PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging

Kevin D. Bowers[1], Catherine Hart[2,⋆], Ari Juels[3,⋆], and Nikos Triandopoulos[1]

[1] RSA Laboratories, Cambridge, USA
[2] Bell Canada, Vancouver, Canada
[3] Cornell Tech (Jacobs Institute), New York, USA

Abstract. *Security analytics* is a catchall term for vulnerability assessment and intrusion detection leveraging security logs from a wide array of *Security Analytics Sources (SASs)*, which include firewalls, VPNs, and endpoint instrumentation. Today, nearly all security analytics systems suffer from a lack of even basic data protections. An adversary can *eavesdrop* on SAS outputs and advanced malware can *undetectably suppress* or *tamper* with SAS messages to conceal attacks.

We introduce *PillarBox*, a tool that enforces *integrity* for SAS data even when such data is buffered on a compromised host within an adversarially controlled network. Additionally, PillarBox (optionally) offers *stealth*, concealing SAS data and potentially even alerting rules on a compromised host. Using data from a large enterprise and on-host performance measurements, we show experimentally that PillarBox has minimal overhead and is practical for real-world systems.

Keywords: Security analytics, forward-secure logging, log integrity and secrecy, self-protecting alerting, secure chain of custody.

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 [17]. To obtain data for such systems, organizations instrument a variety of hosts with a range of *Security Analytics Sources* (SASs) (pronounced "sass"). By SAS here, we mean generically a system that generates messages or alerts and transmits them to a *trusted server* for analysis and action.

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

⋆ Work performed while at RSA Laboratories.

A. Stavrou et al. (Eds.): RAID 2014, LNCS 8688, pp. 46–67, 2014.

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 Section 5.) And logged or buffered SAS messages are generally vulnerable to deletion or modification after host compromise.

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

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

PillarBox. As a solution to these challenges, we introduce a tool called *PillarBox*.[2] 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

[1] Many rootkits remove or obfuscate logs by modifying the binary of the logging facility itself.

[2] 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).

intercept them. *After* host compromise, PillarBox protects already generated SAS messages, even if an attacker can suppress new ones.

2. **Stealth:** Optionally, 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 is equally fast, more robust to message suppression, and important for stealth.

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

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

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

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

We emphasize that we do not address the design of SASs in this paper. 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 threat 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 an experimental evaluation of PillarBox. We review related work in Section 6 and conclude in Section 7. More technical details, which have been omitted from this version due to space constraints, can be found in the full version of this paper [4].

2 Modeling and Design Principles

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

2.1 Threat Model

Our threat model considers three entities, the *SAS* or the *host*, the *attacker*, and the *server*, which itself is a trusted entity, not vulnerable to attack. We model the attacker to be the strongest possible adversary, one attacking a host in the self-protecting setting. (Achieving security against this strong adversary ensures security against weaker ones, e.g., those attacking only the network or a firewall whose SAS only reports on network events.) Recall that in the self-protecting case, a SAS reports alerts about the host itself: While the compromise is taking place, the SAS generates one or more alert messages relevant to the ongoing attack and attempts to relay them to the server.

The adversary controls the network in the standard Dolev-Yao sense [6], i.e., the attacker 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, alert messages, etc.—and fully controls the host's future behavior, including its SAS activity.

To violate integrity, the attacker's goal is to compromise the host *without*: (1) any unmodified alerts reaching the server and (2) the server learning of any modification or suppression of alerts by the attacker. The SAS can only start generating meaningful alerts, of course, once the intrusion is in progress. After the attacker has achieved complete compromise, it can shut down the SAS or tamper with its outputs. So a SAS produces valid and trustworthy alerts only *after* intrusion initiation but *prior* to complete compromise. We call the intervening time interval the *critical window* of an attack, as illustrated in Figure 1. This is the interval of time when intrusions are detectable and alerts can be secured (e.g., buffered in PillarBox) before the attacker intercepts them.

Conceptually, and in our experiments, we assume that the attacker has full knowledge of the workings of the SAS, including any mechanisms protecting alerts en route to the server, e.g., PillarBox. It fully exploits this knowledge to suppress or modify alerts. The attacker doesn't, however, know host state, e.g., cryptographic keys, prior to complete

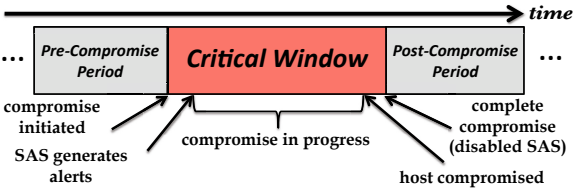

Fig. 1. Event timeline of host compromise

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, by making adaptive use of the network and of post-compromise host state, e.g., PillarBox's buffer. 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 privacy 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 element in our design is the use of a host-side *PillarBox buffer*, for brevity called the *PBB*, where alerts are secured. The objective 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.

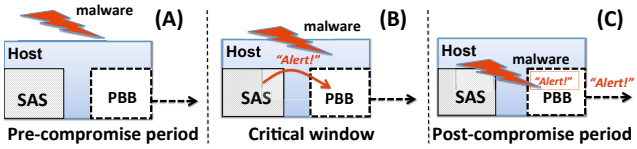

Fig. 2. PillarBox across compromise phases: (A) The host has not yet been attacked. (B) The SAS detects in-progress compromise and places an alert in PBB. (C) The host is under the attacker's full control, but PBB securely stores and transmits the alert.

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

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

– *Imperfect connectivity:* Many host SAS systems lack continuous connectivity to the server (e.g., laptops shuttling between an office and home have limited connection

with corporate security servers but are open to infection in the home). Lightweight embedded devices often cannot ensure or even verify delivery of transmitted messages (e.g., wireless sensor networks 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 (e.g., see CVE-2007-1531 and CVE-2010-2979), 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.)

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

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

Thus, *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. Such buffering, though, poses new security challenges. If an attacker completely compromises a host, there is no way of course to prevent it from disabling a SAS or tampering with its future outputs. But there is a separate problem after host compromise: Inadequately protected buffered SAS messages are vulnerable to modification/suppression and intelligence leakage. We next elaborate on how PillarBox solves these problems.

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

Indeed, a digital signature or message authentication code (MAC) alone, as proposed, e.g., for syslog [12], does not protect against tampering: After host compromise an attacker learns the signing key and can forge messages. Message encryption similarly does not protect messages against deletion, nor does tagging them with sequence numbers, as an attacker with control of a host can forge its own sequence numbers.

Fortunately, post-compromise alert integrity is achievable using the well-known cryptographic technique of *forward-secure* integrity protection. The main idea is to generate new (signing) keys on the host after every alert generation and delete keys immediately after use. This technique is commonly used for forward-secure logging (e.g., [14, 19, 25, 26]),[3] an application closely related to SAS protection. Similarly,

[3] Such systems are designed mainly for forensic purposes rather than detection (e.g., to protect against administrator tampering after the fact), thus they often "close" logs only periodically.

PillarBox uses forward-secure pseudorandom number generation (FS-PRNG) to create MAC keys. Each key is used to secure a single message and then deleted. An FS-PRNG has the property that past keys cannot be inferred from a current key, preventing tampering of messages that have already been secured. The server runs this FS-PRNG to compute the (same) shared keys with the host, allowing it to detect tampering or erasure.

What is new in the use of forward security in PillarBox is primarily its application for self-protecting alerting: Indeed, the main aspect of integrity here is securing alerts in the PBB as fast as possible during a compromise, i.e., in the critical window. Effectively, PillarBox engages in a race to secure alerts before the attacker intercepts them, and winning this race is not a matter of cryptography, but of system design, including the design choice of host-side buffering! An important contribution of our work is an experimental validation (in Section 5) that winning this race, and thus the whole PillarBox approach to securing alerts, is feasible. This is shown to hold even against a fast, local, PillarBox-aware attacker that kills PillarBox processes as quickly as possible.

Achieving Stealth. Stealth, as we define it, requires concealment of the entire alerting behavior of a SAS, including detection rules, alert message contents, alert generation times, and alert message existence in compromised hosts. Stealth is a key defense against sophisticated attackers. (One example: Host contact with "hot" IP addresses can help flag an APT, but an attacker that learns these addresses can just avoid them [15].)

Straightforward encryption alone does not achieve stealth: 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 or even perform black-box probing against a host to test attacks and infer which are or are not detectable. Instead, stealth in PillarBox requires a combination of several ideas.

In particular, PillarBox employs a buffer size T, and buffer transmission-time interval μ, that are *fixed*, i.e., invariant. Each message is also of fixed size (or padded to that size). When PillarBox transmits, it re-encrypts and sends the entire fixed-size buffer, not just fresh (encrypted) alerts. Such fixed-length transmissions prevent an attacker from determining when new alerts have accumulated in the host buffer, while its fixed communication patterns defeat traffic analysis. 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*.[4] This feature creates a need for careful parameterization: T must be large enough to hold all alert messages generated under benign conditions within a time interval μ; this condition ensures that if round-robin overwriting occurs, PillarBox implicitly signals to the server a "buffer-stuffing" attempt by an attacker.[5]

PillarBox generates encryption keys in a forward-secure way 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

[4] Thus messages may be transmitted multiple times. Such persistent transmission consumes bandwidth but it may allow temporarily suppressed messages to eventually reach the server.

[5] 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.

from buffer ciphertext changes.[6] Finally, as these two encryption layers ensure confidentiality in the buffer and over the network, but not in the SAS alerting engine itself, a key complement to stealth in PillarBox is concealment of detection rules in hosts: Our experiments show the viability of instrumenting the SAS with *vanishing rules*.

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

3 Architecture

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

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 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 system, which may be monitored by human operators. 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.[7] 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.

[6] Buffer encryption alone is insufficient: If identical buffer ciphertexts leave a host twice, the attacker learns that no new alert has been generated in between. Semantically secure public-key encryption would enable use of just one layer, but with impractically high cost overheads.

[7] Of course, a SAS includes other components, e.g., a transport layer, update functionality, etc.

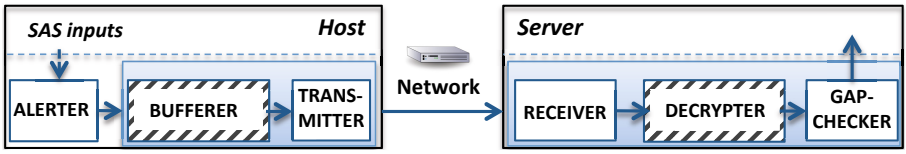

Fig. 3. PillarBox architecture and data flow. Shaded areas show the PillarBox components; striped ones comprise PillarBox's crypto-assisted core reliable channel.

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 relay regular logs through its ordinary low-priority transport layer.

3.2 PillarBox Components

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

Bufferer. This module controls the core message buffer, the PBB (which is detailed in Section 4). It accepts two calls: A Write call from the alerter to insert a message into the PBB (in encrypted form) and a Wrap call from the transmitter—described below—requesting export of the current buffer contents (also in a securely encapsulated form). This module is also responsible for maintaining the secret state of the PBB and updating the cryptographic (MAC and encryption) keys, which are effectively used to securely label messages and buffers with sequence numbers. The bufferer does not discard messages from the buffer when they are transmitted: A message is encapsulated and transmitted until overwritten, offering the extra feature of persistence.[8]

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 outputs either its constituent messages, or else a $\perp$ symbol indicating a buffer corruption. It also labels the buffer and its messages with their corresponding (verified) sequence numbers.

Gap-checker. The gap-checker's main task is to look for lost messages in the SAS message stream, which cause it to output an alert that we call a *gap alert*. These may be caused by one of two things: (1) A flood of alerts on the host (typically signalling an intrusion) or (2) Overwriting of alerts in the buffer by malicious buffer-stuffing on the

[8] This byproduct of stealth can be leveraged to accommodate lossy networks, as explained later.

compromised host (see also Section 4). As messages are labeled with verified sequence numbers, gap checking requires verification that no sequence numbers go missing in the message stream. Because messages continue to be transmitted until overwritten, note that in normal operation sequence numbers will generally *overlap* between buffers. The gap-checker can optionally filter out redundant messages. To detect an attacker that suppresses buffer transmission completely, the gap-checker also issues an alert if buffers have stopped arriving for an extended period of time, as we discuss below.

3.3 Parameterizing PillarBox

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

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

Also, if an attacker suppresses buffer transmission completely, the gap-checker will cease to receive buffers. The gap-checker issues a *transmission-failure alert* if more than β seconds have elapsed without the receipt of a buffer, for parameter setting $\beta > T/\tau$.

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

4 PillarBox Buffer and Protocols

We now present the main component of PillarBox, the PBB, and its protocols (run by the bufferer and decrypter), which realize a reliable messaging channel, as well as the functionality exported to the alerter and gap-checker to secure the SAS chain of custody.

Ideal "Lockbox" Security Model. Conceptually, the PBB serves as a "lockbox" for message transport: It's a buffer of T fixed-size slots that supports two basic operations:

1. `write`: The *sender* $\mathcal{S}$ (the client in PillarBox) inserts individual messages into the buffer via `write` in a *round-robin* fashion. Given currently *available* position $I \in \{0, \ldots, T-1\}$ (initially set at random), a new message is written in slot I (replacing the oldest message), and I is incremented by $1 \pmod T$.
2. `read`: The *receiver* $\mathcal{R}$ (the server in PillarBox) invokes `read`, which outputs the (monotonically increasing) sequence numbers j of the buffer and s_j of the last inserted message, along with the T messages in the buffer starting at position I, with wraparound.

Messages buffered in this ideal "lockbox" can only be read via the `read` interface and can only be modified (authentically) via the `write` interface. When read by $\mathcal{R}$, a message m_i stored at slot i is guaranteed to be either the most recent message written to slot i (the empty symbol $\emptyset$ if no message was ever written), or a special corruption symbol $\perp$ that indelibly replaces all the buffer's contents if the buffer was tampered with or 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 will cause decryption errors indicated by the special symbol $\perp$.
2. *Buffer overwriting:* The attacker can exploit buffer wraparound 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.[9]
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 is not 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 unlikely.

Adversarial buffer modification or destruction, as explained above, is an easily detectable attack. It causes the server to receive a symbol $\perp$, indicating a cryptographic integrity-check failure. The gap-checker in PillarBox detects both buffer overwriting attacks and buffer dropping attacks by the same means: It looks for lost messages, as indicated by a gap in message sequence numbers.[10] 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

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

[10] I.e., a gap alert is issued when the sequence numbers s_j and $s_{j'}$ of (of the last inserted messages in) two successively received buffers j and j' are such that $s_{j'} - s_j \geq T$.

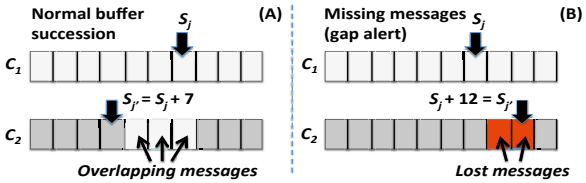

Fig. 4. Gap rule example on successively received buffers C_1, C_2, indexed by sequence numbers j, j' and $T = 10$: (A) Normal message overlap between buffers; (B) A detectable gap: Messages with sequence numbers $s_j + 1$ and $s_j + 2$ have been lost

when the server has received no buffers for an extended period of time, producing a transmission-failure alert, as noted above.

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

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

For our two cryptographic properties, we consider a powerful adaptive adversary $\mathcal{A}$ that operates in two phases: (1) Prior to compromise, $\mathcal{A}$ fully controls the network, and may arbitrarily modify, delete, inject, and re-order transmissions between $\mathcal{S}$ and $\mathcal{R}$; $\mathcal{A}$ may also determine 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 marked as invalid; i.e., even after compromising $\mathcal{S}$, $\mathcal{A}$ cannot undetectably drop, alter or re-order messages in C. *Stealth* means, informally, that $\mathcal{A}$ cannot learn any information about messages buffered prior to compromise. It is stronger than confidentiality. Not only cannot $\mathcal{A}$ learn the contents of messages, it also cannot learn the number of buffered messages—or if any were buffered at all. This holds even after $\mathcal{A}$ has compromised $\mathcal{S}$.

Detailed Construction. Our construction employs (and we assume basic familiarity with) a *forward-secure pseudorandom number generator* FS-PRNG (e.g., [10]) that exports two operations GenKey and Next to compute the next pseudorandom numbers, as well as an *authenticated encryption scheme* (e.g., [2]) that exports operations AEKeyGen, AuthEnc and AuthDec to encrypt messages m of size k to ciphertexts of size $g(k) \geq k$.

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` $[(r_{i+1}, r'_j, i+1, j), C]$

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` $[(r_i, r'_{j+1}, i, j+1), \hat{C}]$

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` $[(r_{l-T+1}, r'_{j'}, l-T+1, j'), (m_0, \ldots, m_{T-1})]$

Fig. 5. Operations Write, Wrap and Read

In particular, the sender $\mathcal{S}$ maintains the following data structure:

1. a *secret key* σ (also kept by the receiver $\mathcal{R}$);
2. a *buffer* C, $C = (C[0], C[1], \ldots, C[T-1])$, initially filled with *random data*, that takes the form of an array of size $T+1$, where $C[i]$, $0 \leq i \leq T$, denotes the ith position in C; we set the size of each slot $C[i]$ to be $s = g(\ell)$, where ℓ is an appropriate given message length (defining message space $\{0,1\}^\ell$).[11]
3. a *current index* I, initialized at a *random* position in C, and itself stored at $C[T]$.

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

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

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.

Then our main protocols operate as shown in Figure 5. 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. 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. Finally, given secret reading key (r_i, r'_j, i, j) and an encapsulated buffer $\hat{C}$, Read decrypts the buffer and all of its contents returning a set of T messages and updates the secret key.

For simplicity, we here consider a fixed-size PBB that holds fixed-size messages (parameters T and $g(\ell)$ respectively). Note that PillarBox can be easily 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; we omit these extensions due to space limitations.

5 Experimental Evaluation

We developed a prototype of PillarBox in C++. To implement authenticated encryption we utilize an open-source version of EAX-mode encryption. We also implemented a custom FS-PRNG as a hash chain for generating the necessary cryptographic keys (for both low- and high-layer secure processing of messages and buffers).

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

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.

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

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

5.2 Race-Condition Experiments

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

To bias our experiments in favor of an attacker, we assume the attacker has gained access to a privileged account that already has the necessary permissions to kill any of the components. We record time required for the attacker to issue a *single command* to kill the process and show that the command itself gets secured by PillarBox before the targeted component is terminated. Our tests were performed on an 2.5GHz Intel Core 2 Duo T9300 processor with 4 GB of memory and Ubuntu 12.04 as the operating system.

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

As Table 1 shows, in the majority of runs the alert message is locked away in $\approx$ 4ms.[13] Alert messages are, on average, secured in PillarBox before it is killed with

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

[13] PillarBox accounts for only a minuscule fraction of this total time.

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

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

almost 3ms to spare.[14] 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.

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

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

Killing Syslog. The alerter (syslog) is the first to handle the log message, and can be shutdown or killed by running sudo kill -9 $<$ syslog_pid $>$.[15] Table 1 shows that the log message is sent by syslog before it is killed. However, presumably due to process scheduling, in several runs the kill command returns before the alert message is secured

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

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

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

in the PBB. Because the message *always* arrives in the PBB (again, there were no failures), we assume these represent runs where the alert is passed to the named pipe before syslog terminates and then read from the pipe when the PillarBox process is later scheduled by the OS. This issue is diminished in the tests against the named pipe and PillarBox, explaining their perceived lower average timings (and standard deviations).

Vanishing Rules. When PillarBox provides stealth, it is best combined with vanishing SAS rules to prevent critical information leakage. Recall that if an attacker cannot prevent PillarBox from securing events in the critical window, the attacker benefits from at least learning how the system is instrumented and what alerts were likely to have been generated. In our test setup, the vanishing alerts generate an alert whenever a root user logs in. To test the race condition, we instrumented PillarBox to delete the vanishing alerts configuration file after securing the alert message. The attacker attempts to create a copy of the sensitive alerter configuration file. As it is shown by the relative timing of events over 100 test runs in Table 2, after securing the alert message, PillarBox always successfully deletes the configuration file at least 2.72 ms. before the attempted copy.

Privilege Escalation. Having shown that PillarBox can win the race conditions related to securing alerts and causing them to vanish, even in the pessimistic case where the attacker *starts with the necessary permissions*, we now consider the issue of privilege escalation. The concern is that if the attacker exploits vulnerabilities the transition to root privilege may not get logged. We assume that most privilege escalations could be detected given the proper instrumentation and that disrupting any of the necessary components in our system (e.g. corrupting its memory address space) without root privilege is infeasible given current architectures (e.g., Address Space Randomization [21], etc.).

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

While the "Full Nelson" attack is representative of other local privilege escalation attacks, this by no means guarantees that faster or quieter privilege escalations don't exist. What it does demonstrate is that the event signaling the end of the critical window

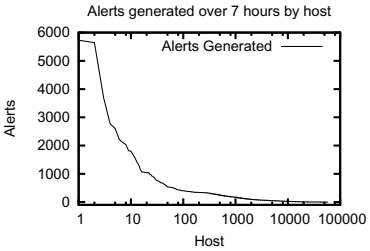

Fig. 6. Host-generated alerts over 7h

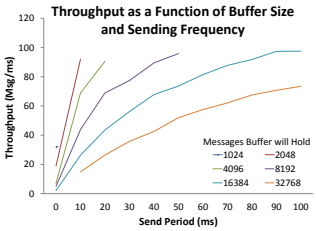

Fig. 7. Throughput vs. $|\text{PBB}|$, μ

(the elevation of privilege giving the attacker full control) can itself often be detected and secured in PillarBox before such privilege enables disruption of the PillarBox tool.

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

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

Defender Event	Attack detected	Log secured	Rule deleted	
Average	1,645.441ms	1,645.609ms	1,645.772ms	
Std. Dev.	1,069.843ms	1,069.842ms	1,069.840ms	
Attacker Event	**Remote shell**	**Privilege escalation**	**Rules copied**	**Log file deleted**
Average	2,692.536ms	2,693.474ms	2,696.524ms	2,696.590ms
Std. Dev.	1,324.419ms	1,324.432ms	1,324.919ms	1,324.990ms

5.3 Observed Alerting Frequencies

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

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

Figure 6 shows the distribution of alerts coming from hosts within the enterprise. The x-axis is in log scale, showing that the majority of machines send very few alert messages, while a small subset send the majority. Over a 7-hour window, the busiest machine generated 8603 alerts, but the average across all machines (59,034 in total) was only 18.3 alerts. Clearly, therefore, if we design the system to handle a throughput of one alert per second (3600 alerts an hour) our system will be able to handle even the busiest of alerters. The maximum observed rate in our dataset was 1707 alerts / hour.

5.4 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 μ with which the buffer is re-encrypted and sent. Obviously the larger the messages, the longer they take to encrypt. The standard log messages generated on our Linux system were typically a few hundred characters long. We note that our hash-chain FS-PRNG required one computation per produced number, thus minimizing key-generation overhead.

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*, or 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., [22], and/or is itself protected by trusted hardware, as in Terra [7]. 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 are not virtualized and some, e.g., embedded devices, probably will not be for a long time. Operating constraints often limit security administrators' 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 VMM 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 [24, 11].

PillarBox builds in part on funkspiel schemes, introduced by Håstad et al. [9]. A funkspiel scheme creates a special host-to-server channel whose existence may be known to an adversary; but an adversary cannot tell if or when the channel has been used, a property similar to stealth in PillarBox. (By implication, an adversary cannot recover message information from the channel either.) As in our work, a funkspiel scheme resists adversaries that see all traffic on the channel and ultimately corrupt the sender.

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

Another closely related technique is forward-secure logging (also called tamper-evident logging), which protects the integrity of log messages on a host after compromise by an adversary (see, e.g., [5, 3, 14, 23, 19, 25, 26, 13, 20, 16]). As already discussed, while these systems use forward-secure integrity protection like PillarBox, they are not designed for self-protecting settings like PillarBox. They aim instead for forensic protection, e.g., to protect against retroactive log modification by an administrator. Some schemes, e.g., [3, 19, 13, 20], are designed to "close" a log, i.e., create forward security for new events, only periodically, not continuously. Additionally, existing forward-secure logging systems do not aim, like PillarBox, to achieve stealth.

Finally, in a different context than ours, the Adeona system [18] uses forward-secure host-side buffering in order to achieve privacy-preserving location tracking of lost or stolen devices. Adeona uses cryptographic techniques much like those in PillarBox to cache and periodically upload location information to a peer-to-peer network. Adeona does not offer integrity protection like PillarBox, 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.

Acknowledgments. We thank the anonymous reviewers for their helpful comments, and also Alina Oprea, Ting-Fang Yen and Todd S. Leetham for many useful discussions.

References

[1] Ettercap, `http://ettercap.sourceforge.net/`
[2] Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. J. Cryptol. 21, 469–491 (2008)
[3] Bellare, M., Yee, B.: Forward-security in private-key cryptography. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 1–18. Springer, Heidelberg (2003)
[4] Bowers, K.D., Hart, C., Juels, A., Triandopoulos, N.: PillarBox: Combating next-generation malware with fast forward-secure logging. Cryptology ePrint Archive, Report 2013/625 (2013)
[5] Crosby, S.A., Wallach, D.S.: Efficient data structures for tamper-evident logging. In: USENIX Sec., pp. 317–334 (2009)
[6] Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Trans. on Inf. Theory 29(2), 198–207 (1983)
[7] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. In: SOSP, pp. 193–206 (2003)
[8] Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM J. Comput. 17(2), 281–308 (1988)
[9] Håstad, J., Jonsson, J., Juels, A., Yung, M.: Funkspiel schemes: An alternative to conventional tamper resistance. In: CCS, pp. 125–133 (2000)
[10] Itkis, G.: Handbook of Inf. Security, Forward Security: Adaptive Cryptography—Time Evolution. John Wiley & Sons (2006)
[11] Karger, P.A.: Securing virtual machine monitors: what is needed? In: ASIACCS, pp. 1–2 (2009)
[12] Kelsey, J., Callas, J., Clemm, A.: RFC 5848: Signed syslog messages (2010)
[13] Kelsey, J., Schneier, B.: Minimizing bandwidth for remote access to cryptographically protected audit logs. In: RAID, p. 9 (1999)
[14] Ma, D., Tsudik, G.: A new approach to secure logging. Trans. Storage 5(1), 2:1–2:21 (2009)
[15] Mandiant. M-trends: The advanced persistent threat (2010), `http://www.mandiant.com`
[16] Marson, G.A., Poettering, B.: Practical secure logging: Seekable sequential key generators. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 111–128. Springer, Heidelberg (2013)
[17] Oltsik, J.: Defining big data security analytics. Networkworld, 1 (April 2013)
[18] Ristenpart, T., Maganis, G., Krishnamurthy, A., Kohno, T.: Privacy-preserving location tracking of lost or stolen devices: Cryptographic techniques and replacing trusted third parties with DHTs. In: USENIX Sec., pp. 275–290 (2008)
[19] Schneier, B., Kelsey, J.: Cryptographic support for secure logs on untrusted machines. In: USENIX Sec., p. 4 (1998)
[20] Schneier, B., Kelsey, J.: Tamperproof audit logs as a forensics tool for intrusion detection systems. Comp. Networks and ISDN Systems (1999)
[21] Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the Effectiveness of Address-Space Randomization. In: CCS, pp. 298–307 (2004)
[22] Sharif, M.I., Lee, W., Cui, W., Lanzi, A.: Secure in-VM monitoring using hardware virtualization. In: CCS, pp. 477–487 (2009)

[23] Waters, B.R., Balfanz, D., Durfee, G., Smetters, D.K.: Building an encrypted and searchable audit log. In: NDSS (2004)
[24] Chen, Y., Chen, Y., Paxson, V., Katz, R.: What's new about cloud computing security? Technical Report UCB/EECS-2010-5, UC Berkeley (2010)
[25] Yavuz, A.A., Ning, P.: BAF: An efficient publicly verifiable secure audit logging scheme for distributed systems. In: ACSAC, pp. 219–228 (2009)
[26] Yavuz, A.A., Ning, P., Reiter, M.K.: Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 148–163. Springer, Heidelberg (2012)