SGX-Log: Securing System Logs With SGX

Vishal Karande, Erick Bauman, Zhiqiang Lin, Latifur Khan
Department of Computer Science, The University of Texas at Dallas
800 W. Campbell Rd, Richardson, TX, 75080
{vishal.karande, erick.bauman, zhiqiang.lin, lkhan}@utdallas.edu

ABSTRACT

System logs are the greatest forensics assets that capture how an operating system or a program behaves. System logs are often the next immediate attack target once a system is compromised, and it is thus paramount to protect them. This paper introduces SGX-Log, a new logging system that ensures the integrity and confidentiality of log data. The key idea is to redesign a logging system by leveraging a recent hardware extension, called Intel SGX, which provides a secure enclave with sealing and unsealing primitives to protect program code and data in both memory and disk from being modified in an unauthorized manner even from high privilege code. We have implemented SGX-Log atop the recent Ubuntu 14.04 for secure logging using real SGX hardware. Our evaluation shows that SGX-Log introduces no observable performance overhead to the programs that generate the log requests, and it also imposes very small overhead to the log daemons.

CCS Concepts

•Security and privacy → Trusted computing; Software security engineering;

Keywords

Trusted hardware; SGX; Application security; Secure logging; Log attacks; Logging Protocols

1. INTRODUCTION

Protecting system logs has always been one of the most security-critical tasks in a computer system. System logs record abundant status information about software execution including system events, configuration changes, and resource usages. Because of their forensic value, system logs are naturally the next immediate attack target for an experienced attacker when a system is compromised, since system logs have recorded all of the footprints of the attacker. To make system logs secure, we must maintain their integrity and ensure log files are not modified in an unauthorized manner.

To guarantee the integrity of system log entries, existing approaches either leverage special write-only hard disks or archive logs outside the local system, ideally on multiple remote machines. Unfortunately, it is practically very expensive to use write-only disks, since logging is a contiguous process which usually generates a large volume of data. Isolating the log from the service machine (e.g., sending the log data to a remote machine), using distributed logs, or logging at the virtual machine monitor layer (e.g., [17]), are all viable approaches. However, logs generated on multiple remote machines, including at the hypervisor layer, may fail to provide *consistency*, *availability* and *partition tolerance* simultaneously [19]. Also, additional efforts are required to protect the isolated logging machines.

Alternatively, logging can also be secured using sophisticated cryptographic solutions. State-of-the-art cryptographic solutions [10, 11, 36, 37] build special protocols assuming logs are generated and stored on a local logging server. In particular, Bellare and Yee [10, 11] introduced a protocol using forward-integrity, and Schneier and Kelsey [36, 37] proposed a technique based on forward-secure MACs and a one-way hash chain. Unfortunately, all these cryptographic approaches fail to maintain the secrecy of the current key stored in main memory. Additionally, power failure attacks result in the loss of the current log messages and affect the continuity of hash key chain generation. To reduce the impact of a compromise, these approaches require frequent key updates by making a trade-off between security and performance. Delaying the key change for better performance allows an attacker to compromise the logs protected using the current key.

Using trusted hardware for secure logging has become an appealing approach since the hardware is able to protect the data and the code operating on it [28, 40]. Interestingly, trusted execution environments (TEE) combined with cryptographic solutions can provide efficient and secure computing environments for log generation, integrity checks, and analysis. Motivated by the recent advances in TEE, we propose using Intel SGX (Software Guard eXtensions) as a hardware solution for securing system logs.

In particular, SGX allows user level code to allocate private regions of memory called enclaves, which are protected from software running even at higher privilege level such as OS kernels and hypervisors. Enclave data is only accessible by the code running inside the enclave. To protect data outside of an enclave, SGX supports sealing and unsealing features through a cryptographic library. SGX's remote attestation allows a remote party (e.g., a log generating machine) to verify the identity of the SGX machine (e.g., the log-server). In addition, unlike other trusted hardware [12], SGX provides some flexibility in customizing the size of the enclave cache that stores sensitive information.

As such, in this paper we propose SGX-Log, a practical system using SGX to ensure the integrity and confidentiality of system logs. SGX-Log uses a client and server architecture. The

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates

DOI: http://dx.doi.org/10.1145/3052973.3053034

client is a log request component, which issues various log messages, while the log server executes the secure logging services. Prior work in [31] first proposes to use SGX for securing medical device information in the cloud. It details the use of SGX, Trusted Platform Module (TPM) and standard encryption for securing logs against replay attacks, eavesdropping attacks and injection attacks. Our work differs from this work by providing a detailed design, implementation, and evaluation of an SGX-enabled log server. The detailed design of a logging system has a large impact on both its security and performance properties. For example, the design determines the size of the Trusted Computing Base (TCB) and size of its interface to untrusted code. In addition, due to the limited memory available to SGX enclaves and the cost of interaction with untrusted code, a logging system must be designed to accommodate for this. Therefore, SGX-Log is a more detailed investigation into practical protection of logs with SGX that provides concrete results and insight into how to effectively use SGX for log protection.

Since the TCB directly relates to the logging system's attack surface, we use a minimal TCB approach by executing only core logging services inside SGX. We also design efficient logging protocols by proposing the use of a block-level hash key chain and detail the security of the system against a variety of attacks, such as power failure attacks, log truncation, rollback attacks, etc. Interestingly, not all logs can be efficiently stored inside an enclave due to the limited size of the enclave page cache (EPC). To solve this, we propose to protect logs outside of the enclave using SGX's sealing feature. For performance improvements, we first buffer the processed logs inside an enclave and then perform block level sealing. Upon reading the sealed messages, we first unseal them inside the enclave and then generate message authentication codes (MACs) for verification.

Contributions. The main contributions of this paper can be summarized as follows:

- We provide a detailed design of an SGX-enabled log server to protect the integrity and confidentiality of system logs. Particularly, we minimize the TCB size to limit the attack surface and reduce overhead from SGX.
- We have implemented SGX-Log by following the standard logging system protocol in Linux, `syslog`, to completely protect the log data both inside and outside enclaves by using the sealing and unsealing primitives provided by SGX.
- For efficient log verification, we have implemented a block-level hash key chain, which enables hierarchical traversal of key chains and thus yields better performance.
- We have evaluated the performance overhead of the proposed solution in log generation, reading and integrity checking, with various benchmark programs. At lower level, we also evaluated the overhead incurred by SGX's trusted APIs and enclave boundary data transfer operations. The evaluation shows that SGX-Log imposes no overhead on the programs that generate the log requests, and also causes very small overhead in the log daemons.

2. BACKGROUND

2.1 Software Guard Extensions (SGX)

At a high level, Intel SGX is a set of extensions to the Intel x86 architecture that allows trusted part of an application to be executed in a secure container called an enclave [14]. The trusted hardware establishes an enclave to protect the integrity and confidentiality of private data in a computation and the code that operates on it. To achieve this, the CPU protects an isolated memory region called Processor Reserved Memory (PRM) against other non-enclave memory accesses, including the kernel, hypervisor and other privileged code. Sensitive code and data is encrypted and stored as 4KB pages in the Enclave Page Cache (EPC), a special region inside the PRM. Although EPC pages are allocated and mapped to frames by the OS kernel, page encryption assures confidentiality and integrity. Moreover, to provide access protection to the EPC pages, the CPU maintains an Enclave Page Cache Map (EPCM) that stores security attributes associated with EPC pages.

SGX instructions and enclave life cycle. To manage the execution of trusted code inside enclaves, Intel introduces a set of SGX instructions. Privileged software such as the OS manages the EPC using SGX-supported ring-0 instructions such as `ECREATE`, `EADD` and `EINIT`. While in user-space, enclave functionality is carried out using ring-3 instructions such as `EENTER`, `EEXIT`, `EGETKEY`, `ERESUME`, and `EREPORT`. In an enclave's life cycle, an enclave is first created using `ECREATE`, which sets base address and range addresses. Next, initial code and data is loaded into the enclave by untrusted software using the `EADD` instruction. While loading, a cryptographic hash of the contents is computed using `EEXTEND` and is finalized as the enclave's measurement hash once the enclave is initialized using `EINIT`. After initialization, enclave code is executed by the CPU in a special mode in which any code (including privileged system software) outside the enclave cannot inspect or tamper with the enclave's execution. Additionally, special control instructions, i.e., `EENTER` and `EEXIT`, prevent external software from using a cached address to access the enclave's private memory during enclave entry and exit operations.

Sealing and Unsealing. Intel SGX enclaves protect secrets while they are within the boundary of an enclave and are lost when the enclave is closed. This means that secrets need to be preserved when an enclave is destroyed, both as part of its ordinary lifecycle and unexpected termination. SGX provides data sealing and unsealing functions to protect secrets outside the boundary of an enclave. To handle this, sealing and unsealing functions retrieve keys unique to the enclave platform. When retrieving a key, first a key request structure is created using `EREPORT` information. Then `EGETKEY` is called with the key request structure to obtain the secret key. The enclave platform uses this key to encrypt data to the platform (i.e., sealing), or to decrypt data already on the platform (i.e., unsealing).

Local and remote attestation. Before performing computation on a remote platform, a user should be able to verify the authenticity of the trusted environment. Attestation is the mechanism by which a third party establishes that desired software is running on an Intel SGX enabled platform and within an enclave. During attestation using SGX, an application that hosts an enclave asks the enclave to produce a report called a quote to identify the platform. A quote containing information about the measurement of code and data, the product ID, security version number and other attributes is securely presented to the remote service provider for identity verification. This quote does not include the hardware key. SGX supports both local and remote attestation for software running on local and remote machines to verify the platform.

2.2 Standard System Logging Architecture

System logging is the process of recording events that occur in an OS or an application. Modern operating systems have a default `syslog` component. Linux/Unix `syslog` is a standard (RFC-5424) [18] for message logging that enables separation of the soft-

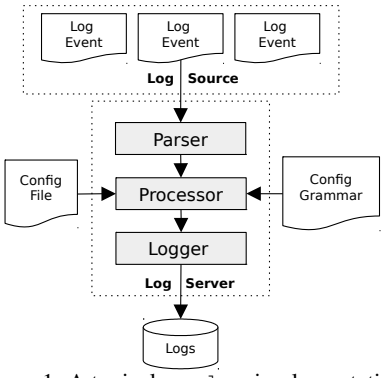

Figure 1: A typical `syslog` implementation.

ware that generates messages, the storage system for those messages, and message reporting and analysis software.

The `syslog` standard uses a client and server architecture. Clients that generate the log messages are called log-sources, and the target system responsible for collecting, processing and recording log messages is referred to as the log-server. The client and the server can use standard inter-process communication (IPC) such as a message queue or network communication based on UDP or TCP/IP.

Figure 1 illustrates a concrete implementation of `syslog`, especially the log-server, which consists of log configuration file(s), log configuration grammar, log-server daemon and the actual log files. In particular, the `config` file consists of information about the source and destination of event messages along with filter rules for processing event messages. The `config grammar` is used to resolve the rules or filters specified in `config` files. The log-server daemon is a continuously listening process for recording the events from log-sources. It has three main modules:

1. **Parser** that tokenizes the event information such as facility (program that generates an event), severity level, message contents, etc., using the standard message format.

2. **Processor** that resolves the config file using a config grammar file and applies user specified filters or rules. It provides the destination file name and message to the Logger.

3. **Logger** that is responsible for recording logs on persistent storage using a log rotation policy.

Recording an event information on the same system it is generated can be problematic because an attacker can access, erase or modify log files when a system is compromised; using a standard for logging such as `syslog` makes it simpler for an attacker to modify the logs. Therefore, it is necessary to secure the log server daemon in order to make tampering with logs more difficult.

3. OVERVIEW

In this section, we provide an overview of SGX-Log. We first describe our security objectives in §3.1, then our assumptions and threat model in §3.2. Next, we describe why we should use SGX for log protection in §3.3, and finally discuss the challenges we encountered and the proposed solutions to address them in §3.4.

3.1 Security Objectives

The primary objective of designing SGX-Log is to ensure the integrity of system logs with SGX: unauthorized users should not be able to modify any sensitive logs stored by log-server. Additionally, we want to keep system logs secret from unauthorized users regardless of whether the logs are inside or outside an enclave.

We also seek to maintain the integrity of the logging code responsible for log generation, log reading and log integrity checking against unauthorized tampering. This is necessary because this code is responsible for handling the sensitive data. If this code is compromised, then attackers can access sensitive log data and get away with unauthorized tampering.

3.2 Scope, Assumptions and Threat Model

Scope and Assumptions. We focus on securing logging systems that use a client/server communication model. Log-sources can run on the same machine as log-server or on multiple different machines connected using a communication network. While using log-sources from different machines, communication between client and server should be secure. For simplicity, we assume both log-server and log-sources run on the same machine. We also assume that log-sources send only valid messages to log-server.

Threat model. We consider two types of attackers: (1) local insiders and (2) remote attackers. Local insiders can control processes on the same machine as the logging process. They have physical access to the logging system, specifically the memory controller or buses. The less powerful remote attackers can attack log-server using a network connection. Both remote attackers and insiders can use software vulnerabilities to compromise log-server.

Various physical attacks (e.g., DRAM bus tapping) can be mitigated using SGX hardware features. Both remote attackers and local insiders cannot read sealed messages without access to the SGX enclave they came from. Any tampering of log messages by an attacker before gaining root privileges can be logged and replicated on a remote machine to avoid deletion. For instance, if a remote attacker aims to compromise the `httpd` daemon process, which usually logs each network connection and remote user's activities, then all of the attacker's footprints right before they compromise the system will be securely logged by SGX-Log.

Regarding the log deletion attack, if an attacker has obtained root privilege, he or she may be able to delete all of the log files. We note that unless a write-only hard disk is used, it is very challenging to defeat such attacks. While using expensive write-only drives can defend against a log deletion attack, SGX-Log replicates log messages on multiple remote machines to make the attacker's life difficult. Note that all existing cryptographic approaches face this issue as well. The purpose of building SGX-Log is to improve the practicality of existing cryptographic approaches by using the compelling features from SGX as discussed below.

3.3 Why SGX

Given our threat model and assumptions, it may appear that many of the existing approaches (e.g., those from cryptographic perspectives) can defeat many of the attacks defeated by SGX-Log. However, we would like to emphasize that existing log protection approaches fail to provide as secure and cost efficient solutions as one can with SGX. More specifically:

- *Cryptographic solutions* using hash key chains fail to protect the recent log entries when the current hash key is compromised. As a result, they require efficient key updates and key protection. They can not maintain the continuity and secrecy of the key chain across a power cycle without hardware support. Moreover, these solutions suffer from 'delayed detection' since they rely on trusted hardware for verification.

- *Tamper resistant software solutions* require code obfuscation and are prone to introduce software vulnerabilities.

- *Tamper resistant hardware solutions* store only the most sensitive data due to high cost and small storage capability.

However, SGX provides enclaves for secure computing, sealing/unsealing for data protection, and remote attestation for authenticity. SGX-Log protects logging services, hash keys and log entries inside the secure memory of SGX enclaves. To protect sensitive data outside of an enclave, SGX provides hardware-supported sealing and unsealing features. Sealing log entries to local commodity hard drives and replicating them on remote machines is a cost efficient solution to discourage log deletion attacks.

For secure communication between multiple enclaves running on the machine, SGX supports local attestation. For example, local attestation enables the collection of an SGX enclave's logs into a separate SGX-based logger running on the same machine. Additionally, logging can be scaled to support a large number of log clients by running multiple enclaves in parallel. Since the performance of enclave applications running on a single system is limited by the EPC size as detailed in [21], we can run multiple instances, each running on a different SGX-enabled machine. SGX supports remote attestation to verify the remote platform as well, further detailed in §8. As a result, SGX provides a secure and cost efficient solution for logging systems.

3.4 Challenges and Solutions

While the idea of using SGX to protect system logs is simple, it is actually non-trivial to implement SGX-Log. Below, we discuss the challenges we encountered and describe how we addressed them.

C1: Logging system partitioning. While building an SGX based secure logger, application developers first need to partition the logging system into trusted components and untrusted components and define the interface for communication between them. Trusted components run inside the enclave while untrusted components run outside of the enclave. Although multiple partitioning schemes exist in practice as detailed in [6], it is still unknown *which* partitioning scheme should be used.

Solution Atamli-Reineh and Martin [6] propose four partitioning schemes: (1) *whole application*, (2) *all secrets*, (3) *separate secrets*, and (4) *hybrid*. The *whole application* scheme suggests running entire logging applications within an enclave. However, as SGX does not support system calls inside enclaves (and therefore also does not support file I/O), we cannot put entire logging applications inside enclaves. In addition, although there are Library OSes that can run inside SGX [8] and therefore support running entire applications inside SGX, such an approach forces a large TCB and poor control over how data is sent across the enclave boundary. The *all secret* scheme proposes to protect only the secrets inside the enclave using a small sized TCB. However, this small sized TCB may not be sufficient to run code in an isolated environment. The *separate secrets* scheme uses multiple enclaves, one for each secret. A logging system needs to protect only the initial key required for hash key chain generation. Thus this approach is not required.

We chose to use their *hybrid* scheme, which allows for more flexible partitioning. When designing SGX-Log, we partitioned the logging system into two components: a trusted and an untrusted component. Only the initial key used for hash key chain generation is a secret, and the code that uses this key naturally becomes trusted. We also need the trusted component to protect the integrity of the logging data. All other code, which executes privileged system calls accessible only outside the secure environment, needs to be in the untrusted component.

C2: Protecting high volume of system logs. In a standard logging system, log-server collects event information from multiple sources. This leads to the problem of protecting a high volume of sensitive system logs. System logs stored in enclave memory are protected by the hardware. However, storing all system logs inside the enclave is not a feasible solution.

Solution To solve this challenge, we propose using SGX's sealing and unsealing features to store log entries securely outside of the enclave. APIs for sealing and unsealing generate keys using the hardware key unique to each SGX platform. Sealing generates a unique seal key for each input data. Thus, it is protected against key wear-out attacks. Sealing stores the key identifier along with the encrypted data blob. Unsealing uses the SGX hardware key and the key identifier for decryption. Thus, an attacker can not read log messages present both inside and outside of the enclave.

C3: Verifying arbitrary subset of logs. In log verification, logging systems based on a hash key chain [31,36,37] require sequential traversal of the hash chain. As a result, verifying an arbitrary subset of logs for modular enforcement of log policies is not efficient.

Solution In SGX-Log, we address this typical logging performance challenge by designing a new efficient block-level hash key chain. SGX-Log divides incoming log messages into a set of blocks, each containing a fixed number of messages. For each block we append a new block key to the block key chain. This block key is then used to generate the hash keys for messages in that particular block. We implement this approach as a two-dimensional block hash key chain by modifying the branched key chain approach detailed in [40]. In efficient verification, we first traverse the block key chain to retrieve the keys for that blocks which contain the messages to be verified. Then, we use these block keys to generate the specific message keys.

C4: Streaming log messages. The goal of log-server is to record all logs received from registered log-sources. Logging is a contiguous process that records real time events; if log-sources overwhelm log-server's capacity, information could be lost.

Solution To solve this typical streaming challenge, we propose using a ring buffer for storing logs. The ring buffer ensures that incoming messages are not lost until it is not full. As soon as the ring buffer contains the number of logs equal to the log block size, the buffer is sealed to the platform. Buffering logs provides two advantages. First, block-sealing reduces individual sealing and unsealing overheads. Second, buffered logs represent the current state of execution and do not require unsealing in serving read requests.

C5: Enclave shutdown detection. Unintended enclave shutdowns during a power failure attack affect the continuity of the hash key chain and may result in a loss of data.

Solution We propose to maintain a sealed flag on the disk to detect unintended shutdowns. We set the flag on an intentional shutdown after all logs in the enclave have been successfully sealed. When the enclave restarts, we retrieve the sealed flag and detect if the enclave was shut down as intended or not. Further, we ensure hash chain continuity by sealing the next block key when a new block is initialized. The sealed key is retrieved upon enclave restart and hash chain generation is resumed. We detail protection against rollback attacks using sealed data later in §5.4.

4. THE LOG SERVER DESIGN

As discussed earlier, our idea for SGX-Log is to secure logs using SGX's features and a hash key chain. To this end, all the computations that use the initial hash key and handle log messages

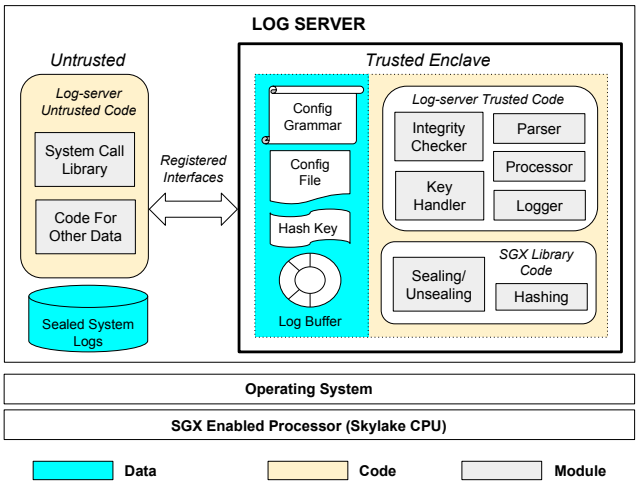

Figure 2: SGX log-server design.

need to be secured using trusted hardware. This logically partitions the system into two parts: the trusted part that uses sensitive data, and the untrusted part that does not, as illustrated in Figure 2, which shows how we partition our SGX-enabled log-server.

4.1 Trusted Component

Trusted data. In SGX-Log, we aim to protect sensitive log messages, log configurations and configuration grammar rules. Particularly, we want to utilize a ring buffer to solve the streamlined logs challenge in § 3.4-C4. As a result, the log buffer, configuration file and configuration grammar rules are protected as trusted data components inside an enclave. In addition, the initial key for the hash chain is important to generate the complete hash chain for message verification. Thus, it also becomes trusted data.

Trusted code. Trusted code performs all secure logging operations by operating on trusted data. Its high level role is to securely record and verify system logs. It therefore requires implementations of sub-tasks involved in logging operations and use of SGX's trusted services to provide security guarantees. As a result, trusted modules can be grouped into two categories: (a) log-server trusted code and (b) SGX library code. Log-server trusted modules, specifically `Logger`, `Parser` and `Processor`, perform log recording, while the `Key Handler` and `Integrity Checker` modules implement log verification. In particular, the log-server modules perform the following tasks:

1. `Parser` tokenizes incoming message fields using standard message format.
2. `Processor` applies configuration grammar rules and user-specified filter rules.
3. `Logger` is responsible for writing sealed blocks to persistent storage.
4. `Key Handler` generates hash chain keys by successively hashing the initial key.
5. `Integrity Checker` compares the stored MAC value with the newly computed one for tampering detection.

In addition, SGX-Log utilizes the following SGX services:

1. `Sealing/Unsealing` enables secure storage of sensitive data outside an enclave.
2. `Hashing` is important to generate the hash key chain used in MAC value computations.

4.2 Untrusted Component

Untrusted data. All logs sealed and stored outside the enclave boundary are considered to be untrusted data.

Untrusted code. The part of the log-server that does not access raw system logs is kept outside the isolation boundary. As a result, our logging system secures the system logs even when this code is compromised. All the OS communication code, the file I/O manager, and the network I/O manager are considered to be untrusted code.

4.3 Registered Interface

In logging operations, system library operations such as writing sealed logs to disk or reading logs from the network sockets are important to ensure the correctness of trusted code operations. However, calling system calls directly inside the enclave is not allowed in SGX.

To solve this, registered interfaces known as edge routines enable the communication between trusted and untrusted components. They can either be untrusted or trusted edge routines. Untrusted edge routines run outside the enclave and allow enclave code to call untrusted functions in the application. Alternatively, trusted edge routines run inside the enclave and allow untrusted application code to call certain enclave functions. These are necessary for all interaction that the enclave has with the outside world.

5. SECURE LOGGING PROTOCOLS

5.1 Notations and Key Hash Chain

Notations. We use the following notations to explain our secure logging protocols.

- T represents a trusted environment containing all functions that run inside an enclave.
- U represents an untrusted environment containing all functions that run outside of an enclave.
- $Seal(X)$ is a function that seals value X to the platform using a seal key derived from an SGX hardware key.
- $Unseal(X)$ is a function that unseals the sealed data X using an unseal key derived from an SGX hardware key.
- $HMAC(X, Y)$ is a function that generates a message authentication code for value X using key Y.
- $Hash(X)$ is a one-way collision-resistant hash function.
- $rootKey$ is the initial key used to generate a hash key chain. It is accessible only inside an enclave.
- D_i is the log message content generated at time t_i.
- M_i is the log message generated at time t_i and represented using a standard message protocol.
- $failure$ is the flag variable used to detect unintended system shutdown.

Two-dimensional (2-D) hash key Chain. A hash chain is a sequence of keys obtained by successively hashing an initial hash key [20]. Schneier and Kelsey [36] first introduced a protocol to

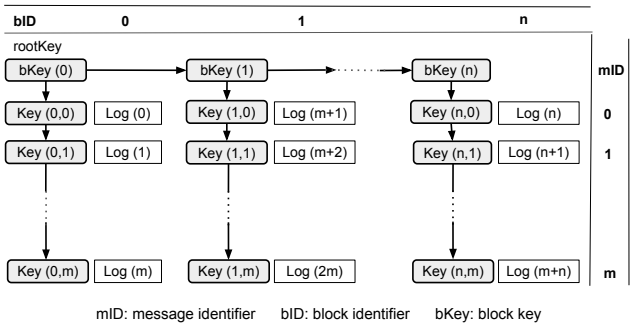

Figure 3: Two dimensional block hash key chain.

generate and verify logs by constructing a linear hash chain. The scheme starts with an initial key. For every new log message, a new hash key is generated using the previous hash key. At any given time point, only the newest key in the hash chain is used to generate the HMAC; older keys are deleted from memory. As a result, an attacker cannot generate valid HMACs for earlier log entries. However, linear hash chains are not practical in (a) power failure, as the current key is lost (challenge §3.4-C3), and (b) efficient verification of an arbitrary log subset (challenge §3.4-C5). We solve these by implementing a two dimensional hash key chain.

A 2-D hash key chain is a modified version of Branched Key Chain [40]. Figure 3 details our two-dimensional hash key chain. We divide log messages into multiple message blocks, each containing a fixed number of log messages ($bSize$). Every block is identified by using a unique identifier called bID. Similarly, a message inside a particular block is identified by using mID. Thus, the two dimensions bID and mID uniquely identify a message.

The scheme starts with an initial key $key(0)$. We initialize it using the $rootKey$. A sequence of hash keys generated using $key(0)$ represents initial keys of each block. The initial key for block k is computed as $key(k) \leftarrow Hash(key(k-1)|bID)$, where bID identifies the k^{th} block. Each block key ($bKey$) forms another hash chain indexed by bID and mID. The i^{th} message key ($mKey$) in block k is computed as $key(k,i) \leftarrow Hash(key(k,i-1)|mID)$.

Using the two dimensional hash key chain has two advantages. First, it enables continuity of hash key chain generation across power cycles. Every time a new block is initialized, we generate the next block key ($nextBKey$) and seal it to the hard disk. When the system is restarted, we read the next block key and resume hash chain generation. Second, two dimensional hash chain allows efficient verification of an arbitrary subset of the log. We traverse the block key chain until we find the hash key for the current block. Later, we use this key for generating subsequent hash keys.

5.2 Protocols

SGX-Log implements protocols for secure logging and integrity check operations. Our protocols use a forward-integrity mechanism [11] implemented using a two-dimensional hash key chain.

5.2.1 Log Generation Protocol

Log generation involves reading event information sent by log-sources and recording it using the log configuration settings. First, we describe the *create-block* sub-routine used to initialize a new block of messages inside the enclave. Later, we detail 3 phases involved in the log generation process: (P1) *startup phase*, (P2) *logging phase* and (P3) *shutdown phase*.

Create-block subroutine. Table 1 details the create-block subroutine. In this subroutine, the sealed object read from secure location is unsealed to obtain the current block key. For the first block we assign an identifier using a monotonic counter. Then, the next block key is computed, and the next count value is sealed to defend against replay attacks using the past sealed data. If a new block is created for the first time then $rootKey$ is used as the first block key. We then initialize the message identifier value. The current block key and the message identifier are later used to generate the message level hash chain corresponding to the messages in that block.

Untrusted: *U*	**Trusted: *T***
Reads sealed data $bKey^{seal}$	
$\xrightarrow{bKey^{seal}}$	
	$x \leftarrow$ Read *monotonic counter*
	if $bKey^{seal}$ is null **then**
	$bKey \leftarrow rootKey$
	$bID \leftarrow x$
	else:
	Unseals block key:
	$(bKey \mid bID) \leftarrow Unseal(bKey^{seal})$
	if $bID \neq x$ **then**
	Report rollback
	$nextBKey \leftarrow Hash(bKey \mid bID+1)$
	$nextBKey^{seal} \leftarrow seal(nextBKey \mid bID+1)$
$\xleftarrow{nextBKey^{seal}}$	
Stores $nextBKey^{seal}$ at secure location	
	$key \leftarrow bKey, mID \leftarrow 0$

Table 1: Create-block subroutine protocol run at the start of logging and after sealing each block to disk.

Untrusted: *U*	**Trusted: *T***
Startup Phase	
Loads enclave data	
	$y \leftarrow$ Read *monotonic counter*
	if $y =$ *firstBlockID* **then**
	Initializes *rootKey*
	else:
	Unseals $failure$ flag, bID, $rootKey$
	if $bID \neq y$ **then**
	Report invalid sealed data
	if *failure* is *true* **then**
	Report power failure attack
	failure $\leftarrow$ *true*
	Runs *create-block* sub-routine
	while no shutdown signal **do**
	Runs *log generation*
Shutdown Phase	
	Receives shutdown signal
	Seals the current state
	failure$\leftarrow$ *false*
	Seal and store
	seal(failure flag | *bID+1* | *rootKey)*

Table 2: Steps in the Startup and Shutdown phase.

P1: Startup phase. Table 2 details the startup phase that runs before the logging phase. The untrusted code creates an enclave. If the logger is executed for the first time then we initialize the $rootKey$ using user input key. If the logger is resumed after a shutdown, we retrieve the last $failure$ flag, bID and $rooKey$ from last sealed data. It then checks for power failure attack and rollback attack using bID and $failure$ flag respectively. Next, it invokes the create-block subroutine and start the log generation phase.

P2: Logging phase. Table 3 details the protocol used for logging operations. In the logging phase, trusted code receives log entries from untrusted code. Streamlined incoming messages are

Untrusted: *U*	**Trusted: *T***
Generates D_i at time t_i Builds standard log: $M_i = f(D_i)$	
$\xrightarrow{M_i}$	
	if ring buffer not full **then** $L_i \leftarrow (M_i \mid bID \mid mID)$ $H_i \leftarrow HMAC(M_i \mid bID \mid mID, key)$ Buffer L_i and H_i $mID \leftarrow mID + 1$ $key \leftarrow Hash(key \mid mID)$
	if *len(ring buffer) = bSize* AND *bSize <= Max buffer size* **then** Reads L_k to L_i from buffer: L_{ki} (L_k is the oldest log in buffer, *i-k = bsize*) $L_{ki} \leftarrow < L_k, L_{k+1}, .., L_i >$ Seals L_k to L_i messages: $M_{ki}^{Seal} = Seal(L_{ki}\|bID)$ Reads H_k to H_i from buffer: H_{ki} $H_{ki} \leftarrow < H_k, H_{k+1}, .., H_i >$
$\xleftarrow{M_{ki}^{Seal}\|H_{ki}}$	
Writes to file: $M_{ki}^{Seal} \mid H_{ki}$	
	Increment *monotonic counter* Cleans ring buffer Runs create-block sub-routine

Table 3: Protocol for secure logging.

Untrusted: *U*	**Trusted: *T***
Reads sealed logs: **for each** *entry* $\in$ Logs **do** $M_{ki}^{seal}\|H_{ki} \leftarrow entry$	
$\xrightarrow{M_{ki}^{seal}\|H_{ki}}$	

	Unseals data:
	$L_{ki} = Unseal(M_{ki}^{seal})$
	Retrieve: *bID*, *bKey*
	if $bID \neq n$ **then**
	Report incorrect block
	$n \leftarrow n+1$
	$bKey \leftarrow Hash^{bID}(rootKey \mid bID)$
	$key \leftarrow bKey$
	for $j \leftarrow k$ *to* i **do**
	$(M_j \mid bID \mid mID) \leftarrow L_j$
	$\hat{H}_j \leftarrow HMAC(M_j \mid bID \mid mID, key)$
	if $H_j \neq \hat{H}_j$ **then**
	Detect tampering in current entry
	$key \leftarrow Hash(key \mid mID)$

Table 4: Protocol for log integrity check.

buffered along with a block identifier and message identifier. For each message, we generate a HMAC value using the corresponding 2-D hash chain key and store it inside the ring buffer. When the buffer contains the maximum number of messages in a block ($bSize$), we seal the buffered messages and store them outside the enclave along with their MAC values. Next, the monotonic counter value is incremented to generate the next block identifier and the buffer is cleaned. We then invoke the create-block subroutine.

P3: Shutdown phase. Table 2 details the shutdown phase, which runs after the enclave receives a shutdown signal. In secure shutdown, the enclave securely seals the current state on the disk.

5.2.2 Integrity Check Protocol

Table 4 details our log integrity check protocol. This protocol checks integrity of all input blocks generated in sequence after the n^{th} block. Similar to the logging phase, trusted code runs until sealed blocks are received from untrusted code. For each block, we first unseal the sealed messages and check for a roll-back attack. Next, we compute the block key by successively hashing the root key using block identifiers. Once the block key is generated, we compute the hash key for each message in the block using that message's identifier. Finally, we compare the MAC value of each message with the original value to detect tampering.

5.3 Security of SGX-Log

SGX-Log utilizes the security guarantees of Intel SGX to protect the confidentiality of logs and their integrity by utilizing a block-level hash key chain. Enclave code and data is protected from any outside privileged components such as a malicious OS, hypervisor, SMM, etc. Therefore, we rely on SGX memory encryption to protect access to logs and security keys by placing log data, hash keys and log configuration parameters inside the enclave. In addition, log generation and integrity code is placed inside the enclave. However, as detailed in [14], SGX is vulenerable to side channel attacks (e.g, cache-timing attacks), physical attacks (e.g, directly unpacking CPU) and microcode attacks (e.g, reprogram machine code) and thus SGX-Log does not defend against these attacks. Outside the enclave, SGX-Log protects confidentiality using sealing/unsealing. However, sealed logs and flag values can be tampered with, deleted or rolled back by malicious user. Thus, we discuss the mitigation of such attacks in §5.4.

5.4 Mitigation of Attacks in SGX-Log

Our SGX-Log design implemented with secure logging protocols provides mitigation against the following attacks.

Unauthorized log reading. In SGX-Log, log messages are protected using SGX hardware features while they are within an enclave boundary. When exported outside of an enclave, messages are sealed to the platform. Reading sealed messages requires access to the seal key reproducible only inside the enclave it came from. Moreover, the SGX seal API provides a strong protection against key-wear-out attacks by generating a unique seal key for each input data block. A key identifier is stored along with the encrypted data blob and later used for unsealing. Thus, an attacker can not read log messages stored inside or outside of the enclave.

Compromising security keys. Our SGX-Log uses seal keys for sealing/unsealing, a root key for generating a hash key chain and a current hash key for generating the next hash key. A seal key is derived by the seal/unseal APIs using the SGX hardware key and offers strong security. In addition, the root key and current hash key are secured inside the enclave, and in case of power failure attack or shutdown, both keys are sealed to the platform and can not be compromised without access to the seal key. We discourage key deletion by replicating them on remote machines.

Log tampering. In SGX-Log, log tampering is protected using both the hardware and cryptographic approaches. SGX memory protection and sealing/unsealing protect log entries inside and outside of the enclave respectively. In addition, SGX-Log deletes the current hash key as soon as a new hash key is generated. Without correct hash keys, the attacker can not generate valid HMACs that pass integrity checks. As a result, the attacker cannot tamper with the logs generated prior to the attacker gaining root privileges, i.e., modify logs, replace logs, or remove logs.

Log deletion. A log deletion attack requires an attacker to delete the sealed messages stored outside of an enclave. Log deletion can be prevented by using expensive read-only disks. Alternatively, we can discourage log deletion by replicating logs on a remote machine using a secure communication channel. The logging enclave can establish a secure connection to a remote enclave for replication. Then storing sealed messages inside the logging enclave until

replication is complete will require an attacker to delete all copies stored on both the log-server and remote machines.

Log truncation. Log truncation is a special type of log deletion attack. In truncation attack, an attacker deletes the most recent entries generated right before the break-in. SGX-Log discourages truncation attacks as the log buffer protects the recent messages inside the enclave using memory protection. If the system is shut down, the system can detect whether any logs had potentially not yet been written to disk prior to shutdown.

Rollback attack. In a rollback attack, an attacker can replace the sealed data from the past recorded entries. For example, the attacker can replace the sealed block containing the next block key value ($nextBKey$) with a past entry to roll back the hash-key chain. However, SGX-Log defends against such attacks by utilizing monotonic counters. Specifically, for each block we assign a unique block identifier (bID) derived using the value of the monotonic counter. In create-block, we seal the $nextBKey$ and the next block identifier ($bID+1$) together. In the subsequent call to create-block, we read the sealed data and retrieve the identifier. If the retrieved value of the block identifier is not equal to the current value of the monotonic counter then we detect the rollback attack. Note that the monotonic counter value is incremented after a block is successfully sealed during log creation.

Similarly, to avoid a rollback of the $failure$ flag, we seal the $failure$ flag along with the next block identifier ($bID + 1$) during shutdown-phase. Upon reloading the enclave, we check the authenticity of the sealed block by comparing ($bID + 1$) with the current value of the monotonic counter. Our choice of utilizing a unique block identifier can also detect missing or incorrect blocks during log verification. We can utilize the $firstBlockID$ to generate the sequence of block identifiers inside SGX.

Power failure attack. SGX-Log maintains a flag to store if shutdown is secure or malicious. In case of intended shutdowns, the enclave securely seals the current logs and the shutdown flag. Upon system restart, the enclave retrieves the flag by unsealing the recently sealed data, verifies its authenticity, and then checks the $failure$ flag to detect unintended shutdowns. Similar to all cryptographic approaches, recovering the current logs which are lost during malicious shutdown is possible only with advanced hardware that is capable of securely sealing the data during power failure.

6. IMPLEMENTATION

We use an SGX-enabled Linux system with an i7-6700 CPU (Skylake) and 64GB RAM operating at 3.40GHz with 8 cores, running Ubuntu 14.04. We installed the latest Intel SGX SDK and SGX driver [1]. SGX-Log's implementation has 4.7K lines of code written in C and can easily be extended to use components from the Linux `rsyslog` [34] server. Note that we do not have to develop any additional code for any client log-source implementation. Instead, we modify the configuration file, from which each log-source knows how to communicate to our log-server. We detail the client configuration process in §7.1.

We implement log parsing, reading/writing, processing, and integrity checking functions as trusted components. On the other hand, untrusted components include the code that reads messages from sockets, writes sealed logs on disk, and dispatches sealed log data to the remote server, etc. We use enclave definition language to register interface functions.

During initial setup, the log configuration file and configuration grammar rules are loaded into the enclave. Our log-server reads messages using an untrusted `reader` component and delivers them to the enclave for further processing. The enclave applies parsing rules to extract meta-data from the log message. For this, we utilize regular expressions from the `syslog` message parser used in Ubuntu's `rsyslog`. After processing, the `writer` module stores sealed data from the enclave on the hard disk. We implement a log rotation policy to configure the file sizes of logs stored on non-volatile storage. We set the ring buffer size to 256KB and each block consists of a maximum of 100 messages.

Log Sources	Total Log Size (KB)	Number of Log Messages	Average Size Per Log (Bytes)
RKHunter	515	7926	65
DCC-Server	558	6649	84
Apache-Unix	697	6184	112
Postfix	540	3923	137
Kernel-Ubuntu	506	3567	141
Amavis	654	3195	204

Table 5: Dataset of log messages from applications generating logs using syslog protocol.

7. EVALUATION

In SGX-Log's evaluation, we are interested to know the overhead of using SGX's trusted execution environment for secure logging. We conduct several tests at both a log client (application using SGX-Log) and server side. §7.1 details the overhead of using SGX-Log to its client applications and §7.2 details the result of the server side evaluation.

7.1 Testing of SGX-Log Client

In the client side evaluation, first we describe how a client application uses SGX-Log for logging and then measure the logging overhead of client applications, e.g., an Apache web server daemon. In our evaluation, we use a typical logging configuration, with a log server running on a separate machine from the clients, which send event information to the log server over the network.

Integration of SGX-Log. Integrating central logging functionality is similar in all applications. First, we register the destination log-server information in the log-client's configuration file. Next, the client application invokes a standard logging API implementing a standard message transfer protocol to send log entries. For example, in order to use Ubuntu's `rsyslog` as a logging solution, Apache Web Server allows to customize its default logger to `rsyslog` via the configuration file. In order to log events, `httpd` (Apache Web Server daemon) uses the `syslog` API to transfer event information to the `rsyslog` server. Using a similar approach, all log clients can be easily configured to use SGX-Log.

Results. In our client side evaluation, we aim to measure the overhead of using SGX-Log in `httpd`'s performance. Apache's `httpd` process is responsible for both serving HTTP requests and simultaneously calling log handlers to record event information. We generated 10,000 web requests at 20 requests/second from 4 concurrent connections using Apache Jmeter [5] and measured the overhead in `httpd`'s throughput with SGX-Log and without SGX-Log.

Interestingly, we observed zero overhead of the web server's throughput because `httpd` uses asynchronous logging. From the daemon's point of view, logging is 'fire and forget'. The responsibility of transferring messages to the server lies with the implementation of the message transferring protocol. Standard `syslog` uses the UDP protocol for communication. On the other hand, its secure version, `Syslog-ng`, uses the TCP protocol to ensure reliable message transfer. As `httpd` uses asynchronous logging, integrating SGX-Log with Apache server has no performance overhead.

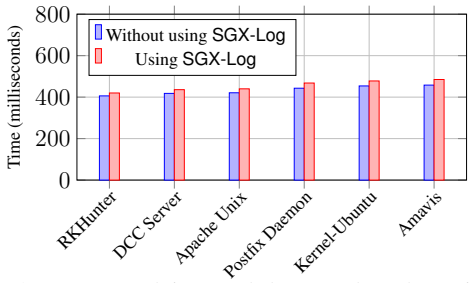

Figure 4: Average total time needed to record 100 logs without and with using SGX ($bSize = 50$ i.e, 2 blocks).

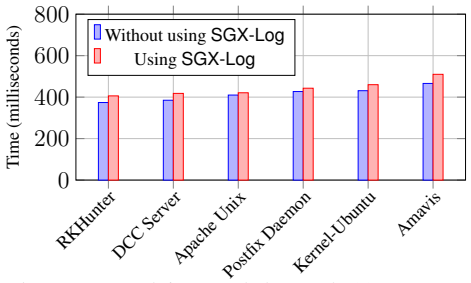

Figure 5: Average total time needed to read 100 messages without and with using SGX ($bSize = 50$ i.e, 2 blocks).

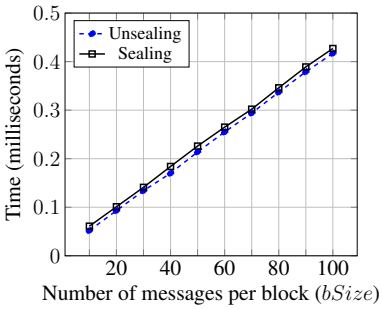

Figure 6: Average time taken by Intel SGX's seal and unseal APIs to process one block of logs containing varying number of entries.

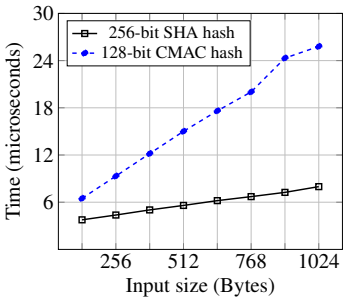

Figure 7: Average time taken by hash and MAC APIs to process input of different size using 256-bit SHA and 128-bit CMAC algorithms respectively.

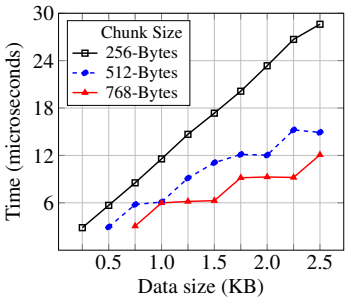

Figure 8: Average time required for copying data blocks (in either direction) across an enclave boundary using different chunk sizes.

7.2 Testing of SGX-Log Server

In SGX-Log's server evaluation, first we measure end-to-end logging overhead. Then, we explore the real sources of overhead by evaluating security critical APIs used in secure logging.

7.2.1 *Overhead of Our Logging Services*

We evaluate the performance of log generation and log reading services carried out at the log-server. Since logging services run continuously as daemon processes and measuring the execution time is impossible, we measure the response time for each operation. In log generation, we measure the time taken by the log-server from receiving the log entries to recording them in the server. Similarly, in log reading we measure the time taken from receiving a read request to handing over the logs to the component used for dispatching logs to the log reader. For a thorough evaluation, we test our log-server using log entries generated by a set of real world programs.

Dataset. To evaluate SGX-Log, we use logs from 6 widely used programs that use the `syslog` protocol for logging. Table 5 details the dataset used in the SGX-Log evaluation. For each program, it shows the total log file size, the total number of log messages and the average message size. It includes logs from 3 server daemons, i.e., the Apache Unix daemon [4], Distributed Checksum Clearinghouse's (DCC) server daemon [15] and the Amavisd: an open source content filter for electronic mails daemon [2]; 2 application daemons, i.e., a Rootkit Hunter (RKHunter) program [33], and Postfix, an open source mail transfer agent [32]; and system logs generated by a remote Ubuntu kernel. Note that the average size per log messages is different for each program.

Results. We evaluate log generation and log reading overheads by using the log entries from all 6 programs. For both operations, we report the average total value of 30 measurements as our final value.

Figure 4 shows the average total time needed to generate 100 logs with using SGX-Log and without using SGX-Log. In each measurement, we read messages from the log files of the corresponding programs and send them to the enclave. Overall, log generation has 4.84% overhead with SGX. The median performance overhead of using SGX-Log in log generation is 21.5ms for 100 messages.

Figure 5 shows the average total time needed to read 100 logs from the server using SGX-Log and without using SGX-Log. In reading, we unseal a block, verify each message and then send messages to the log dispatcher. The median performance overhead of using SGX-Log in reading 100 logs sealed in a single block is 23.5ms. On average, log reading has 6.29% overhead with SGX.

Overall, results show that time taken by logging services increases with the size of message. This is expected because the cost of copying data on disk increases with an increase in the size of the message. Comparatively, total average time for log generation in all programs is higher than total average time for log reading. This is expected as log generation involves parsing of log messages using log configuration rules. To further explore why SGX-Log has a overhead of 4.84% in log generation and 6.29% in log reading, we evaluate the overhead of security APIs and enclave boundary data transfer functions.

We simulate a monotonic counter in software without any latency to measure the maximum throughput of log-server. In practice, monotonic counters use a small latency value to avoid counter overflow, and thus limit the log processing rate. However, we only increment the counter every block, which significantly reduces impact on overhead. Our results from Figure 4 demonstrate a time of roughly 200ms for a 50 message block; assuming monotonic counter latency is less than 200ms, this would be sufficient to avoid any additional overhead from monotonic counter latency.

7.2.2 *Overhead of Using SGX Trusted APIs*

Understanding the costs of logging services requires analysis of

security critical APIs used in SGX-Log. We believe that the trusted services used for sealing, unsealing, hashing and MAC computation are the main sources of overhead. Thus, we measure the overhead of SGX trusted services used in SGX-Log's implementation.

Overhead of block sealing and unsealing. In SGX-Log, seal/unseal APIs efficiently protect logs outside the enclave boundary by operating at the block level. As a result, the amount of data processed by seal/unseal APIs depends on the total number of messages in a block, i.e., $bSize$. We test APIs using log messages from the dataset detailed in Table 5 (avg message size=110KB).

Figure 6 shows the average time needed to seal and unseal one block of logs containing a varying number of log messages ($bSize$). Results show that both sealing and unsealing time grow linearly. Sealing is slower by a constant margin because sealing runs an encryption algorithm on input data padded with extra bits.

Overhead of key hashing and MAC generation. Key hashing is critical in the 2-D block hash key chain implementation. For every message, a new key is generated by hashing the previous key and the log identifier. Using this key, we compute MAC value for the current message to ensure its integrity. According to the `syslog` message transfer protocol, the maximum size of a log message cannot be more than 1024 bytes. Thus, we evaluate the MAC generation API by varying the input data size from 128 bytes to 1024 bytes. Using similar input sizes, we report the overhead of using an initial hash key of up to 1024 bytes.

Figure 7 shows the average time needed to generate hash and MAC values based on different input data sizes. Results show that the overhead of both hashing and MAC computation increases with the increase of input data size. As expected, we observe that the 256-bit SHA hashing algorithm runs faster in comparison with AES 128-bit CMAC algorithm.

7.2.3 *Overhead of Enclave Boundary Data Transfer*

Secure logging operations involve copying sealed data across the enclave boundary. Thus, we evaluate the overhead of copying data across the enclave boundary. The cost of copying data from the application into the enclave is equal to the cost of copying data from inside the enclave to the application.

Figure 8 shows the average time required to copy data inside the enclave using multiple fixed-size chunks of data. Unsurprisingly, data transfer time increases with an increase of data size. More interestingly, when data is divided into smaller chunks, the number of enclave entry and exit operations increases and thus results in a higher overhead, and when the data buffer size is not a multiple of the chunk size, we observe a step-like performance overhead. This result is useful for selecting the optimal size for a data chunk. For example, if we expect to transfer data from 1 KB to 1.5 KB in size, then a chunk size of 768 bytes requires only two copy operations. However, if the chunk size is reduced to 512 bytes, then copying requires more than two copy operations for data greater than 1024 bytes, resulting in a higher overhead.

8. DISCUSSIONS

SGX-Log is not perfect. In this section, we discuss the limitations of our implementation and outline our future work on how we could address them.

Log Availability. In SGX-Log when messages are stored outside the enclave boundary, we protect the confidentiality by utilizing SGX's sealing feature. Thus, in log reading we first need to unseal the messages, verify their integrity, and then deliver them securely to the requester. Our evaluation of trusted APIs detailed in §7.2.2 show that SGX has very efficient sealing and unsealing APIs. However, single logger instance may not scale well given the limited size of EPC. Moreover, since sealed data can only be unsealed on the same SGX machine, log availability is restricted by depending on a single SGX machine. To support scalable access to logs, we propose partitioning the logs across multiple enclaves, each running on different machines.

In secure remote communication, SGX supports remote attestation for verifying the remote platform. Mutual authentication with remote enclaves ensures that logs are communicated from a valid local enclave to a valid remote enclave. Additionally, we can utilize cloud attestation, detailed in [38], to verify that the SGX processor is running in a valid cloud provider's data center. Once the platform is authenticated, two enclaves can share keys using the trusted Key Exchange (KE) library in the Intel SDK, detailed in Intel SGX's developer reference manual [23]. After a successful key exchange, an encrypted connection can be established between two enclaves for secure transferring of unsealed logs. Remote enclaves can follow the same logging protocols as the original logger (§5.2) in order to securely store logs on each remote machine. The enclave that originally generates all the logs will store metadata about log partitions. Whenever a block is created, it is securely sent to the appropriate remote enclave. When a user requests a log message from the original enclave, this enclave directs the request to the corresponding remote enclave, which unseals the block corresponding to the requested message and delivers it over a secure channel.

Ring-3 level protection. The main objective of a logging system is to receive and record event information. Moreover, logs recorded should easily be available to a user or administrator for analysis. Intuitively, a ring-3 level implementation best serves these requirements as the logging application does not need to interact with the lower level hardware. As a result, standard log systems on Linux, such as `rsyslog`, implement their logging mechanism at the ring-3 level. Interestingly, SGX only allows applications running at the ring-3 level to leverage isolated enclave execution. OS level code is used only during enclave creation and initial data loading.

Ring-3 implementations incur an overhead of pulling kernel logs from OS level code to the less privileged logging code. In Ubuntu, this is achieved by using a kernel level log buffer whose contents are accessible using system calls. Since system calls are untrusted, an enclave relies on a non-enclave component to retrieve kernel log events. Our SGX-Log does not aim to defend against any attacks that target kernel log buffers or the channel between the kernel and the ring-3 logging component. Our focus is to secure valid logs from authorized sources during recording, after recording, and when reading from log-server.

Limitation on memory capacity. Intel SGX technology has a hard limit on the protected memory size, typically 64MB or 128MB. Applications whose memory requirements exceed the EPC size swap pages between the EPC and unprotected DRAM. As a result, the number of active enclaves that can run efficiently is limited. Despite this performance constraint, 128MB of memory is sufficient to run 2 enclave instances running log generation and log reading operations separately. Memory can be utilized more efficiently by increasing the block size used for sealing.

Future Work. Our future work involves extending the SGX-Log implementation to use secure and reliable communication in remote logging. We plan to use the open source `syslog` protocol implementation, i.e., `syslog-ng`, which implements a secure TCP protocol for communication. With the rise of the Internet-of-Things (IoT), a large number of systems producing continuous logs poses a scalability challenge. To address this, we aim to build a distributed log-server based on SGX where each local server runs

a trusted environment similar to SGX-Log. With a master-slave model, all local log servers can be controlled over a secure channel by a master server, again running SGX-Log.

9. RELATED WORK

9.1 Systems Securing the Logging

Private Key Based Schemes. Bellare and Yee (BY) [11] were the first to propose a secure logging system that uses the forward-integrity property by periodically updating the key used to compute a MAC over audit logs. To achieve this, they feed different secret values to a family of pseudo-random functions. Thus, even if the attacker compromises the current MAC value, it is impossible for him or her to determine the MACs used for past log entries. He or she can delete the log entirely, but cannot modify it without detection. Schneier and Kelsey (SK) [36] introduced a protocol to generate and verify logs using forward-secure MACs and one-way hash chains. All the logs generated prior to the logging machine's compromise are secured by constructing linear hash chains, where the current hash value is computed using previous hash values calculated over past log entries.

BY and SK rely on a trusted machine to detect tampering. This may result in delayed detection cases. Another approach, called Forward-Secure Sequential Aggregate (FssAgg) [29], offers a private verifiable approach for practical secure logging without any reliance on a third party or secure hardware. To achieve this, forward-secure signatures (or MACs) generated by the same signer are combined sequentially into a single aggregate signature. Aggregating signatures enables space efficient integrity checking in comparison to BY and SK.

Public-Key Based Schemes. Public-key based schemes aim at securing logging systems without the need to store the secret on the same machine. Logcrypt [22] uses public-key cryptography to allow signatures to be created with one key and verified with a different one. Such signatures void the need of using a trusted environment in signature verification. Another public key mechanism proposed in FssAgg [29] uses this principle to protect the key used in aggregate signature generation. Although these schemes do not require a trusted environment, they still rely on multiple keys for providing security. Moreover, the attack surface increases as the number of keys increases. Again, they rely on a third party for secure key generation and distribution.

Another set of key-insulated public key crypto-systems [9, 16] propose a hybrid approach to use trusted hardware to protect the master key. This key is used to update individual local keys stored on untrusted machines. Threshold cryptography ensures that exposure of such local keys does not have significant impact on the security of the system. Unfortunately, communication with the trusted hardware is expensive. Our SGX-Log not only stores the secret in the enclave but also runs the code in an isolated environment.

Tamper Resistant Schemes. Tamper resistant schemes can use trusted software or trusted hardware. Tamper resistant software solutions can be realized either using homomorphic encryption [35] or code obfuscation [13]. Homomorphic encryption adds extra security by performing calculations over encrypted data rather than on raw text, while obfuscation performs program transformations to make the code difficult to analyze. Unfortunately, software solutions often fail to meet acceptable performance requirements.

Alternatively, Chong et al. [12] presented using tamper-resistant hardware in conjunction with a protocol proposed by SK to implement a secure logging system. In this scheme, tamper resistant hardware protects only the secret and does not allow the execution of critical operations in an isolated environment. Additionally, it has limited memory and incurs the cost of communication with other components. Similarly, Levin et al. [28] proposed to use a trusted counter and key to combat equivocation in large distributed systems. In recent work, Sinha et al. [40] leveraged trusted hardware features to maintain the continuity and secrecy of the key chain across a power cycle.

Systems	C1	C2	C3	C4	C5	C6	C7	C8
Schneier-Kelsey [36]	✓	✗	✗	✗	✗	-	-	-
Bellare-Ye [10]	✓	✗	✗	✗	✗	-	-	-
FssAgg [29]	✓	✗	✗	✓	✗	-	-	-
Logcrypt [22]	✗	✗	✓	✗	✗	-	-	-
FssAgg-Public [29]	✗	✗	✓	✓	✗	-	-	-
iButton [12]	✓	✗	✓	✓	✓	✗	✗	✗
TPM [40]	✓	✓	✗	✗	✓	✗	✗	✗
CloudLoggger [31]	✓	✗	✓	✓	✓	✓	✗	✓
Our SGX-Log	✓	✓	✓	✓	✓	✓	✓	✓

C1: stores less than 3 keys? C2: efficient hash key chain?
C3: without delayed detection? C4: defeats truncation attack?
C5: detect power failures? C6: commodity hardware?
C7: h/w based seal/unseal? C8: large protected memory?

Table 6: Comparison with the related work.

Comparison. Table 6 shows an overview of the effectiveness of three private key based logging schemes [10, 29, 36], two public key schemes [22, 29] and three hardware based solutions [12, 31, 40] when compared with our SGX-Log. We use 8 factors coming from 3 domains, namely *key utilization* (C1,C2), *defeating special attacks* (C3,C4) and *hardware features* (C5-C8).

Overall, SGX-Log is better than all single key crypto-based solutions because we utilize trusted hardware for key protection. We propose a secure logging system that securely stores one key as opposed to the public key crypto-systems. This limits the attack surface. SGX-Log beats both the private-key based and the hardware-based solutions in efficiently detecting integrity violations. We achieve this by using a block hash key chain similar to the branched key chain in [40]. In comparison with the hardware solutions proposed in [12, 40], SGX-Log uses commodity Intel SGX hardware. It implements memory protection at the processor level to reduce the overhead of communication through an isolation boundary. Moreover, our solution is efficient as it uses platform supported data sealing and unsealing features.

9.2 SGX and Its Applications

With the emergence of SGX as a commodity trusted environment, a lot of efforts have focused on enabling new features in the platform and evaluating the security of SGX. For instance, [3, 21, 30] detail the importance of using hardware features to perform trustworthy computations. Moat [41] studied the confidentiality properties of applications running on SGX. Xu et al [42] introduced a controlled side-channel attack that allows untrusted operating systems to extract sensitive information from protected applications.

Another set of research focuses on using SGX to secure end-to-end user applications [27], cloud applications [8, 38], network applications [25, 39] etc. In particular, Lal and Pappachan [27] proposed an architecture for securing video conferencing using Intel SGX. With shielded execution on Intel SGX, Haven [8] protects the code and data in a cloud environment. Krawiecka et al [26] secured password databases with SGX. Another study in [7] proposed a framework and design principles for integrating games with SGX. OpenSGX [24] provides an open source platform with a fully functional and instruction compatible emulator to enable the development of TEE based applications. Kim et al [25] leveraged OpenSGX to secure software defined networks and Tor networks.

Shih et al [39] used OpenSGX to secure Network Function Virtualization (NFV) applications with isolated execution principles.

10. CONCLUSION

Securing a log system is a critical problem because system logs not only provide a valuable view of both current and past states of a computer system, but they also possess important forensics value. This paper introduces SGX-Log, a new secure logging system to protect both the integrity and confidentiality of log data. Our SGX-Log is motivated by the recent advances in trusted execution environments and uses commodity Intel SGX as a TEE solution as opposed to using expensive custom hardware. SGX-Log partitions the standard log system and then executes the core logging operations in an isolated environment. Moreover, it leverages data sealing and unsealing in SGX for log security. We have implemented SGX-Log in a real SGX platform, and measured the performance overhead incurred in securing the log operations. Our evaluation shows SGX-Log imposes no overhead to the applications that generate log data, and very small overhead (less than 7%) to the log daemons.

Acknowledgement

We thank our shepherds Manuel Barbosa and Andrew Paverd as well as the anonymous reviewers for their valuable comments. This research was supported in part by NSF awards CNS-1564112 and CNS-1629951. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the NSF.

11. REFERENCES

[1] Intel software guard extensions for linux os, 2016. https://01.org/intel-software-guard-extensions/downloads/intel-sgx-sdk-linux-1.5-beta-release.

[2] Amavis: Open source content filter for electronic mail. https://www.ijs.si/software/amavisd/.

[3] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for cpu based attestation and sealing, 2013.

[4] Apache: Http server project. https://httpd.apache.org/.

[5] Apache jmeter. http://jmeter.apache.org/.

[6] Ahmad Atamli-Reineh and Andrew Martin. Securing application with software partitioning: A case study using sgx. In *Security and Privacy in Communication Networks*, volume 164, pages 605–621. Springer International Publishing, 2016.

[7] Erick Bauman and Zhiqiang Lin. A case for protecting computer games with sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX '16, pages 4:1–4:6, New York, NY, USA, 2016. ACM.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[9] Mihir Bellare and Adriana Palacio. Protecting against key-exposure: strongly key-insulated encryption with optimal threshold. *Applicable Algebra in Engineering, Communication and Computing*, 2006.

[10] Mihir Bellare and Bennet Yee. *Topics in Cryptology — CT-RSA 2003: The Cryptographers' Track at the RSA Conference 2003 San Francisco, CA, USA, April 13–17, 2003 Proceedings*, chapter Forward-Security in Private-Key Cryptography. 2003.

[11] Mihir Bellare and Bennet S. Yee. Forward integrity for secure audit logs. Technical report, 1997.

[12] Cheun Ngen Chong, Zhonghong Peng, and Pieter H Hartel. Secure audit logging with tamper-resistant hardware. pages 73–84, 2003.

[13] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[14] Victor Costan and Srinivas Devadas. Intel sgx explained. Technical report.

[15] Distributed checksum clearninghouses (dcc). https://www.dcc-servers.net/dcc/.

[16] Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Key-insulated public-key cryptosystems. Cryptology ePrint Archive, Report 2002/077, 2002. http://eprint.iacr.org/.

[17] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2002.

[18] R. Gerhards. The syslog protocol, 2009.

[19] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 2002.

[20] Hash chain:. https://en.wikipedia.org/wiki/Hash_chain.

[21] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. ACM.

[22] Jason E. Holt. Logcrypt: Forward security and public verification for secure audit logs, 2005.

[23] Intel sgx sdk: Linux developer reference. https://01.org/intel-software-guard-extensions/documentation/intel-sgx-sdk-developer-reference.

[24] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. Opensgx: An open platform for sgx research. 2016.

[25] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 7. ACM, 2015.

[26] Klaudia Krawiecka, Andrew Paverd, and N. Asokan. Protecting password databases using trusted hardware. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX '16, pages 9:1–9:6, New York, NY, USA, 2016. ACM.

[27] R. Lal and P. M. Pappachan. An architecture methodology for secure video conferencing. In *Technologies for Homeland Security (HST), 2013 IEEE International Conference on*, pages 460–466, Nov 2013.

[28] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 1–14, Berkeley, CA, USA, 2009. USENIX Association.

[29] Di Ma and Gene Tsudik. A new approach to secure logging. *ACM Transactions on Storage (TOS)*, 5(1):2, 2009.

[30] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, New York, NY, USA, 2013. ACM.

[31] H. Nguyen, B. Acharya, R. Ivanov, A. Haeberlen, L. T. X. Phan, O. Sokolsky, J. Walker, J. Weimer, W. Hanson, and I. Lee. Cloud-based secure logger for medical devices. In *2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pages 89–94, June 2016.

[32] Postfix: Open source mail transfer agent. http://www.postfix.org/.

[33] Rootkit hunter project. http://rkhunter.sourceforge.net/.

[34] Rsyslog: The rocket-fast system for log processing. http://www.rsyslog.com/.

[35] Tomas Sander and Christian F Tschudin. Protecting mobile agents against malicious hosts. In *Mobile agents and security*, pages 44–60. Springer, 1998.

[36] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98. USENIX Association, 1998.

[37] Bruce "Schneier and John" Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 1999.

[38] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015.

[39] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-nfv: Securing nfv states by using sgx. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks*, New York, NY, USA, 2016. ACM.

[40] Arunesh Sinha, Limin Jia, Paul England, and Jacob R Lorch. Continuous tamper-proof logging using tpm 2.0. In *International Conference on Trust and Trustworthy Computing*, pages 19–36. Springer, 2014.

[41] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15. ACM, 2015.

[42] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.