International Journal of Cyber-Security and Digital Forensics (IJCSDF) 5(3): 122-131
The Society of Digital Information and Wireless Communications, 2016 (ISSN: 2305-0012)

# SBBox: A Tamper-Resistant Digital Archiving System

Monjur Alam[a], Zhe Cheng Lee[b], Chrysostomos Nicopoulos[c], Kyu Hyung Lee[d], Jongman Kim[b], Junghee Lee[e]

[a]Georgia Institute of Technology, Atlanta, Georgia, USA
[b]Soteria Systems, Atlanta, Georgia, USA
[c]University of Cyprus, Nicosia, Cyprus
[d]University of Georgia, Athens, Georgia, USA
[e]University of Texas at San Antonio, San Antonio, Texas, USA

## ABSTRACT

Reliable forensic data (kernel, socket, audit, other user defined data, etc.) is imperative when investigating cybercrimes. While static and dynamic forensic data collection techniques have already been proposed, none of them pay attention to the process of storing the collected data securely. If the forensic data is tampered with while in storage – after collection – investigators cannot rely on the collected data. In this paper, we propose a hardware/software collaborative novel mechanism for capturing forensic data. The proposed BlackBox Engine ensures data integrity, whereby distorted data can be traced out. The BlackBox Engine runs on a proposed hardware device called Server BlackBox (SBBox). Compared with append-only storage, SBBox offers a more comprehensive solution for forensic data collection and storage, including capturing changes in any file, compressing data, and reconstructing distorted data. Due to close integration between application software, kernel, and proposed hardware, it is virtually impossible for an intruder to interfere with the system by any unfair means. After running standard benchmarks, we observe that the proposed solution incurs minimal overhead of 3.2%, 4.9%, and 1.4% for the CPU, memory, and network, respectively.

## KEYWORDS

Hardware-assisted security, Secure storage

## 1 INTRODUCTION

As cybercrimes evolve and become more intelligent and threatening, it becomes imperative to collect various forensic data. Once an administrator detects that an intrusion has taken place, the next steps are to investigate how it happened and what damage has been caused to the system. An administrator typically utilizes system and network logs to understand the source of an attack and the damage to the system.

To analyze the system, various static and dynamic techniques have been proposed to collect forensic data [4-10]. Recently, researchers have focused on generating accurate [11-14], space-efficient [15-17], and privacy-preserving [41] forensic information to provide accurate attack analysis. However, an important obstacle to forensic analysis is the lack of secure storage. An intelligent attacker tries to falsify or destroy forensic data to avoid detection and/or prosecution. The attacker tries to get root privilege (e.g., a kernel rootkit, a privilege escalation attack, or an inside attacker) to contaminate forensic information. Another important problem we need to consider is how to deliver the data to an investigator safely, under any condition. Forensic data is frequently updated, and the investigator may want to only look at log entries within a specific time period. The investigator typically requires a software method to reconstruct forensic data before he/she can analyze it. However, it is difficult to guarantee a secure reconstruction and delivery from the compromised host. If the host kernel is compromised, it is a challenge to reconstruct and deliver the forensic data to the investigator.

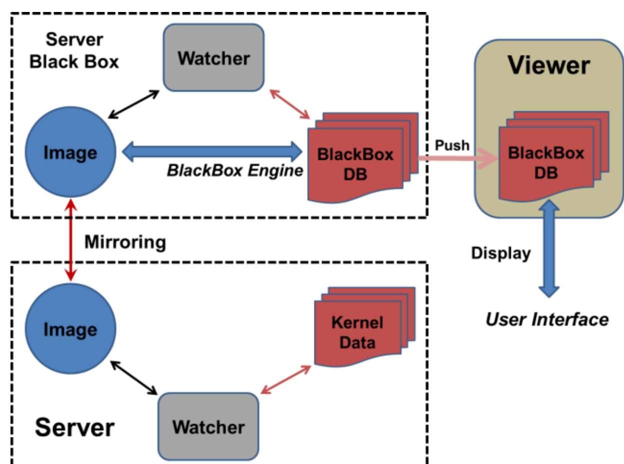


**Figure 1.** A high-level view of the architecture of the proposed SBBox solution.

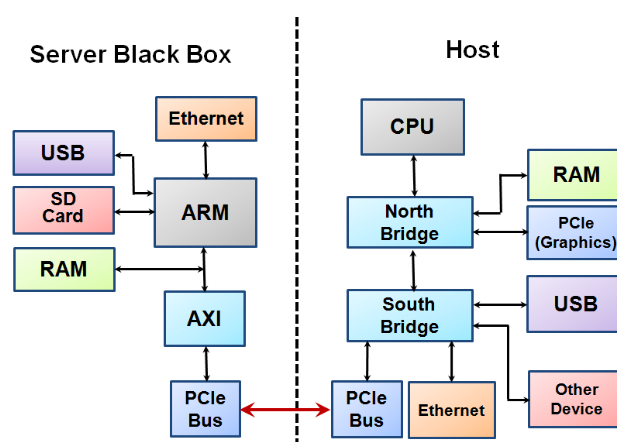


**Figure 2.** The hardware components of the SBBox and host machines, and their interactions.

Unfortunately, current storage solutions for forensic data suffer from one or more limitations. Traditional storage systems or even recent tamper-evident storage mechanisms, such as versioning filesystem, cannot protect the data from the attacker who has root privileges. Recent tamper-resistant storage solutions, such as WORM (Write-Once-Read-Many), or append-only storage [18, 19], may provide secure data keeping. However, they do not provide secure end-to-end delivery. If the metadata, or the reconstructing software, are compromised, the investigator may miss something important, even if all relevant data is stored in the storage device.

In this paper, we propose a hardware-assisted end-to-end secure log storage device, called *Server BlackBox* (SBBox), to achieve the above goals. The SBBox is an extension of append-only storage [1], which allows only read and append operations, prohibiting erase and overwrite operations. The append-only storage has been demonstrated to be effective in protecting reference data [1]. This work is the first attempt to introduce the concept of append-only storage and to apply it to protect reference data of security solutions. SBBox is still a type of append-only storage, but it is more comprehensive. It consists of a kernel module for log acquisition, an isolated storage device for secure record keeping, and independent computing resources to safely reconstruct data and deliver it to the end-user under any attack. More specifically, we have developed a kernel module that is located on the host machine to monitor log-update events, and to send incremental data to SBBox through a PCI bus. We also implement a hardware solution that has dedicated storage, to keep data secure. Due to the close integration between application software, kernel, and proposed hardware, it is virtually impossible for an intruder to interfere with the system by any unfair means.

The rest of this paper is structured as follows: after the proposed solution is presented in Section 2, Section 3 presents experimental results. Section 4 discusses related work and, finally, Section 5 concludes this paper.

## 2 THE PROPSOED SERVER BLACKBOX FRAMEWORK

The goal of our work is to design and implement a procedure that can make an accurate copy of forensic data generated by the kernel, and keep the data secure. The word “secure” signifies that the data should not be modified by intruders, or data can be traced back, even if it has somehow been modified. The intention of data modification by intruders is to hide their activities from the administrator during the analysis/investigation of

forensic data. For example, the intruder can login to a host, and then it can delete the login history.

The threat model assumed in this paper is that intruders may acquire root privileges and can make changes to any software (including the operating system), device driver, and any type of software-based security solutions. The goal is to protect forensic data, even if intruders acquire root privileges.

Toward this end, we have developed SBBox, which ensures that, even if data is tampered with, the original data can be retained. SBBox has a dedicated hardware component and various software components. A high-level overview of the SBBox system architecture is presented in Figure 1. When a host machine generates log data and stores it into log files, the host machine's Watcher module, which monitors the host machine's logs at the kernel level, immediately sends an incremental log to SBBox through the PCI bus (faithful record acquisition). Once the log is stored in SBBox, it is physically impossible to tamper with (e.g., falsifying or destroying) its data (secure record keeping). Upon detection from SBBox's Watcher, which monitors for incoming log data sent to SBBox, the BlackBox Engine creates a delta object from the image data. The BlackBox DB keeps all images of the host and pushes the data to the viewer on a request basis. To retrieve the data, we run the reverse BlackBox Engine on the viewer side.

### 2.1 The SBBox Architecture

SBBox consists of multiple hardware components, as depicted in Figure 2. SBBox receives the most recent log data through the PCI bus. The received data is first stored in the memory, and the BlackBox Engine is used to apply δ-compression using SBBox computing power (ARM CPU and RAM). The details of δ-compression will be discussed in the next section. Compressed data is finally stored into the storage device, which can be flash memory, or a regular hard disk. Note that the ARM CPU, RAM, and storage in SBBox are physically independent from the host machine. Thus, it is impossible to tamper with stored data from the compromised host.

To work with the hardware, an application has been developed, which is capable of collecting forensic data in real-time and is capable of storing it into the SBBox without tampering. The data gets transferred through the PCI expansion card, by using Direct Memory Access (DMA) technology, without interfering with the processor. Initially, the PCI driver (an integral part of the operating system) and the proposed dedicated hardware are installed, before the system is ready to work. Since we design the hardware controller of the PCI expansion card, it is quite possible to make it resistant to response requests from the host system. After the system is powered on, the PCI bus is enabled, and the system is ready to capture data. At this time, the data is visible to the user. However, the attacker cannot tamper with data stored in the device, because the device accepts only append operations. The appended data is processed by the BlackBox Engine and stored in the device as delta objects. Moreover, it is virtually impossible for the attacker to inject malicious code into the device driver, as the driver is an integral part of the hardware design of the SBBox itself. When the PCI controller is enabled, it takes control of the PCI bus. The data in the virtual memory on the host machine gets mapped into a physical address space and then DMA is used to copy the contents of the physical memory to an external volatile storage device, such as the SBBox RAM in Figure 2.

### 2.2 The Storage Structure

One of the major hurdles in storing forensic data in practice is the size of the log. It was reported that the audit logs can grow to 800 MB ~ 3.18 GB per day [11, 12, 15]. It is not a trivial task to store such a large log in secure storage. A popular technique to reduce the size of the log is data compression. However, compression incurs overhead on the CPU, memory, and I/O bandwidth. Note that compressing data during idle

times is not a feasible solution, because the data must be stored in secure storage (such as WORM or append-only storage) as soon as possible, to prevent modifications from the attacker. The proposed SBBox solution addresses this problem by compressing incremental data and using its own computing power. The monitoring module sends over the incremental data to SBBox, as soon as the data is available, and SBBox applies δ-compression on the data and stores it into the *BlackBox DB*.

### 2.2.1 The BlackBox Engine

The *BlackBox Engine* employs the concept of $\delta$-compression [35]. This technique deploys a mechanism to derive a meaningful difference $\delta$ between two versions of data, so that any version can be retrieved by using $\delta$ (provided that one version is present). We hereby formally define the $\delta$-compression scheme that is used throughout the following discussion.

Figure 3 presents the BlackBox DB, which comprises two tables. The first table consists of key-value pairs. The time stamp is used as *key*. The *value* (in each key-value pair) is the data itself. The data is divided into equal-length segments. The second table in the BlackBox DB has the same format, but it contains only the incremental data; i.e., the data difference between two consecutive time stamps. For example, in Table 2 in Figure 3, a row with key $t_{1,2}$ contains the data that constitutes the difference between the corresponding rows with keys $t_1$ and $t_2$ in Table 1.

Let us denote the image data at $t$ and $(t + 1)$ as $\partial_t$ and $\partial_{t+1}$, respectively. We define $\delta$-compression as $\partial_\delta = f(\partial_t, \partial_{t+1})$. In general, $\partial_{\delta k} = f(\partial_{ti}, \partial_{tj})$, where $j \neq i$ and $j > i$.

1. We use two tables: one for the initial file, the other for the updated file. For a big data portion, we use a multi-level table. First, we split the big data into a chunk of data sets. For example, $\partial = \partial^1 + \partial^2 + \cdots + \partial^n$. For each $\partial^i$ (for $i = 1,2,\cdots,n$), we compute $f(\partial^i)$, which is, essentially, a compression of $\partial^i$. At the second level, we combine the sets and generate $f(\partial)$. If there are more levels, we repeat this procedure until completion.

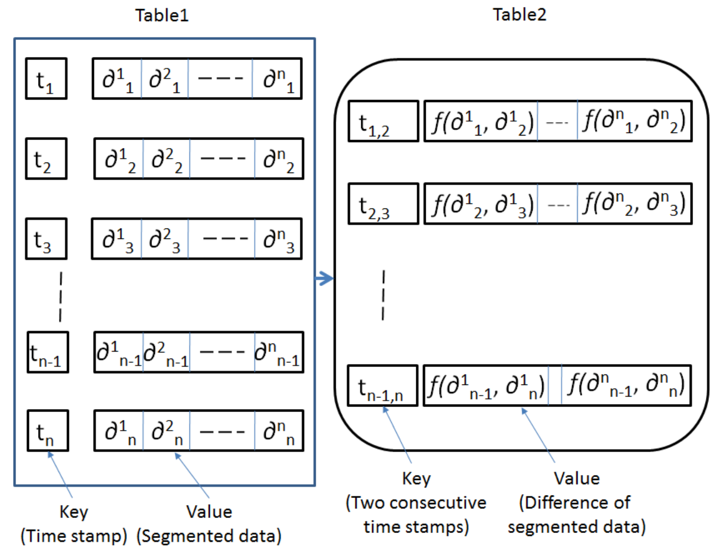


**Figure 3.** The BlackBox DB, which employs two tables to implement the δ-compression scheme.

2. The *delta* $f(\partial)$ for data $\partial$ is stored in the *BlackBox DB*. Let us denote $\partial_i$ as the $i^{th}$ image, and its corresponding *delta* is $f(\partial_i)$. Now, the image is modified to $\partial_{(i+1)}$. In this case, this incremental change must be used to construct $f(\partial_{(i+1)})$. We use the second table and we compare with the first table. The second table is constructed with knowledge from the first table, with very minor computations. Let us divide $\partial_{(i+1)} = \partial^1_{(i+1)} + \partial^2_{(i+1)} + \ldots + \partial^n_{(i+1)}$. It is not necessary that $\partial^1_i = \partial^1_{(i+1)}$. However, for incremental changes, most of the cases will, indeed, result in $\partial^i_i = \partial^j_{(i+1)}$ (for $j = 1,\cdots,n$). Thus, most of the time, we do not need to compute *delta*. Let us say, for example, that $\partial^k_i \neq \partial^k_{(i+1)}$ (for any $k = 1,\cdots,n$). We need to generate $f(\partial^k_{(i+1)})$ only at level one; all the other *delta* will be taken by using the first table. The same logic is followed for the next level. After constructing the final $f(\partial_{(i+1)})$, we keep it in the *BlackBox DB* as a substitute for $\partial_{(i+1)}$. Now, we have two tables; we modify the

first table by the second one, and we compute the new table for the new image.

Our observations indicate that a three-level table is sufficient to handle a big image, and the latter's size is drastically reduced using the afore-mentioned process.

### 2.2.2 The BlackBox DB as Secure Storage

The proposed SBBox framework takes a prominent role in detecting and retrieving the Linux configuration file. As the SSBox has direct access to the host's memory, it can capture the image – for any change in configuration – and create a *delta*, which is stored in the *BlackBox DB*. Let us assume that an attacker logs into the host machine at time $(t + \delta)$. The state of the kernel */proc* file system at time $t$ is, say, $\partial_t$, and the corresponding *delta* is $f(\partial_t)$. Keep in mind that the data capture using SBBox is performed in real-time; thus, whatever the contents of */proc* file may be, there will be a duplicate copy of the image in SBBox with the same contents. Let us assume the attacker has finished with his/her attack, and has modified the */proc* file to its previous state. This implies that if one analyzes the kernel file system state, there will be no way to detect the specific malicious activity that the attacker has conducted. Let the state of the */proc* file be $\partial_{(t+\delta)}$ after the malicious activity. After the modification, the image becomes $\partial_{(t+\delta+\varepsilon)}$. The attacker is intelligent enough to ensure that $\partial_t = \partial_{(t+\delta+\varepsilon)}$. Therefore, the state of the kernel, or */proc*, is identical at times $t$ and $\partial_{(t+\delta+\varepsilon)}$. However, the *BlackBox DB* contains three different hashes for these three different time instances. Specifically, it contains $h(\partial_t)$, $h(\partial_{(t+\delta)})$, and $h(\partial_{(t+\delta+\varepsilon)})$ for time $t$, $(t + \delta)$, and $(t + \delta + \varepsilon)$, respectively. Even though $\partial_t = \partial_{(t+\delta+\varepsilon)}$, $h(\partial_t) \neq h(\partial_{(t+\delta)}) \neq h(\partial_{(t+\delta+\varepsilon)})$, every time there is any incremental change in the image, the *BlackBox Engine* creates a corresponding *delta*, which is stored in the *BlackBox DB*.

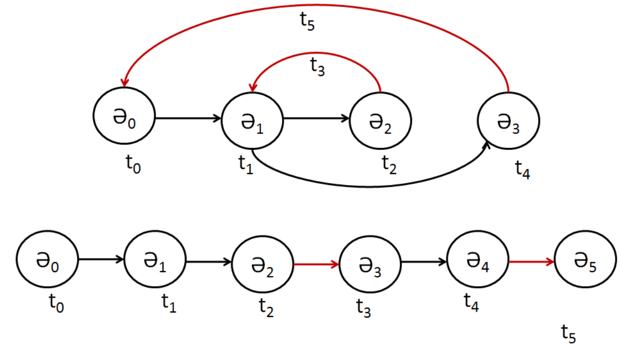


**Figure 4.** The top figure illustrates file states at different times $t_i$. The bottom figure depicts the states of the corresponding delta object.

This process can be illustrated by using Figure 4. Here, the state of the */proc* file at $t_0$ is $\partial_0$. At time $t_1$ and $t_2$, the states of the file change to $\partial_1$ and $\partial_2$, respectively, where $t_1 = t_0 + \delta$ and $t_2 = t_1 + \delta$ $(\delta > 0)$. At $t_3$ $(t_3 > t_2)$, the state of $\partial_2$ returns to the state of $\partial_1$. So, $\partial_{t1} = \partial_{t3}$. Now, let us see the corresponding *delta* object created by the *BlackBox Engine*, as shown at the bottom of Figure 4. We can see that the corresponding *deltas* for the */proc* file at $t_1$ and $t_3$ are $\partial_1$ and $\partial_3$, respectively; i.e, $\partial_{t1} \neq \partial_{t3}$. The same situation is encountered at $t_0$ and $t_5$, where the state of the file is identical, but the *BlackBox Engine* can capture the changes by generating $\partial_0$ and $\partial_5$. These are stored into the secure *BlackBox DB*. In general, if two events with identical states ($\partial_i = \partial_j$; $i = j$) are triggered by the *Watcher*, the *BlackBox DB* always stores two events with different states ($\partial_i = \partial_j$; $i \neq j$).

## 3 EVALUATION

### 3.1 Performance Evaluation

The prototype of the SBBox framework was implemented on the ZC706 evaluation kit [37]. The ZC706 kit is connected to the host through a PCI Express (PCIe) Gen2 x4 slot. The main controller is the Zynq-7000 [38]. It has an ARM Cortex-A9 processor, 2 GB DDR3 memory, and external I/O peripherals. PetaLinux [39] was

ported to the ZC706, and SBBox was developed on top of PetaLinux.

The performance and overhead of SBBox were measured using a regular desktop as a host. The host machine is equipped with an Intel Core i3 processor running at 3.3 GHz and having 4 GB DDR3 SDRAM. In terms of operating system, Ubuntu 12.04 is installed with the Linux kernel 3.9.2.

To calculate the incurred performance overhead, the Apache benchmark [3] is run on a system with and without SBBox. The log generated by the web server, as well as the kernel, is stored into the *BlackBox DB*. The resulting log size is in the range of 500 KB to 2.5 MB. The experiment was conducted for a 1000-request set. The same experiment was also performed in the absence of SBBox. The two different response times are depicted in Figure 5. The response time increases by 1.6%, on average, when using SBBox.

To evaluate the memory and processor overheads, the STREAM benchmark [2] is run on a system with and without SBBox. The STREAM benchmark is a simple synthetic benchmark program that measures sustained memory bandwidth (in MB/s). The memory bandwidth is measured and averaged over 10 different executions of the same experiment. Figure 6 illustrates the sustained memory bandwidth when the STREAM benchmark is run with and without SBBox. This experiment quantifies the memory overhead (in terms of additional memory bandwidth) when using SBBox. The resulting sustained memory bandwith is measured over a specific time interval (known as the Shot Interval). There is an approximately 5% overhead, on average, when using SBBox. Note that there is a sudden jump in the required memory bandwidth when the shot interval increases from 1 msec to 10 msec.

### 3.2 Potential Threats

As we assume the attacker may acquire administrative privilege, the helper software running on the host might be killed. To cope with this issue, the helper software could be integrated with the kernel.

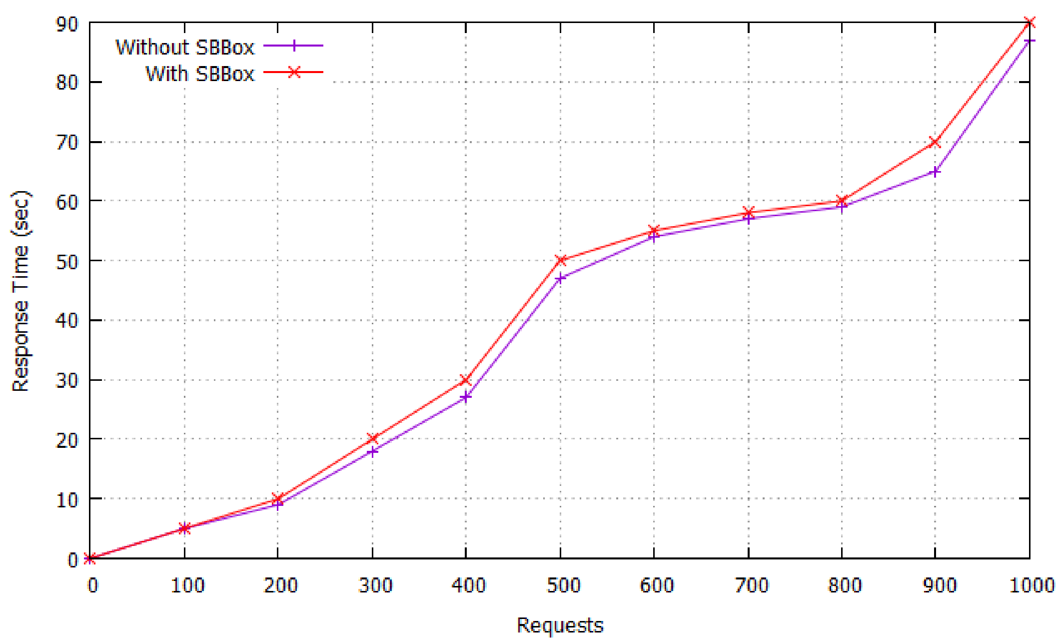


**Figure 5.** The response time of the Apache benchmark, with and without SBBox.

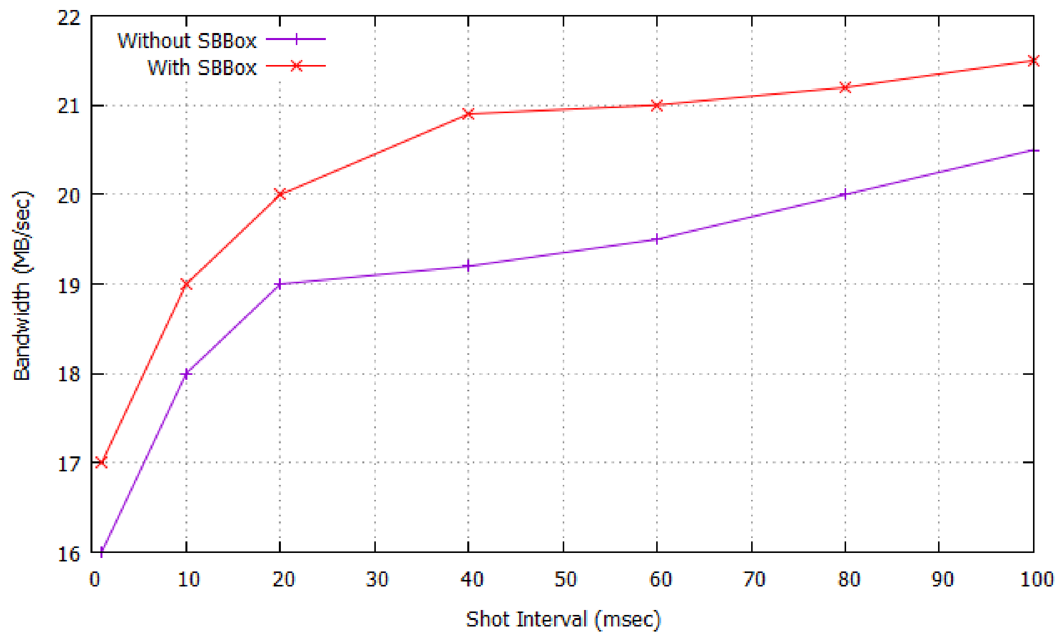


**Figure 6.** The sustained memory bandwidth when the STREAM benchmark is run with and without SBBox. This experiment quantifies the memory overhead (in terms of additional memory bandwidth) when using SBBox.

Secure logging systems have the potential to be secure against not only outsider attacks but also sophisticated insider attacks. For example, the attacker may try to modify the device-driver (helper software). It is generally difficult to figure out the protocol between the host and the hardware device. We believe this kind of attack is hard to achieve.

A DMA attack is a side-channel attack, whereby an attacker can overhear the communication through PCI. The main idea of this attack is that the attacker may reprogram the ROM on the target PCI device to allow the execution of arbitrary code on the device. However, SBBox is safe from this attack because: 1) SBBox does not allow for the reprogramming of its ROM, and 2) since rebooting of the host machine is required to activate this attack, the SBBox reports reboot events to the administrator immediately.

## 4 RELATED WORK

### 4.1 Virtual Machines

One protection mechanism that is available to users of Linux is a Virtual Machine (VM), such as VMWare [20]. A VM is an application that emulates a computer environment in which an operating system and other applications can be executed. If a system running in a VM is compromised, the memory and disk contents can be saved by suspending the VM and making copies of the files that correspond to the memory and disk areas of the virtual system. Some existing VMs save the disk and memory contents in a raw file and others save them in a proprietary format [21, 22].

One may consider using a hypervisor to introspect and protect data stored in the VM [22-25]. Since a hypervisor usually has higher privileges over VMs, it can play a similar role with the dedicated hardware we propose in this work. However, a hypervisor is, itself, software that might be compromised. HyperSentry addresses this problem by using the Intelligent Platform Management Interface (IPMI) [26], which is an

out-of-band channel commonly available to server platforms. A hypervisor monitor is invoked through IPMI and it runs on the host processor. However, the hardware (IPMI) is utilized only for triggering the monitor and the monitor might still be under attack.

Furthermore, a major disadvantage of these approaches is the negative impact over system performance. In general, any system that runs inside an emulated environment will effectively degrade the performance as a whole. At the same time, it will incur increased hardware cost.

### 4.2 Tamper-evident Systems

Another line of work in preventing forensic data from contamination is tamper-evident systems that use cryptographic signatures to add a tamper-evident layer of protection to forensic data. In the techniques presented in [27, 28], the authors proposed an efficient tamper-evident file system using a cryptographic and specialized storage structure (e.g., log-structured, versioning file system, or version-controlled database) to prove that each log event is valid and present. Versioning file systems [29, 30] and log-structured file systems [31, 32] are widely used in tamper-evident storage to keep all changes of the data. A user-centric log archival architecture has been proposed by extending the Simple Object Access Protocol (SOAP) [40].

However, tamper-evident approaches do not protect the data from destruction. If the attacker acquires root shell, he might not be able to forge the data, but he/she can completely destroy it by deleting files, or formatting a whole storage device. As mentioned in the previous section, secure storage should protect the data, not just from altering, but also from destruction. The proposed SBBox solution achieves this goal by using dedicated storage. The attacker who has full control of the host machine cannot change or delete any data in SBBox, because the data is stored in an isolated storage device, which cannot be altered or destroyed by the host machine.

### 4.3 Tamper-resistant Techniques

Chong [33] proposed a tamper-resistant hardware system for secure audit logging. This work provides a hardware-based monitoring system to achieve tamper resistance on the end-user machine. The authors assume that the confidential data (e.g., audit log) is collected and stored securely in a trusted server. The work in [33] is focused on tamper resistance in the end-user environment by using dedicated hardware (iButton), which provides unforgeable time-stamps. The proposed SBBox is complementary to the above work, since SBBox helps to build a trusted server that securely collects, stores, and delivers logs to the end-user. More importantly, SBBox can guarantee that the log is not tampered with in the end-user setup.

### 4.4 WORM-based System

Instead of dedicated hardware, a physically separated server can be employed to store forensic data. A representative example is a logging server [34]. Whenever log data is generated, it is transferred to a central logging server. The central logging server collects and stores the data on hard disks, or on Write-Once-Read-Many (WORM) devices. Since data is usually transferred over the network, it can be tampered with while being transferred. Also, the central logging server can be compromised by attackers. More importantly, if an insider is involved in a cybercrime, the forensic data stored in the logging server may not be trustworthy. Radu et. al. [19] identify the vulnerability of WORM devices. Using off-the-shelf resources, an insider can penetrate storage medium enclosures to access the underlying data. Due to lack of a tamper-resistant mechanism, authentication does not guarantee prevention of this type of attack. By accessing checksum keys, an adversary can construct a new key that cannot be detected [19].

## 5 CONCLUSIONS

This paper introduces the Server BlackBox (SBBox) platform. It is a hardware-based version-controlled storage solution, which extends the concept of append-only storage [1]. While SBBox accepts only append operations, its BlackBox Engine stores differences of consecutive data versions in compressed form. If the administrator wishes to retrieve data, the BlackBox Engine reconstructs any version of the data from the BlackBox DB. As demonstrated with examples, SBBox is aimed at storing forensic data. Nevertheless, it can also be employed to secure any type of reference data, and it can be incorporated into other security solutions. The conducted experiments clearly indicate that SBBox's overhead to the host system is negligible.

## 6 REFERENCES

1. J. Lee, C. Nicopoulos, G. H. Oh, S.-W. Lee, J. Kim, Hardware-assisted intrusion detection by preserving reference information integrity, in: Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing, 2013, pp. 291-300.
2. J. D. McCalpin, Stream: Sustainable memory bandwidth in high performance computers, Tech. rep., University of Virginia, Charlottesville, Virginia, a continually updated technical report. http://www.cs.virginia.edu/stream/ (1991-2007). URL http://www.cs.virginia.edu/stream/
3. A. S. Foundation, Apache HTTP server. URL http://www.apache.org
4. F. Azmandian, M. Moffie, M. Alshawabkeh, J. Dy, J. Aslam, D. Kaeli, Virtual machine monitor-based lightweight intrusion detection, SIGOPS Oper. Syst. Rev. 45 (2) (2011) 38–53.
5. A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, P. Khosla, Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems, SIGOPS Oper. Syst. Rev. 39 (5) (2005) 1–16.
6. X. Zhang, L. van Doorn, T. Jaeger, R. Perez, R. Sailer, Secure coprocessor-based intrusion detection, in: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, EW 10, ACM, New York, NY, USA, 2002, pp. 239–242.
7. N. L. Petroni, Jr., T. Fraser, J. Molina, W. A. Arbaugh, Copilot - a coprocessor-based kernel runtime integrity monitor, in: Proceedings of the 13th conference on USENIX Security Symposium - Volume 13, 2004.
8. J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, M. Rosenblum, Understanding data lifetime via whole system simulation, in: Proceedings of the 13th conference on USENIX Security Symposium - Volume 13, SSYM'04, USENIX Association, 2004.
9. X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. Buchholz, Y.-M. Wang, Provenance-aware tracing of worm break-in and contaminations: A process coloring approach, in: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, ICDCS '06, IEEE Computer Society, 2006, p. 38.
10. J. Newsome, D. X. Song, Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, in: NDSS, 2005.
11. S. T. King, P. M. Chen, Backtracking intrusions, in: Proceedings of the nineteenth ACM symposium on Operating systems principles, SOSP '03, ACM, 2003.
12. A. Goel, K. Po, K. Farhadi, Z. Li, E. de Lara, The taser intrusion recovery system, in: Proceedings of the twentieth ACM symposium on Operating systems principles, SOSP '05, ACM, 2005.
13. T. Kim, X. Wang, N. Zeldovich, M. F. Kaashoek, Intrusion recovery using selective re-execution, in: Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10, USENIX Association, 2010, pp. 89-104.
14. S. Krishnan, K. Z. Snow, F. Monrose, Trail of bytes: efficient support for forensic analysis, in: Proceedings of the 17th ACM conference on Computer and communications security, CCS '10, ACM, 2010, pp. 50-60.
15. K. H. Lee, X. Zhang, D. Xu, Loggc: garbage collecting audit log, in: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ACM, 2013, pp. 1005–1016.
16. Y. Xie, D. Feng, Z. Tan, L. Chen, K.-K. Muniswamy-Reddy, Y. Li, D. D. Long, A hybrid approach for efficient provenance storage, in: Proceedings of the 21st ACM international conference on Information and knowledge management, CIKM '12, 2012, pp. 1752-1756.
17. A. P. Chapman, H. V. Jagadish, P. Ramanan, Efficient provenance storage, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, 2008, pp. 993-1006.
18. Y. Wang, Y. Zheng, Fast and secure append-only storage with infinite capacity, in: Second IEEE International Security in Storage Workshop, Citeseer, 2003, pp. 11–19.

19. R. Sion, Y. Chen, Fighting mallory the insider: Strong write-once read-many storage assurances, IEEE Transactions on Information Forensics and Security 7 (2) (2012) 755–764. doi:10.1109/TIFS.2011.2172207. URL http://dx.doi.org/10.1109/TIFS.2011.2172207
20. VMWare, Vmware gsx server. URL http://www.vmware.com
21. B. Payne, M. Carbone, M. Sharif, W. Lee, Lares: An architecture for secure active monitoring using virtualization, in: Security and Privacy, 2008. SP 2008. IEEE Symposium on, 2008, pp. 233–247. doi:10.1109/SP.2008.24.
22. M. I. Sharif, W. Lee, W. Cui, A. Lanzi, Secure in-vm monitoring using hardware virtualization, in: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS ’09, ACM, New York, NY, USA, 2009, pp. 477–487.
23. B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, W. Lee, Virtuoso: Narrowing the semantic gap in virtual machine introspection, in: Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP ’11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 297–312. doi:10.1109/SP.2011.11.
24. B. Payne, M. de Carbone, W. Lee, Secure and flexible monitoring of virtual machines, in: Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, 2007, pp. 385–397. doi:10.1109/ACSAC.2007.10.
25. A. Seshadri, M. Luk, N. Qu, A. Perrig, Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses, SIGOPS Oper. Syst. Rev. 41 (6) (2007) 335–350.
26. Intel, HP, NEC, Dell, IPMI intelligent platform management interface specification second generation v2.0 (2004). URL http://download.intel.com/design/servers/ipmi/IPMI v2 0rev1 0.pdf
27. D. Molnar, T. Kohno, N. Sastry, D. Wagner, Tamper-evident, history-independent, subliminal-free data structures on prom storage-or-how to store ballots on a voting machine, in: Security and Privacy, 2006 IEEE Symposium on, IEEE, 2006, pp. 6–pp.
28. R. Accorsi, A secure log architecture to support remote auditing, Mathematical and Computer Modelling 57 (7) (2013) 1578–1591.
29. Z. Peterson, R. Burns, Ext3cow: a time-shifting file system for regulatory compliance, ACM Transactions on Storage (TOS) 1 (2) (2005) 190–212.
30. B. Cornell, P. A. Dinda, F. E. Bustamante, Wayback: A user-level versioning file system for linux, in: Proceedings of Usenix Annual Technical Conference, FREENIX Track, 2004, pp. 19–28.
31. F. Douglis, J. Ousterhout, Log-structured file systems, in: COMPCON Spring’89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers., IEEE, 1989, pp. 124–129.
32. T. M. Peters, M. A. Gondree, Z. N. Peterson, Defy: A deniable, encrypted file system for log-structured storage.
33. C. N. Chong, Z. Peng, P. H. Hartel, Secure audit logging with tamper-resistant hardware, Springer, 2003.
34. Splunk, Splunk syslog server. URL http://www.splunk.com/Syslog
35. Z. Ouyang, N. Memon, T. Suel, D. Trendafilov, Cluster-based delta compression of a collection of files, in: Web Information Systems Engineering, 2002. WISE 2002. Proceedings of the Third International Conference on, 2002, pp. 257–266. doi:10.1109/WISE.2002.1181662.
36. A. Chuvakin, Ups and downs of UNIX/Linux host-based security solutions, The USENIX Magazine.
37. Xilinx, Xilinx zynq-7000 all programmable SoC ZC706 evaluation kit. URL http://www.xilinx.com
38. Xilinx, Xilinx zynq-7000 all programmable SoC. URL http://www.xilinx.com
39. Xilinx, Xilinx zynq-7000 petalinux tools. URL http://www.xilinx.com/tools/petalinux-sdk.htm
40. T. Shitamichi, and R. Sasaki, "A Proposal and Evaluation of User Centric Trusted Log Archival Architecture", International Journal of Cyber-Security and Digital Forensics (IJCSDF), Vol. 4, No. 3, 2015, pp. 442-452
41. S. Hou, S. Yiuy, T. Ueharaz, and R. Sasakix, "A Privacy-Preserving Approach for Collecting Evidence in Forensic Investigation," International Journal of Cyber-Security and Digital Forensics (IJCSDF), Vol. 2, No. 1, 2013, pp. 70-78