Securing the Data in Big Data Security Analytics

Kevin D. Bowers
RSA Laboratories
kevin.bowers@rsa.com

Catherine Hart*
Bell Canada
catherine.hart@bell.ca

Ari Juels*
ajuels2@gmail.com

Nikos Triandopoulos
RSA Laboratories
nikolaos.triandopoulos@rsa.com

ABSTRACT

Big data security analytics is an emerging approach to intrusion detection at the scale of a large organization. It involves a combination of automated and manual analysis of security logs and alerts from a wide and varying array of sources, often aggregated into a massive ("big") data repository. Many of these sources are host facilities, such as intrusion-detection systems and syslog, that we generically call *Security Analytics Sources (SASs)*.

Security analytics are only as good as the data being analyzed. Yet nearly all SASs today lack even basic protections on data collection. An attacker can *undetectably suppress* or *tamper* with SAS messages to conceal attack evidence. Moreover, by merely monitoring network traffic they can *discover* sensitive SAS instrumentation and message-generation behaviors.

We introduce *PillarBox*, a tool for securely relaying SAS messages in a security analytics system. PillarBox enforces *integrity*: It secures SAS messages against tampering, even against an attacker that controls the network and compromises a message-generating host. It also (optionally) offers *stealth*: It can conceal alert generation, hiding select SAS alerting rules and actions from an adversary.

We present an implementation of PillarBox and show experimentally that it can secure messages against attacker suppression or tampering even in the most challenging environments where SASs generate real-time security alerts. 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 big data 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 [16]. 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 (NIDS), 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/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 5.1.) Logged or buffered SAS messages are vulnerable to deletion or modification after host compromise.

Consider, for instance, a rootkit Trojan that exploits a host vulnerability to achieve a privilege escalation on an enterprise server. 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, Infostealer.Shiz, Android.Bmaster).[1] 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 the Aurora attack,[2] Stuxnet and Duqu, 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 [14].

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 flaw in big data security analytics architectures.

*Work performed while at RSA Labs

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

[2] `http://www.mcafee.com/us/threat-center/aurora-enterprise.aspx`

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.

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 sniffs 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*. Self-protection 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., don't 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 a security analytics systems.

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.

[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).

We emphasize that we don't address the design of SASs in this paper. How SAS messages are generated and the content of messages are outside the scope of this paper. 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. Section 5 demonstrates a simple attack on existing SAS systems and presents a prototype implementation and experimental evaluation of PillarBox. We review related work in Section 6 and conclude in Section 7. Cryptographic formalisms 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 Adversarial model

We model our setting in terms of three entities, the *SAS* (which we refer to interchangeably as the *host*), the *attacker* (also called the *intruder*), and the *server*. We model 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., an adversary that only attacks the network or attacks a firewall whose SAS only reports on network events.)

Recall that in the self-protecting case, a SAS generates messages 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 [5]. 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. The server itself is a trusted entity: It is not vulnerable to attack.

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.,

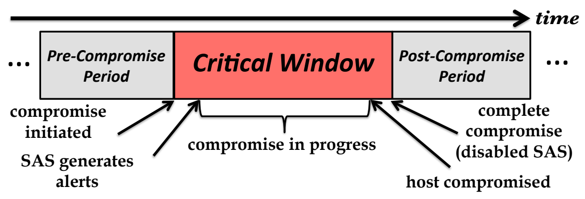

Figure 1: Event timeline of host compromise

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.

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. The attacker makes 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. Vanishing rules (rules that are deleted if they ever trigger an alert) can be used to protect against adversarial rule discovery in the SAS. By analogy with cryptographic indistinguishability definitions, a concise definition of stealth is possible: An attacker violates stealth if, for any SAS detection rule, it can distinguish between PillarBox instantiations with and without the rule.

2.2 Secure alert relaying via buffering

A key design choice in PillarBox, as mentioned, is the use of host-side alert buffering. We now explain why buffering is important to secure the SAS chain of custody in PillarBox and how we address the technical challenges it introduces.

For brevity, we refer to the *PillarBox buffer* 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.

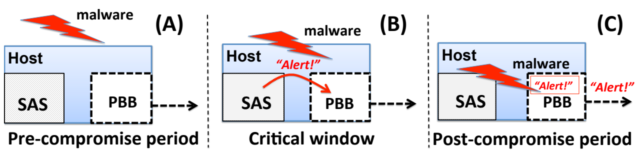

Figure 2: PillarBox across compromise phases: (A) The intruder hasn't yet attacked the host. (B) The SAS detects in-progress compromise and places an alert in the PillarBox buffer (PBB). (C) The intruder has complete control of the host, but the PBB securely stores and transmits the alert.

Why buffering is necessary. The approach of most SAS systems today, e.g., syslog and IDSs, 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 *can't 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 can't ensure or even verify delivery of transmitted messages. E.g., wireless sensor networks[4] often experience transmission failures.

- *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,[5],[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 below.)

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 doesn't solve this problem: Traffic analysis alone can reveal SAS rule-triggering. (As noted above, PillarBox overcomes this problem via regular alert-buffer transmission.)

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's no way of course to prevent it from disabling a SAS or tampering with its *future* outputs. But there's 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.

2.3 Integrity

One main challenge in creating a secure chain of custody in PillarBox is the need to secure alert messages *after* compromise, while they're 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.

At first glance, this might seem unachievable. A digital signature or message-authentication code (MAC) *alone*, as proposed, e.g., for syslog [11], *doesn't* protect against tampering: After host compromise an attacker learns the signing key and can forge messages. Message encryption similarly doesn't 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

[4]These systems don't have full-blown SASs but can have lightweight versions, e.g., the hardware tripwire proposed for authentication tokens [8].

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

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

forward-secure logging (e.g., [13, 18, 24, 25]), an application closely related to SAS protection. Forward-secure logging systems, however, are designed mainly for forensic purposes rather than detection, e.g., to protect against administrator tampering after the fact. Because of this, some forward-secure logging systems "close" logs only periodically. What's new in the use of forward security here is primarily its application for self-protecting alerting.

PillarBox uses *forward-secure pseudorandom number generation* (FS-PRNG) to create MAC keys. An FS-PRNG has the property that past keys can't be inferred from a current key. The FS-PRNG run in a PillarBox host is also run on the server to establish shared keys. The host, however, deletes a MAC key as soon as it's used. Thus an attacker can't modify buffered messages, but the server can detect tampering or erasure, using indexing of keys.

Recall that the other aspect of integrity in PillarBox 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 our experimental validation that winning this race, and thus the whole PillarBox approach to securing alerts, is feasible. Our experiments in Section 5 show this to hold even against a fast, local, PillarBox-aware attacker that tries to kill PillarBox processes as quickly as possible.

2.4 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 [14].)

Straightforward encryption alone doesn't achieve stealth. For example, if buffer alerts are encrypted on a host, an attacker can infer alert generation simply by counting buffer ciphertexts upon host compromise. Similarly, encrypted host-to-server traffic leaks information: An attacker can determine via traffic analysis when a host has triggered an alert. An attacker can even perform black-box probing attacks against a host to test attacks and infer which are or aren't detectable. Stealth in PillarBox thus requires a combination of several ideas.

PillarBox employs a buffer size T, and buffer transmission-time interval μ, that are *fixed*, i.e., invariant. Each message is also of fixed size[7]. When PillarBox transmits, it re-encrypts and sends the entire fixed-size buffer, not just fresh alerts. PillarBox's 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.

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: Temporarily suppressed messages may eventually arrive at the server.) The 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; in this case, the buffer is also large enough so that if round-robin overwriting occurs, it signals to the server a "buffer-stuffing" attempt by an attacker. (Below we develop a framework for parameterization of T and μ and then 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.[8]

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

Vanishing detection rules. A key complement to stealth in PillarBox is concealment of detection rules in hosts: Encryption alone ensures such confidentiality in the buffer, but not in the SAS alerting engine itself. In our experiments, however, we show the viability of instrumenting the SAS with vanishing rules, as described below.

3. ARCHITECTURE

We next describe our general architecture for PillarBox and its main software components used to secure the host-to-server chain of custody in a SAS system (cf. Figure 3).

As explained, PillarBox secures alerts in the PBB to protect against tampering, suppression or leakage, even in post-compromise environments. Using simple cryptographic protections and buffer transmission protocols, the PBB essentially implements a low-level reliable channel that ensures tamper-evident and stealthy delivery of transmitted messages, even under adversarial conditions.

With this functionality, 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 can't 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's 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

[7] We discuss the option to have variable-sized messages in Section 4.

[8] Semantically secure public-key encryption would enable use of just one layer, but with impractically high computational costs.

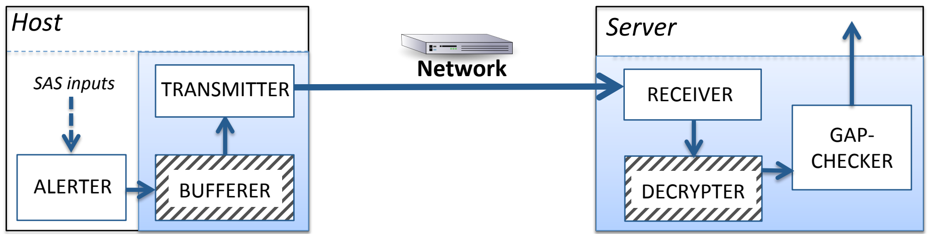

Figure 3: PillarBox architecture and flow of information. Shaded areas show the PillarBox components (which exclude the alerter) and striped ones (the bufferer and decrypter) those that make up PillarBox's crypto-assisted core reliable channel.

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, with monitoring 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.[9] 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 (e.g., those that seem to indicate impending compromise) to PillarBox, and others 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. It receives messages from the alerter and inserts them into the PBB. It also receives buffer-wrapping requests from the transmitter (see below) and responds by returning the current buffer contents in a securely encapsulated form (see Section 4). This module is also responsible for maintaining the secret state of the PBB and updating cryptographic keys. The bufferer accepts two calls: A Write call from the alerter to insert a message to the PBB and a Wrap call from the transmitter requesting encapsulated export of the buffer contents. The bufferer doesn't discard messages from the buffer when they're transmitted. A message is encapsulated and transmitted until overwritten. (While a byproduct of stealth, this feature of persistence can also be leveraged to accommodate lossy networks, as explained below.)

Transmitter. This module schedules and executes buffer transmissions from the host to the server. Transmissions may be scheduled every μ seconds, for parameter μ, 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 labels messages with their (verified) buffer 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 signalling 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 buffer 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 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's 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 interval μ (the time between "snapshots" of buffers sent by the host) is at most T/τ seconds, a normal host won't trigger a false gap alert.

We characterize τ, the maximum SAS message-generation rate of normal hosts, in Section 5. 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, the more repeated transmissions of every message.

Handling message disruptions. If an attacker suppresses buffer transmission completely, the gap-checker will cease to receive buffers.

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

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.

PillarBox can't 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*. Its connection is restored only when PillarBox again receives a buffer from the host and can detect alerts. In a benign network outage, this policy won't 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. Thus such interfacing of PillarBox with network-access policies *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). The PBB and its protocols realize a reliable messaging channel for PillarBox. We discuss the functionality that the PBB exports to the alerter and gap-checker to secure the SAS chain of custody.

4.1 Ideal "lockbox" security model

In its *ideal* form, the PBB serves as a "lockbox" for message transport: It's a buffer consisting of T fixed-size slots that supports the following two 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 current 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 and s_j of the buffer and respectively the current message, along with the T messages in the buffer starting at position $s_j \bmod T$, with wraparound.

In this ideal model, buffered messages 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 corresponding to slot i is guaranteed to be either: (1) The most recent message written to slot i (the empty message $\emptyset$ if no message was ever written), or (2) A special corruption symbol $\perp$ that indelibly replaces all the buffer's contents if the buffer was tampered with or modified otherwise than 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 $\perp$.

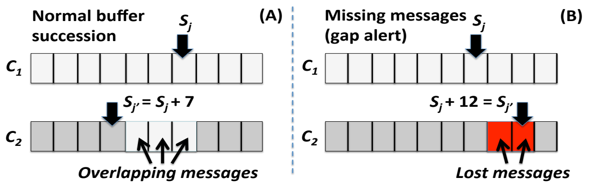

Figure 4: Gap rule example on successively received buffers C_1, C_2, with indices j, j' and $T = 10$: (A) Normal message overlap between buffers; (B) A detectable gap: Messages with sequence numbers $s_j + 1$ and $s_j + 2$ have been lost.

2. *Buffer overwriting:* The attacker can exploit buffer wraparound to refill the PBB, by writing T relatively benign messages into it to *overwrite* and thereby destroy messages generated during the critical window.

3. *Buffer dropping:* The attacker can simply drop buffers or delay their transmission.[10]

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 5 show in some settings of interest that this attack is infeasible.

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.[11] 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 "Lockbox" security definitions

Our actual construction 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 $\mathcal{R}$. We require two security properties, *immutability*, and *stealth* and two non-cryptographic ones, *persistence* and *correctness*. We now give brief informal descriptions; formal definitions are in Appendix A.

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's possible to produce a given message in C more than once.

[10]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.

[11]That is, a gap alert is issued when the current sequence numbers s_j and $s_{j'}$ of two successively received buffers are such that $s_{j'} - s_j \geq T$.

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 when $\mathcal{S}$ encapsulates and sends the PBB, and may also 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 dropped or marked as invalid; i.e., even after compromising $\mathcal{S}$, $\mathcal{A}$ can't undetectably alter or re-order messages in C. *Stealth* means, informally, that $\mathcal{A}$ can't learn any information about messages buffered prior to compromise. It's stronger than confidentiality. Not only can't $\mathcal{A}$ learn the contents of messages, it also can't learn the number of buffered messages—or if any were buffered at all. This holds even after $\mathcal{A}$ has compromised $\mathcal{S}$.

4.3 Detailed construction

Our construction employs a *forward-secure pseudorandom number generator* FS-PRNG (see, e.g., [9]) that exports two operations GenKey and Next to compute the next pseudorandom numbers, as well as an *authenticated encryption scheme* (see, e.g., [2]) 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 are in Appendix B.

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

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*, seen as an array of size $T+1$, where $C[i]$, $0 \leq i \leq T$, denotes the ith position in C; we assume that each slot $C[i]$ is of fixed size s;
3. a *current index* I, initially pointing to a *random* position in C, and itself stored at $C[T]$.

We assume that $m \in \{0,1\}^\ell$ and also set $s = g(\ell)$.[12]

Key generation and evolution. 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, along with their sequence numbers. 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.

Write, transmit and read operations. 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 Algorithm 1.) 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 Algorithm 2.) 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 Algorithm 3.)

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.

[12]In our EAX encryption implementation: $\ell = 1004$ and $g(\ell) = 1024$.

Algorithm 1 Operation Write

```
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. $C[C[T]] = (\mathsf{AuthEnc}_{r_i}(m), i)$
  2. $C[T] = C[T] + 1 \bmod T$
  3. $(r_{i+1}, r'_j, i+1, j) \leftarrow \mathsf{KEvolve}(r_i, r'_j, i, j, 1, \texttt{low})$
  4. delete $r_i$
  5. return $[(r_{i+1}, r'_j, i+1, j), C]$
```

Algorithm 2 Operation Wrap

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

Algorithm 3 Operation Read

```
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. $(C[0], \ldots, C[T]) = C \leftarrow \mathsf{AuthDec}_{r'_{j'}}(c')$; $I = C[T]$
  4. if $C = \perp$ then return $[(r_i, r'_j, i, j), \perp]$
  5. 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) $m_k \leftarrow \mathsf{AuthDec}_{r_l}(c)$; $LAST = l$
  6. return $[(r_{l-T+1}, r'_{j'}, l-T+1, j'), (m_0, \ldots, m_{T-1})]$
```

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 Algorithms 1 and 3 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 transmission interval choice 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 can't 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.[13] 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 is the buffer adaptation policy in 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 same time, an adversary can't 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. EXPERIMENTAL EVALUATION

We developed a prototype of PillarBox in C++. To implement the authenticated encryption necessary in our system we utilize an open-source version of EAX-mode encryption. We also developed a custom FS-PRNG for generating the necessary encryption keys (for both the message encryption before addition to the buffer, as well as the re-encryption of the buffer before transmission).

We next experimentally validate the effectiveness of PillarBox in securing alerts during the critical window. We show that messages can be locked away within the PBB faster than they can be put on the network (showing their speed benefit over current systems), and we demonstrate 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.

5.1 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/receiver detection) in a rather simple setting.

Using the Ettercap tool [1] 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.

[13]Note that detecting network connection failures doesn't require PillarBox server message acknowledgements. E.g., an enterprise device can simply observe whether it is connected to the enterprise's network.

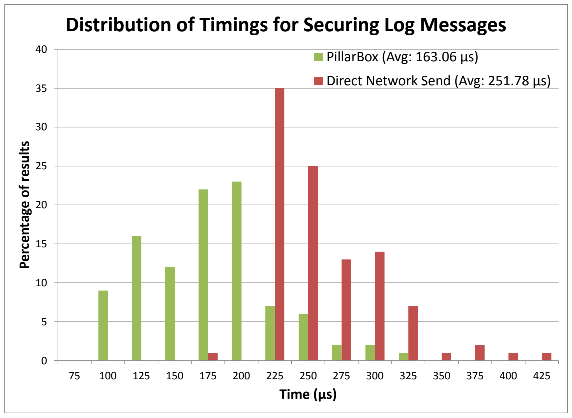

Figure 5: Distribution of timings for PillarBox cryptographically securing log messages locally vs. direct network transmission.

If executed within a subnet, the attack described here would in many settings be undetectable even by a forensic tool performing network packet capture, as these tools are typically deployed to monitor only traffic across subnets or in externally facing communications.

5.2 PillarBox vs. direct alert pushing

In our evaluation of PillarBox, we first show that messages can be secured locally in the PBB faster than they can be sent over the network. (the typical mode of today's SASs). We have already discussed some of the limitations of the on-the-wire approach, but additionally show here that PillarBox can secure alerts *faster* than such systems. To avoid the overhead of socket connections we use UDP and measure only the time necessary to send a packet, in comparison to the time needed to encrypt the same information and store it locally in the PBB. The networking stack is un-optimized, but so is the encryption used to secure alerts in PillarBox. Distributions over 100 tests are presented in Figure 5.

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.

5.3 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. Our tests were performed on an Intel Core 2 Duo T9300 processor running at 2.5GHz with 4 GB of memory and Ubuntu 12.04 as the operating system. In our implementation, 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).

A majority of attacks against hosts leverage some form of privilege escalation to gain root privilege. As SAS design is beyond the scope of our work, we don't discuss detection mechanisms for such privilege escalations. We first explore direct process killing by an attacker under the pessimistic assumption that it *already has root privileges*, i.e., doesn't need to mount a privilege escalation attack.

We then demonstrate as an example that for one common privilege escalation attack (the "Full Nelson" attack), detection is possible and PillarBox can secure an alert before the attacker can feasibly disrupt alert recording. 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 the PillarBox address space in memory) without root privilege is infeasible given current architectures (e.g., Address Space Randomization [20] and other techniques that enforce process separation).

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 the 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.

PillarBox is a simple application that is easily terminated by an attacker, although it can be run as root to provide some protection. Alerts must be generated, routed by syslog to the named pipe, and then picked up by PillarBox, encrypted and added to the buffer to be secured. 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 shut down or killed by invoking root privilege and issuing a kill command.[14] 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 by PillarBox.

As Table 1 shows, in the majority of runs the alert message is locked away in $\approx$ 4ms.[15] Alert messages are, on average, secured in PillarBox before it is killed with almost 3ms to spare.[16] 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.

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, nor 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 Intel Xeon E5506 Quad Core processor running at 2.13GHz 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. In the end we expect CPU scheduling to be at fault. Where possible, running PillarBox with higher priority may further lower the probability of an attacker winning the race condition.

[14]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.

[15]Unlike our previous experiments, these timings include alert generation, routing by syslog through the named pipe, and securing by PillarBox. As Fig. 5 shows, PillarBox accounts for a minuscule fraction of the total time.

[16]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. These results show the timings observed in those cases.

Table 2: Timeline of events for execution of sudo cp /etc/rsyslog.d/vanish.conf /home/vanish.copy.

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

We also considered attacks against the other components (syslog and the named pipe). We use a named pipe (created using the mkfifo command) to pass alerts from syslog to PillarBox. A named pipe is essentially 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 by closing the pipe pass through the pipe before it is closed. There were no failures in these tests.

The alerter (syslog) is the first to handle the log message, and can be shutdown or killed by running sudo kill -9 $<$ syslog_pid $>$.[17] Table 1 shows that the log message is created and 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, (there were no failures in these tests either), 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. This issue is diminished in the killing of the named pipe and PillarBox itself, explaining their perceived lower average timings (and standard deviations).

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 (contained in /etc/rsyslog.d/vanish.conf) generate an alert whenever a root user logs in (e.g., by using sudo) to modify any of the components necessary for PillarBox. 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 using the command sudo cp /etc/rsyslog.d/vanish.conf /home/vanish.copy. 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 copy takes place.

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 here is that as the attacker exploits vulnerabilities, it may not be possible to detect it running as root and killing processes.

As an example of a common privilege escalation, we consider the "Full Nelson" local privilege escalation, which exploits CVE-2010-4258, CVE-2010-3849, and CVE-2010-3850 to gain root access. We find that the "Full Nelson" attack generates kernel messages

[17]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 doesn't actually generate them itself.

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

	Message Secured	Std. Dev.	Component Disrupted	Std. Dev.	Command
Syslog (typical)	4.09ms	0.30ms	8.86ms	2.43ms	sudo kill -9 $<$ syslog_pid $>$
Syslog (bad scheduling)[11]	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 $>$

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. Any other events that occur prior to privilege escalation in the critical window that can be detected and logged are also likely to be secured by PillarBox.

5.4 Observed alerting frequencies

We performed an analysis of a large enterprise (>50,000 users) dataset gathered from an SIEM 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 certainly not be applicable to PillarBox.

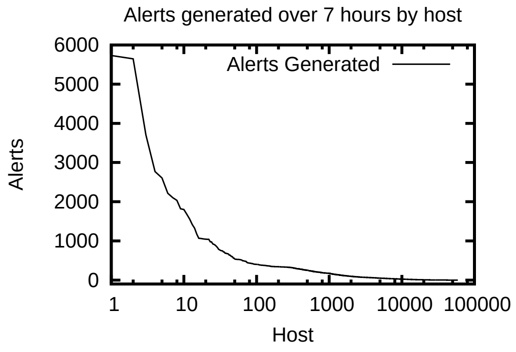

Figure 6: Alerts generated over a 7-hour window by host

Figure 6 shows the distribution of alerts coming from hosts within the enterprise reflected in our dataset. 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 seven-hour window, the busiest machine generated 8603 alerts, but the average across all machines (59,034 in total) was only 18.3 alerts in 7 hours. 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.

5.5 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.

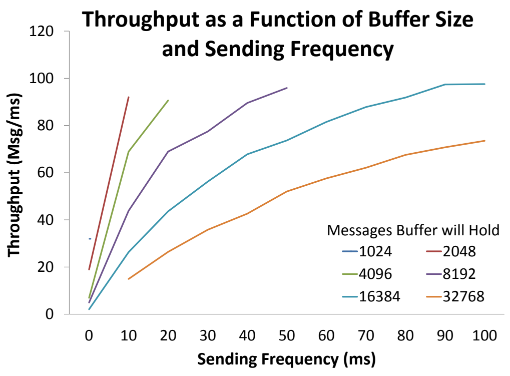

Figure 7: Host throughput vs. sized buffers, send rates

Figure 7 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 7 shows, throughput seems to top out just shy of 100 messages / ms, further evidence of the minimal overhead of PillarBox.

6. 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. A *hypervisor* (virtual machine monitor (VMM)), for instance, has higher privilege than a guest OS, isolating it from OS-level exploits. Thus, as an alternative to 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., [21], and/or is itself protected by trusted hardware, as in Terra [6]. Where available, a hypervisor-based approach is 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 [23, 10].

PillarBox builds in part on funkspiel schemes, introduced by Håstad et al. [8]. A funkspiel scheme creates a special host-to-server channel. This channel's existence may be known to an adversary, but an adversary can't tell if or when the channel has been used, a property similar to stealth in PillarBox. (By implication, an adversary can't recover message information from the channel either.) As in PillarBox, 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 isn't 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., [4, 3, 13, 22, 18, 24, 25, 12, 19, 15]). While these systems use forward-secure integrity protection like PillarBox, they aren't 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., [3, 18, 12, 19], are designed to "close" a log, i.e., create forward security for new events, only periodically, not continuously. Additionally, existing forward-secure logging systems don't aim, like PillarBox, to achieve stealth.

Finally, in a different context than ours, the Adeona system [17] 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 doesn't 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.

7. 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.

8. REFERENCES

[1] Ettercap. http://ettercap.sourceforge.net/.

[2] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *J. Cryptol.*, 21:469–491, 2008.

[3] M. Bellare and B. Yee. Forward-security in private-key cryptography. *CT-RSA*, pp. 1–18, 2003.

[4] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. *USENIX Sec.*, pp. 317–334, 2009.

[5] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[6] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. *SOSP*, pp. 193–206, 2003.

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

[8] J. Håstad, J. Jonsson, A. Juels, and M. Yung. Funkspiel schemes: an alternative to conventional tamper resistance. *CCS*, pp. 125–133, 2000.

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

[10] P. A. Karger. Securing virtual machine monitors: what is needed? *ASIACCS*, pp. 1:1–1:2, 2009.

[11] J. Kelsey, J. Callas, and A. Clemm. RFC 5848: Signed syslog messages. 2010.

[12] J. Kelsey and B. Schneier. Minimizing bandwidth for remote access to cryptographically protected audit logs. *RAID*, 1999.

[13] D. Ma and G. Tsudik. A new approach to secure logging. *Trans. Storage*, 5(1):2:1–2:21, March 2009.

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

[15] G. A. Marson and B. Poettering. Practical secure logging: Seekable sequential key generators. *ESORICS*, pp. 111–128, 2013.

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

[17] T. Ristenpart, G. Maganis, A. Krishnamurthy, and T. Kohno. Privacy-preserving location tracking of lost or stolen devices: Cryptographic techniques and replacing trusted third parties with DHTs. *USENIX Sec.*, pp. 275–290, 2008.

[18] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. *USENIX Sec.*, pp. 4–4, 1998.

[19] B. Schneier and J. Kelsey. Tamperproof audit logs as a

forensics tool for intrusion detection systems. *Comp. Networks and ISDN Systems*, 1999.

[20] H. Shacham, M. Page, B. Pfaff, E. J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. *CCS*, pp. 298–307, 2004.

[21] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM monitoring using hardware virtualization. *CCS*, pp. 477–487, 2009.

[22] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. *NDSS*, 2004.

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

[24] A. A. Yavuz and P. Ning. BAF: An efficient publicly verifiable secure audit logging scheme for distributed systems. *ACSAC*, pp. 219–228, 2009.

[25] A. A. Yavuz, P. Ning, and M. Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. *FC*, 2012.

APPENDIX

A. FORMAL DEFINITIONS

We provide formal security definitions for a *lockbox scheme*, a novel general-purpose crypto primitive for reliable, confidential and tamper-evident message transmissions.

Consider a *sender* $\mathcal{S}$ that communicates with a *receiver* $\mathcal{R}$ through a *channel* C that transmits messages from universe $U = L \cup \{\emptyset, \bot\}$ for a bounded-size language L ($|L| = 2^\ell$) and two special *null* ($\emptyset$) and *failure* ($\bot$) messages. Channel C is a *fixed-capacity memory buffer* which consists of T *slots*, each storing a message in U, and a special slot storing the *current index* I specifying the slot at which a new message written in the channel is to be stored. That is, $C = (C[0], C[1], \ldots, C[T])$ where $C[T] = I \in \{0, \ldots, T-1\}$.

Definition 1 (Lockbox scheme.) *A* lockbox scheme *LS comprises five PPT 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** KEvolve**:** *Key evolution algorithm* 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* 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* 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* KEvolve *for some $t \geq 0$, and returns a sequence $M = (m_1, \ldots, m_T)$ of T messages in $U = L \cup \{\emptyset, \bot\}$. We say that* Read produces *messages M and write $(\sigma_{r,j+t}, M) \leftarrow \mathsf{Read}(\sigma_{r,j}, \hat{C})$.*

Under normal operation, we expect a lockbox scheme to transmit messages *reliably* and in a *order-preserving* manner.

Definition 2 (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 $M = (m_{k-T+1}, \ldots, m_{k-1}, m_k)$.

We capture the two security properties of a lockbox scheme using two games played by a powerful adversary $\mathcal{A}$ which has access to a special oracle that makes use of the lockbox algorithms:

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 an adversary $\mathcal{A}$ that at a time of its choice *fully compromises* $\mathcal{S}$ to capture its secret state and get control over the buffer C. Prior to the compromise, $\mathcal{A}$ may *actively control* the transmissions between $\mathcal{S}$ and $\mathcal{R}$, by adaptively selecting the messages that $\mathcal{S}$ adds in C and when to encapsulate C to $\hat{C}$, and by arbitrarily modifying, deleting, injecting or reordering the set of encapsulated buffers produced by $\mathcal{S}$ before their delivery to $\mathcal{R}$.

Immutability captures an integrity property for the sequence of messages that are produced by $\mathcal{R}$: Any received non-empty message is either an invalid message that $\mathcal{A}$ has modified or deleted, or it's 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.* This holds even when $\mathcal{A}$ launches an adaptive chosen message attack prior to or after the compromise (similar to the standard notion of security for digital signatures [7]) and learns the secret state of $\mathcal{S}$ at the time of compromise.

Definition 3 (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* $\mu_1, \ldots, \mu_{l_1}$*,* $\mu_i = (m_1^i, \ldots, m_{z_i}^i)$*,* $i \in \{1, \ldots l_1\}$*, 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* $(m_1^{l_1+1}, C_1'), \ldots, (m_{k_2}^{l_1+1}, C_{k_2}')$ *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* $m = m_{i_1}^{j_1}$ *and* $m' = m_{i_2}^{j_2}$ *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:*[18]

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* $m^* \notin \{m_1^{l_1+1}, \ldots, m_{k_2}^{l_1+1}\}$*.*
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.*

Stealth captures a privacy property for the set of messages that are encoded and encapsulated by $\mathcal{S}$: Any encapsulated buffer satisfies ciphertext indistinguishability with respect to their contents, i.e., $\mathcal{A}$ cannot distinguish if a given encapsulated buffer contains one of two messages $\mathcal{A}$ selected, or whether it contains a given message selected by $\mathcal{A}$. This holds even when $\mathcal{A}$ learns the secret state of $\mathcal{S}$ at the time of compromise and launches a type of adaptive chosen ciphertext attack prior to or after the compromise (i.e., similar to IND-CCA2), where access to the decryption oracle is restricted to prevent trivial learning of $\mathcal{A}$'s challenge.

Definition 4 (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, reading the channel frequently enough, it should be possible to produce a given message more than one times.

Definition 5 (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.*

B. CRYPTOGRAPHIC PRIMITIVES

Authenticated encryption. This primitive provides combined confidentiality and integrity protection over transmitted messages.[19]

A symmetric-key authenticated encryption scheme consists of algorithms $\mathsf{AEKeyGen}$, $\mathsf{AuthEnc}$ and $\mathsf{AuthDec}$:

- $\mathsf{AEKeyGen}$ takes as input a security parameter and returns a secret key σ_{AE}, $\sigma_{AE} \leftarrow \mathsf{AEKeyGen}(1^\kappa)$.
- $\mathsf{AuthEnc}$ takes as input secret key σ_{AE} and message m and returns a ciphertext c, $c \leftarrow \mathsf{AuthEnc}_{\sigma_{AE}}(m)$.
- $\mathsf{AuthDec}$ takes as input secret key σ_{AE} and ciphertext c and either returns m or $\perp$, $\{m, \perp\} \leftarrow \mathsf{AuthDec}_{\sigma_{AE}}(c)$.

The security property satisfied by these algorithms combines message privacy and integrity [2]: If the ciphertext has not been tampered with decryption returns the originally encrypted message, whose confidentiality is protected by the ciphertext. As shown

[18] Protecting against these cases ensures that any received non-empty message is either an invalid message $\perp$ (tampered by $\mathcal{A}$) or a valid one in L that has been written in C after the most recently received message was written in C and has arrived while preserving its global order.

[19] 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).

in [2], 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.

We use a forward-secure variant of this primitive, where the secret key σ_{AE} evolves over time through a forward-secure pseudorandom number generator (FS-PRNG).

Forward-secure pseudorandom generator. This primitive supports a form of leakage-resilient cryptography by providing a sequence of strong cryptographic keys that achieve *forward security*: When a secret key evolves over time, being refreshed by the next key in the sequence, leaking the current key to an attacker doesn't affect the security of older keys.

The survey in [9] characterizes an FS-PRNG scheme in terms of a binary tree where the root is a random seed s and children are derived from their parent by application of a suitable one-way function. We use a generic description of such schemes that consist of algorithms GenKey and Next:

- GenKey takes as input a security parameter and returns an initial state s_0, $s_0 \leftarrow \mathsf{GenKey}(1^\kappa)$.
- 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}, $(r_{i+t-1}, s_{i+t}) \leftarrow \mathsf{Next}(s_i, t)$. For simplicity, we use $r_{i+t} \leftarrow \mathsf{Next}(r_i, t)$ and $r_{i+1} \leftarrow \mathsf{Next}(r_i)$ to generate the pseudorandom number that is $t > 1$ and $t = 1$ steps ahead respectively.

With respect to the security of an FS-PRNG scheme, it holds that given $\{s_{i+t}\}$, $t > 0$, it is hard to compute any older pseudorandom number r_j, $j \leq i$. For the common case where $t = 1$ above, we can consider sequence $\mathbf{r} = (r_0, r_1, r_2, \ldots)$ corresponding to a one-way hash chain.