \leftarrow \mathsf{KGen}(1^\kappa)$.

- **Algorithm** $\mathsf{KEvolve}$**:** *Key evolution algorithm* $\mathsf{KEvolve}$ *takes as input a secret writing $\sigma_{w,j}$ or reading $\sigma_{r,j}$ key, an integer t and auxiliary information b, and updates the writing, or reading, key to $\sigma_{w,j+t}$, or $\sigma_{r,j+t}$ respectively. We write respectively* $\sigma_{w,j+t} \leftarrow \mathsf{KEvolve}(\sigma_{w,j}, t, b)$ *and* $\sigma_{r,j+t} \leftarrow \mathsf{KEvolve}(\sigma_{r,j}, t, b)$.

- **Algorithm** Write**:** *Algorithm* Write *takes as input a secret writing key $\sigma_{w,j}$, a message $m \in \{0,1\}^\ell$ and a buffer C, encodes m in buffer C at a slot determined by $\sigma_{w,j}$ and C, updates the writing key to new key* $\sigma_{w,j+1} \leftarrow \mathsf{KEvolve}(\sigma_{w,j}, 1, b)$ *by invoking* $\mathsf{KEvolve}$ *and returns an updated buffer C'. We say that* Write adds *m in C and we write* $(\sigma_{w,j+1}, C') \leftarrow \mathsf{Write}(\sigma_{w,j}, m, C)$.

- **Algorithm** Wrap**:** *Algorithm* Wrap *takes as input a secret writing key $\sigma_{w,j}$ and a buffer C, encodes C, updates the writing key to new key* $\sigma_{w,j+1} \leftarrow \mathsf{KEvolve}(\sigma_{w,j}, 1, b)$ *by invoking* $\mathsf{KEvolve}$ *and returns an encapsulated buffer $\hat{C}$. We say that* Wrap encapsulates *C to $\hat{C}$ and we write* $(\sigma_{w,j+1}, \hat{C}) \leftarrow \mathsf{Wrap}(\sigma_{w,j}, C)$.

- **Algorithm** Read**:** *Algorithm* Read *takes as input a secret reading key $\sigma_{r,j}$ and an encapsulated buffer $\hat{C}$, decodes all buffer slots, updates the reading key to new key* $\sigma_{r,j+t} \leftarrow \mathsf{KEvolve}(\sigma_{r,j}, t, b)$ *by invoking* $\mathsf{KEvolve}$ *for some $t \geq 0$, and returns a sequence $M = (m_1, \dots, m_T)$ of T messages in $U = L \cup \{\emptyset, \perp\}$. We say that* Read produces *messages M and write* $(\sigma_{r,j+t}, M) \leftarrow \mathsf{Read}(\sigma_{r,j}, \hat{C})$.

Under normal use, a lockbox scheme generally operates as follows. First the initial evolving keys are distributed to the sender $\mathcal{S}$ and the receiver $\mathcal{R}$ by running algorithm KGen: $\mathcal{S}$ gets $\sigma_{w,0}$ and $\mathcal{R}$ gets $\sigma_{r,0}$. Also the public key pk is known to any algorithm. These keys are part of the secret states that $\mathcal{S}$ and $\mathcal{R}$ maintain at all times. Evolving keys implement forward security for symmetric-key schemes. With an initially empty buffer C, where the current index is initially set to point to a random position in C, sender $\mathcal{S}$ can then start adding messages in an encoded format in C using algorithm Write. At any point in time, algorithm Wrap can be used by $\mathcal{S}$ to finalize the current contents of buffer C into an encapsulation format $\hat{C}$ which is then transmitted to receiver $\mathcal{R}$. Each such received encapsulated buffer $\hat{C}$ can be unpacked, decoded and finally read by $\mathcal{R}$ using algorithm Read. Finally, algorithms Write, Wrap and Read use as a subroutine algorithm $\mathsf{KEvolve}$. We note that algorithm Read needs not run in synchronization with algorithms Write and Wrap: in particular, keys $\sigma_{w,j}$ and $\sigma_{r,j}$ need not evolve in coordination or at the same pattern or speed; instead, at a given point in time $\mathcal{S}$ and $\mathcal{R}$ generally store keys σ_{w,j_w} and σ_{r,j_r} respectively, with $j_w \geq j_r$.

At a minimum, under normal operation, we expect a lockbox scheme to *reliably* transmit messages in L, in a *order-preserving* manner within individual encapsulated buffers. Accordingly, we define:

Definition 4 (Correctness.) *We say that lockbox scheme* $LS = \{\mathsf{KGen}, \mathsf{KEvolve}, \mathsf{Write}, \mathsf{Wrap}, \mathsf{Read}\}$ *is* correct *if it holds that:*

1. *There exists a buffer* $C_\emptyset$ *so that its corresponding encapsulation* $\hat{C}_\emptyset$ *produces the empty-message sequence* $E = (\emptyset, \ldots, \emptyset)$*. That is, for any* j_w*, there exists* $C_\emptyset$ *such that* $(\sigma_{w,j_w+1}, \hat{C}_\emptyset) \leftarrow \mathsf{Wrap}(\sigma_{w,j_w}, C_\emptyset)$ *and also there exist* j_r *and* $t \geq 0$ *such that* $(\sigma_{r,j_r+t}, E) \leftarrow \mathsf{Read}(\sigma_{r,j_r}, \hat{C}_\emptyset)$*.*

2. *Let* $C_\emptyset$ *be as above. Then if any* $k > 0$ *non-empty messages are added in* $C_\emptyset$*, a corresponding encapsulation will produce exactly* $\min\{T, k\}$ *most recent such non-empty messages. That is, for any* j_w*,* k*,* s*, any sequence of messages* $(m_1, \ldots, m_k) \in L^k$*, and any bit string* $(b_1, \ldots, b_{k+s}) \in \{0,1\}^{k+s}$ *such that* $\sum_{i=1}^{k+s} b_i = s$*, if*

   (a) $C_1 = C_\emptyset$*,*

   (b) *for* $1 \leq l \leq k+s$*, if* $b_l = 0$ *then* $(\sigma_{w,j_w+l}, C_{l+1}) \leftarrow \mathsf{Write}(\sigma_{w,j_w+l-1}, m_l, C_l)$ *or otherwise* $(\sigma_{w,j_w+l}, \hat{C}_l) \leftarrow \mathsf{Wrap}(\sigma_{w,j_w+l-1}, C_l)$ *and* $C_{l+1} = C_l$*, and*

   (c) $(\sigma_{w,j_w+k+s+1}, \hat{C}) \leftarrow \mathsf{Wrap}(\sigma_{w,j_w+k+s}, C_{k+s+1})$*,*

   then, with all but negligible probability:

   (a) *there exist unique index* j_r *and* $t \geq 0$ *so that* $(\sigma_{r,j_r+t}, M) \leftarrow \mathsf{Read}(\sigma_{r,j_r}, \hat{C})$*, and*

   (b) *if* $k < T$ *then* $M = (\emptyset, \ldots, \emptyset, m_1, \ldots, m_k)$*, or otherwise* $M = (m_{k-T+1}, \ldots, m_{k-1}, m_k)$*.*

Further, we capture the two main security properties of a lockbox scheme, *immutability* and *stealth*, using two corresponding security games played by an adversary $\mathcal{A}$ while having access to a special oracle that makes use of the lockbox algorithms. In particular, we define the following oracle:

Oracle $\mathtt{WriteO}_\sigma$**.** On input a possibly empty sequence of messages $(m_1, \ldots, m_k) \in U^k$ and keeping state $(\sigma_{w,j}, C)$, $\mathtt{WriteO}_\sigma$ updates its state to $(\sigma_{w,j+k+1}, C_k)$ and returns encapsulated buffer $\hat{C}$, where $(\sigma_{w,j+k+1}, \hat{C}) \leftarrow \mathsf{Wrap}(\sigma_{w,j+k}, C_k)$, $(\sigma_{w,j+l}, C_{l+1}) \leftarrow \mathsf{Write}(\sigma_{w,j+l-1}, m_l, C_l)$, $1 \leq l \leq k$, and $C_0 = C_1 = C$.

We consider a powerful adversary that *fully compromises* the sender $\mathcal{S}$, thus capturing its secret state and getting control over the current buffer C. Prior to the compromise, the adversary is assumed to *actively control* the transmissions between $\mathcal{S}$ and the receiver $\mathcal{R}$. In particular, the adversary may adaptively select the messages that $\mathcal{S}$ adds in its buffer C and when to encapsulate C to $\hat{C}$. Moreover, each such encapsulated buffer produced by $\mathcal{S}$ is handed back to the adversary who then decides either to deliver it, or to indefinitely deny delivery of it, or to modify $\hat{C}$ to one or more encapsulated buffers of its choice which are delivered to $\mathcal{R}$. That is, the adversary can effectively perform arbitrary modifications, deletions, injections or reordering to the set of encapsulated buffers produced by $\mathcal{S}$ before their delivery to $\mathcal{R}$. This means that the adversary is in fact the communication channel (at the transmission level), similarly to the Dolev-Yao adversarial model. Overall, the adversary acts adaptively, including the time at which it chooses to compromise $\mathcal{S}$.

First, immutability refers to an integrity property for the sequence of messages that are produced by the receiver: any received non-empty message is either an invalid message that the adversary has tampered with (e.g., modified or deleted) or a *valid* message in L that *(1) has been written in the channel after the time the most recently received message was written in the channel and (2) has arrived while preserving its order*. And this holds even when the adversary launches an adaptive chosen message attack prior to or after the compromise (similar to the standard notion of security for digital signatures [22]) and learns the secret state of $\mathcal{S}$ at the time of compromise. More formally:

Definition 5 (Immutability.) *We say that lockbox scheme* $LS = \{\mathsf{KGen}, \mathsf{KEvolve}, \mathsf{Write}, \mathsf{Wrap}, \mathsf{Read}\}$ *is* immutable *if no PPT adversary* $\mathcal{A}$ *can win non-negligibly often in the security parameter* κ *in the following game:*

- **Initialization:** KGen *runs on* κ $((\sigma_{w,0}, \sigma_{r,0}), pk) \leftarrow \mathsf{KGen}(1^\kappa)$ *and oracle* $\mathtt{WriteO}_\sigma$ *is initialized with* $(\sigma_{w,0}, C_\emptyset)$ *where* $C_\emptyset$ *is the empty buffer with corresponding encapsulated buffer* $\hat{C}_\emptyset$*.*
- **Phase I:** $\mathcal{A}$ *is given the empty encapsulated buffer* $\hat{C}_\emptyset$ *and access to oracle* $\mathtt{WriteO}_\sigma$*. That is,* $\mathcal{A}$ *makes arbitrary use of* $\mathtt{WriteO}_\sigma$ *on inputs* $\mu_1, \ldots, \mu_{l_1}$, $\mu_i = (m^i_1, \ldots, m^i_{z_i})$, $i \in \{1, \ldots l_1\}$*, all of its choice, where* $\sum_{i=1}^{l_1} z_i = k_1$*. At all times,* $\mathcal{A}$ *may query* Read *on any encapsulated buffer* $\hat{C}$ *on its choice to get the sequence of produced messages* $M_{\hat{C}}$ *corresponding to* $\hat{C}$*. At any time of its choice,* $\mathcal{A}$ *proceeds to the next phase.*
- **Phase II:** $\mathcal{A}$ *is given the state* $(\sigma_{w,j}, C)$ *of oracle* $\mathtt{WriteO}_\sigma$*, where* $(\sigma_{w,j}, \hat{C}) \leftarrow \mathsf{Wrap}(\sigma_{w,j-1}, C)$ *is the last invocation of* Wrap *by* $\mathtt{WriteO}_\sigma$ *in phase I. Then* $\mathcal{A}$ *may run* Write *and* Wrap *on inputs* $(m^{l_1+1}_1, C'_1), \ldots, (m^{l_1+1}_{k_2}, C'_{k_2})$ *and* $\bar{C}_1, \ldots, \bar{C}_{l_2}$ *of its choice, where* $k_1 + k_2 = k$, $l_1 + l_2 = l$ *and* $k, l \in poly(\kappa)$*. At all times,* $\mathcal{A}$ *may query* Read *on any encapsulated buffer* $\hat{C}$ *on its choice to get the sequence of produced messages* $M_{\hat{C}}$ *corresponding to* $\hat{C}$*. At any time of its choice,* $\mathcal{A}$ *proceeds to the attack phase.*
- **Attack:** *Finally,* $\mathcal{A}$ *outputs an encapsulated buffer* $\hat{C}^*$*.*

Let $\mathcal{M}$ *be the sequence of all messages produced by* $\mathcal{R}$ *by invoking* Read *on every buffer* $\hat{C}$ *encapsulated in phases I and II through* Wrap *in the same order as these buffers were encapsulated. Then, let* $M^* = (m_1, \ldots, m_T)$ *denote the messages that are produced by running* Read *on* $\hat{C}^*$*. If* $m = m^{j_1}_{i_1}$ *and* $m' = m^{j_2}_{i_2}$ *are messages written in the channel in phase I or II above, we say that* m precedes m' *if* $j_1 < j_2$ *or* $j_1 = j_2$ *and* $i_1 < i_2$*, i.e., if* m *was written in the channel before* m'*.*

We say that $\mathcal{A}$ wins *if* any *of the following three occurs:*

1. *There exists a message* $m^* \notin \{\emptyset, \perp\}$ *such that* $m^* \in \mathcal{M} \cup M^*$ *but at the same time* $m^* \notin \mu_i$*, for all* $1 \leq i \leq l_1$*, and* $m^* \notin \{m^{l_1+1}_1, \ldots, m^{l_1+1}_{k_2}\}$*.*
2. *There exist messages* $m^*, m^{**} \in M^* = (m_1, \ldots, m_T)$*, such that* $m^* = m_i \notin \{\emptyset, \perp\}$ *and* $m^{**} = m_j \notin \{\emptyset, \perp\}$ *with* $i > j$ *but at the same time* m^* *precedes* m^{**}*.*
3. *There exist messages* $m^*, m^{**} \in M^*$ *with* $m^*, m^{**} \notin \{\emptyset, \perp\}$*, where* m^* *precedes* m^{**} *by more than* $T - 1$ *messages.*

Second, stealth refers to a privacy property for the set of messages that are encoded and encapsulated by the sender: any encapsulated buffer satisfies ciphertext indistinguishability with respect to their contents. In particular, the adversary $\mathcal{A}$ cannot distinguish if a given encapsulated buffer contains one of two messages $\mathcal{A}$ selected, or distinguish if the encapsulated buffer contains one adversarially selected message or not. And this holds, in the strongest possible setting where the adversary learns the secret state of $\mathcal{S}$ at the time of compromise and launches some sort of adaptive chosen ciphertext attack prior to or after the compromise (i.e., similar to IND-CCA2), where access to the encryption oracle is unrestricted and access to the decryption oracle is only restricted to prevent trivial decryption of the challenge given to the adversary. More formally:

Definition 6 (Stealth.) *We say that lockbox scheme $LS = \{\mathsf{KGen}, \mathsf{KEvolve}, \mathsf{Write}, \mathsf{Wrap}, \mathsf{Read}\}$ is* stealthy *if no PPT adversary $\mathcal{A}$ can win non-negligibly often in the security parameter κ in the following game against a challenger $\mathcal{C}$:*

- **Initialization:** *KGen runs on κ $((\sigma_{w,0}, \sigma_{r,0}), pk) \leftarrow \mathsf{KGen}(1^\kappa)$ and oracle $\mathtt{WriteO}_\sigma$ is initialized with $(\sigma_{w,0}, C_\emptyset)$, where $C_\emptyset$ is the empty buffer with corresponding encapsulated buffer $\hat{C}_\emptyset$.*
- **Phase I:** *$\mathcal{A}$ is given the empty encapsulated buffer $\hat{C}_\emptyset$ and access to oracle $\mathtt{WriteO}_\sigma$. That is, $\mathcal{A}$ makes arbitrary use of $\mathtt{WriteO}_\sigma$ on inputs $\mu_1, \ldots, \mu_{l_1}$, $\mu_i = (m_1^i, \ldots, m_{z_i}^i)$, all of its choice, where $\sum_{i=1}^{l_1} z_i = k_1$. At all times, $\mathcal{A}$ may query Read on any encapsulated buffer $\hat{C}$ on its choice to get the sequence of produced messages $M_{\hat{C}}$ corresponding to $\hat{C}$. At any time of its choice, $\mathcal{A}$ proceeds to the next phase.*
- **Selection:** *$\mathcal{A}$ forms messages m_0, m_1 and m^*.*
- **Challenge:** *$\mathcal{C}$ flips a random bit $b \xleftarrow{\$} \{0,1\}$ and is given (m_0, m_1), m^* and access to oracle $\mathtt{WriteO}_\sigma$, used by $\mathcal{A}$ in phase I. Then:*
  - **Case I:** *$\mathcal{C}$ invokes $\mathtt{WriteO}_\sigma$ on input m_b to compute encapsulated buffer $\hat{C}^*$, or*
  - **Case II:** *$\mathcal{C}$ invokes $\mathtt{WriteO}_\sigma$ on input c to compute encapsulated buffer $\tilde{C}^*$, where $c = m^*$ if $b = 0$ and $c = \emptyset$ (empty set) if $b = 1$.*
- **Phase II:** *$\mathcal{A}$ is given the encapsulated buffer $\hat{C}^*$ or $\tilde{C}^*$ computed in the challenge phase and the state $(\sigma_{w,j}, C)$ of $\mathtt{WriteO}_\sigma$, where $(\sigma_{w,j}, \hat{C}) \leftarrow \mathsf{Wrap}(\sigma_{w,j-1}, C)$ is the last invocation of Wrap by $\mathtt{WriteO}_\sigma$ in the challenge phase. Then $\mathcal{A}$ may run Write and Wrap on inputs $(m_1^{l_1+1}, C'_1), \ldots, (m_{k_2}^{l_1+1}, C'_{k_2})$ and respectively $\bar{C}_1, \ldots, \bar{C}_{l_2}$ of its choice, where $k_1 + k_2 = k$, $l_1 + l_2 = l$ and $k, l \in poly(\kappa)$. At any time of its choice, $\mathcal{A}$ proceeds to the attack phase.*
- **Attack:** *Finally, $\mathcal{A}$ outputs a bit $\hat{b}$.*

We say that $\mathcal{A}$ wins *if $\hat{b} = b$ in either case I or II above.*

Finally, we mention that a lockbox scheme may satisfy the following non-cryptographic property of persistence. We say that a lockbox scheme is *persistent* if T sequential writings of messages $m_1, \ldots, m_T$ in C, each followed by a Wrap operation, result in encapsulated buffers $\hat{C}_1, \ldots, \hat{C}_T$ that produce m_1 exactly T times. That is, by reading the channel frequently enough (through separate encapsulations), it is possible to produce a given message more than one times. This is a particularly important property in cases where the underlying transmission channel is lossy.

C Cryptographic primitives

To cryptographically support PillarBox, we make use of the following two cryptographic primitives.

C.1 Authenticated encryption

This primitive provides combined confidentiality and integrity protection over transmitted messages.

More formally, in the symmetric key setting, an authenticated encryption scheme is a triple of algorithms ($\mathsf{AEKeyGen}$, $\mathsf{AuthEnc}$, $\mathsf{AuthDec}$), defined as follows:

- Algorithm $\mathsf{AEKeyGen}$ takes as input a security parameter and returns a secret key σ_{AE} to be used by the encryption and decryption algorithms. We write $\sigma_{AE} \leftarrow \mathsf{AEKeyGen}(1^\kappa)$.
- Algorithm $\mathsf{AuthEnc}$ takes as input the secret key σ_{AE} and a message m and returns a ciphertext c. We write $c \leftarrow \mathsf{AuthEnc}_{\sigma_{AE}}(m)$.
- Algorithm $\mathsf{AuthDec}$ takes as input a secret key σ_{AE} and a ciphertext c and either returns m or $\perp$, the latter denoting invalid, i.e., non-authenticated, encryption. We write $\{m, \perp\} \leftarrow \mathsf{AuthDec}_{\sigma_{AE}}(c)$.

With respect to the security of an authenticated encryption scheme, we have the combined properties of an encryption scheme and a signature scheme (or MAC), i.e., privacy and integrity. That is, if the ciphertext has not been tampered with decryption returns the original encrypted message, whose confidentiality is protected by the ciphertext. For more information, see the work by Bellare and Namprempre [16], where in particular it is shown that an encrypt-then-MAC scheme provides NM-CPA secrecy and INT-PTXT, provided that the MAC scheme is strongly unforgeable.

Public-key variants are possible.

In our construction, we use a forward-secure variant of the above authenticated encryption scheme, where the secret key σ_{AE} evolves over time through a forward-secure pseudorandom number generator.

C.2 Forward-secure PRNGs

Forward-secure pseudorandom number generators (FS-PRNGs) support a form of leakage-resilient cryptography by providing a sequence of strong cryptographic keys that achieve *forward security*: In a system where the secret key evolves over time, being refreshed by a new key (the next in the sequence) if the system's current secret state is leaked to an attacker, then the system's operation with respect to older keys remains secure.

More formally, an FS-PRNG scheme is a pair of algorithms (GenKey, Next), defined as follows:

- Algorithm GenKey takes as input a security parameter and returns an initial state s_0. We write $s_0 \leftarrow \mathsf{GenKey}(1^\kappa)$.
- Algorithm Next takes as input the current state s_i, $i \geq 0$, and an integer $t > 0$, updates the state to s_{i+t} and returns a pseudorandom number r_{i+t-1}. We write $(r_{i+t-1}, s_{i+t}) \leftarrow \mathsf{Next}(s_i, t)$. For simplicity, $r_{i+t} \leftarrow \mathsf{Next}(r_i, t)$ denotes the generation of the pseudorandom number that is t steps ahead,[16] and for the common case where $t = 1$ and $i > 0$, we write $r_{i+1} \leftarrow \mathsf{Next}(r_i)$ to denote the generation of the next pseudorandom number.[17]

With respect to the security of an FS-PRNG scheme, it holds that given $\{s_{i+t}\}$, $t > 0$, it is computationally hard to compute any older pseudorandom number r_j, $j \leq i$.

C.3 Possible instantiations

The cryptographic primitives of authenticated encryption and FS-PRNG schemes as defined above can be instantiated using different existing techniques. Below we discuss some such possibilities.

[16] We abuse notation and more conveniently use as input the number r_i instead of the state s_i and state s_{i+1} is omitted as output.

[17] The intuition is better, because for the common case where $t = 1$, we can consider that the sequence $\mathbf{r} = (r_0, r_1, r_2, \ldots)$ corresponds to a one-way hash chain.

Concrete authenticated encryption schemes. Six different authenticated encryption modes, namely OCB 2.0, Key Wrap, CCM, EAX, GCM and Encrypt-then-MAC, have been standardized in ISO/IEC 19772:2009 (Authenticated encryption).

Moreover, a simple "MAC-then-encrypt" type authenticated encryption scheme can be defined as follows:

$$\mathsf{AuthEnc}_K(m) = K \oplus (m \parallel MAC_r(m) \parallel r)$$

for a random r, where K is the secret key σ_{AE} of the scheme and MAC_r is any keyed message authentication code. Here, the sizes of m, $MAC_r(m)$ and r should be fixed and their sum should be equal to the size $|K|$ of K. In this case, $\mathsf{AuthDec}_K(c)$ is defined in the obvious way through the following steps:

1. Compute $d = c \oplus K$.
2. View d as a string of type $m \parallel t \parallel r$, where the sizes of m, t and r are fixed and known.
3. Compute $t' = MAC_r(m)$.
4. If $t' = t$ then return m, else return $\perp$.

Alternatively, an "Encrypt-then-MAC" type authenticated encryption scheme can be used (see, e.g.,[39]), defined as follows. The secret key σ_{AE} is a pair of numbers (s, k). The encryption algorithm $\mathsf{AuthEnc}_{\sigma_{AE}}(m)$ is defined using a pseudorandom function F_s mapping n bits to l bits, where $|m| = l$, and a MAC which is an unpredictable hash function H_k mapping $n + l$ bits to w bits through the following steps:

1. Randomly choose string x of size n bits.
2. Set $c \leftarrow F_s(x) \oplus m$.
3. Set $t \leftarrow H_k(x \parallel c)$.
4. Return $\psi = (x, c, t)$.

Finally, the decryption algorithm $\mathsf{AuthDec}_K(\psi)$ is defined through the following steps:

1. View ψ as (x, c, t), where $|x| = n$, $|c| = l$ and $|t| = w$.
2. If $t = H_k(x \parallel c)$ then return $m = F_s(x) \oplus c$, else return $\perp$.

Concrete FS-PNG schemes. Our construction may generally use any existing FS-PRNG scheme. In the simplest form, an FS-PNG scheme may correspond to a one-way hash chain as follows: Starting from an initial random seed s_0, apply a suitable one-way function f_0 to get $s_{i+1} = f_0(s_i)$ and a suitable one-way function f_1 to get $r_i = f_1(s_i)$. Here, typical instantiation of an one-way function may be a pre-image resistant hash function, e.g., the SHA-1 family of hash functions.

In a survey of forward-secure cryptography [24], Itkis characterizes an FS-PRNG in terms of a binary tree. The root is a random seed s. A left child is derived from its parent by application of a suitable one-way function f_0, the right child by application of f_1. A node at depth d may then be defined in terms of a binary label $B = b_0 b_1 \ldots b_d$ specifying the sequence of applications of the one-way functions, where b_i denotes application of f_{b_i}. Itkis notes that an FS-PRNG may be defined for any (in)finite set of labels $\mathcal{L}$ that is *prefixless*: The t^{th} output l_t corresponds to the t^{th} largest label in $\mathcal{L}$.[18]

[18]The intuition here is as follows. Suppose there are two labels a and b, such that a is a prefix of b. Then a is an ancestor of b. Thus, after a is output, it is necessary to *retain* a in order to compute b. But retaining a after a has already been output compromises forward security.

The simplest such scheme is one in which $\mathcal{L}$ consists of all labels of the form 0^t1. This simple scheme is used in, e.g., [17, 28]. A drawback is that computing an arbitrary output l_t requires t computations. Itkis proposes a simple scheme that requires only $O(\log t)$ computations to compute l_t from s: Let the label B for l_t be a prefixless encoding of t followed by t itself. However, Itkis does not consider the *storage* costs for his proposed FS-PRNG, which are linear in t.

To rectify this omission, we next briefly discuss a novel refinement of the FS-PRNG family defined by Itkis [24] that instead of balanced (binary) trees employs an unbalanced (multi-way) tree to achieve optimal trade-offs between the basic two cost associated parameters: (1) the storage needed to maintain state; and (2) the time needed to advance the state by one or more steps in a forward secure way. This unbalanced tree consists of hierarchical one-way hash chains recursively connected as follows: Starting from a random master seed and using an initial one-way hash function, a high-level infinite (one-way hash) chain defines (produces) seeds, each of which, in turn, defines the head/root of a lower-level finite-size chain using a distinct one-way hash function, and so forth, until a maximum number of such levels have been used. Then, pseudorandom numbers are produced by using a new one-way hash function over the "leaf" seeds defined (by the chains at) the lowest level, in the ordering defined by considering seeds at intermediate levels according to increasing (or "left-to-right") order in their chain. A scheme like this with k levels and intermediate chains of size ℓ, has the advantage that by only keeping $O(k)$ state (namely, one unused seed per level), advancing the FS-PRNG state by t steps requires $O(t\ell^{1-k} + k\ell)$ computations, which is $O(\log t)$ when k is $O(\log t)$. In essence, in our FS-PRNG scheme above, higher-level chains serve as shortcut paths that allow very efficient transitions from state s_j to state s_{j+t} even for steps t that are exponentially large in the number of levels k.[19] (However, these type of FS-PRNG optimizations are outside the scope of our work and not considered in our PillarBox implementation.)

D Security analysis

For our main construction of Section 4.4, we can show the following result.

Theorem 1 *The message-locking channel* $LS = (\mathsf{KGen}, \mathsf{KEvolve}, \mathsf{Write}, \mathsf{Wrap}, \mathsf{Read})$ *described in Section 4.4 is correct, immutable and stealthy according to definitions presented in Appendix B.*

We next briefly provide the intuition behind the above useful properties.

Correctness. Since messages are included in the buffer in a FIFO manner, i.e., in a first-in-first-out manner, and since the indices of both the low layer and the high layer encryption keys are included in the encapsulated buffer (step 2 in algorithms Write and Wrap), under normal operation of the system, it is always possible for the receiver to find the corresponding secret keys for decrypting and reconstructing the exact sequence of the T most recent messages that were written in the channel before the encapsulation of the buffer.

Immutability. Since any message that is added in the buffer is encrypted in a verifiable way through the low layer authenticated encryption scheme (steps 1 and 2 in algorithm Write), individual messages cannot be undetectably tampered with (enforced by step 8.f in algorithm Read). Additionally, since the indices of the secret keys of the low layer authenticated encryption are authenticated through the high layer authentication

[19]This is a very important property for any FS-PRNG scheme to be practically useful in support of PillarBox: As messages often reach the receiver out of order or do not reach it at all, the receiver needs to be capable of efficiently and on-demand synchronizing its secret FS-PRNG state with the, in general, constantly varying state (or sequence number) of any received buffer, or routinely catching up with the correct current state of the PillarBox host, after possibly long periods of interrupted transmissions, without regenerating all intermediate states from scratch.

encryption (step 2 in algorithm Write and step 1 in algorithm Wrap), messages belonging in an individual encapsulated buffer cannot be undetectably received out of order, either across different buffers (step 8.b in algorithm Read) or within the same buffer (step 8.d.i in algorithm Read).

Stealthiness. Since both individual messages and encapsulated buffers are encrypted through the low layer and respectively high layer authenticated encryption, and since the buffer is of fixed size T, observing encapsulated buffers before the compromise or actual buffer contents after the compromise reveals no information about the actual messages previously added in the channel or about whether messages have been ever added in the channel.