Architectural Support for Lease-Regulated Secret Data

Lanfang Ren, Hongtao Bai, Fei Liu
Editorial Department, Institute of Security China Mobile,
Beijing, China
lanfangren@gmail.com

Abstract— **Contemporary cloud software stack is large and complex, where security vulnerabilities are routinely discovered. Hence, it is hard or even impossible to place trust on such a fragile platform to process some security-critical data. In this paper, we propose an architectural solution that extends processors with cryptographic support and policy enforcement engine to regulate the usage of security-critical data in an untrusted cloud platform. Specifically, we make a specific case of using such a platform for protecting the secrecy of lease-regulated data, which should only be accessed within a predefined amount of times or before a specific date. We further discuss how such protection is necessary in several security-critical usage scenarios.**

Keywords-Security, Data Security

I. INTRODUCTION

In the past few years, we have witnessed the commercial successes of a number of cloud computing companies such as Amazon [1], SafeForce, RackSpace, Google and Microsoft. With the continually spreading of cloud computing into more serious application domains, it is no surprise to see cloud users are now increasingly put their private data into the cloud platform.

Unfortunately, with the growing popularity of cloud computing, the security guarantee of typical cloud platform is not going towards better, but actually becomes worse than ever before. On on hand, major cloud vendors usually leverages a stack of commodity software to build their platforms. For example, Amazon leverages Xen as the hypervisor, a Linux-like manage virtual machine, as well as a large number of user-level software written in python, ruby and C. This essentially forms a virtually unbounded trusted computing base where security vulnerabilities are discovered in a daily base [4]. On the other hand, with the inexperienced and/or even malicious cloud operators managing such a complex platform, it is very easy for an operator to divulge users' secret data, either intentionally or unintentionally [14]. Hence, it is no surprise to see that Gartner has ranked "privileged operators" as the top threat to cloud security [6].

On the other hand, there are usually some data that should be processed in a cloud platform, partially due to the massively available computing resources. In many cases, such data only needs to be accessed within a limited number of time or before a specific date and time. However, with the complexity of the cloud platform and virtually unbounded trusted computing base, it is hard or even impossible to ensure users' data access policies using existing cloud regulating software. Hence, an attacker may easily violate such policies by arbitrarily extend the access duration and times for such data.

To address this issue, this paper proposes an architectural approach that extends processors with cryptographic logics and a policy enforcement engine to ensure users' secret data is accessed while strictly respecting user-specific security policies. To this end, we use the notion of lease-regulated secrete data (LRSD for short) to describe a piece of data that requires such regulation. LRSD is encapsulated together with the code that is allowed to access the secrete data, which is self-contained such that the code cannot invoke external code or reference external data in order to complete its functionalities. LRSD is encrypted and signed by a data provider, with access policies being embedded inside.

We propose to extend processors with cryptographic logics that can decrypt LRSD only in on-chip caches such that the secrete data will never exist in plain text in external memory. The processor can also use its private key to verify originality of an LRSD and determine whether it is able to access such data.

Currently, LRSD is built with three kinds of policies. First, it contains a list of processors that can access the secret data. A processor not in the list cannot access the data. This also regulates the communication of two LRSDs running on two machines. Second, it can be embedded with the exact amount of times that it could be accessed. Here, the number of times is the number of times that the LRSD code module could be invoked, not how the processor reference the secrete data. Third, LRSD can be embedded with policies regarding before or after a specific date when LRSD can be accessed. The deadline of access will be judged using the internal timestamp counter inside the processor such that an adversary cannot fake a timestamp to arbitrarily extend the lifecycle.

Protecting LRSD by a secure processor can reduce the trusted computing base (TCB) to the processor. This can significantly increase the trustworthiness of a cloud platform. To illustrate the potential usage of LRSD, we further use two cases such as secure coupon accesses and decalcification of some secrete data until a specific date.

II. RELATED WORK

The notion of lease-regulated secret data has some similarities with policy-sealed data [11]. However, policysealed data requires trusting the cloud platform to ensure such policies, while LRSD is protected by processors. Further, the policies in [11] mostly concern with where a specific piece of data can be accessed, while LRSD provides additional policies regarding the lifecycle of secret data.

LRSD is also related to self-destruction data in vanish [5]. However, the policies in LRSD is more expressive than that in vanish. Further, vanish cannot protect against malicious operating systems but purely relies on the churn characteristics in distributed hash tables to ensure data destruction.

The secure processor design is related to prior work on secure processors [8], [9], [12], [7]. For example, Bastion [2] and SecureME [3] have also extended processor with security functionalities to protect against hardware attacks. A most recent work, called HyperCoffer [13], extend processors with security and integrity protection engines such that all data goes out of on-chip cache will be encrypted and hashed. However, these designs target at a complete different levels and granularities of protection. For example, The protection granularities of SecureME and HyperCoffer are at the processes and virtual machines accordingly. Bastion, instead, need to trust the hypervisor to securely protect a module. More importantly, they do not provide the lease policies as that in LSRD.

III. Design

We use secure processor to protect user's secret data on the cloud, and further extend the processor to support extend memory, which could be used to store persistent data across executions. By adopting the secure processor substrate, the system is able to enforce policies defined by user that are used to constrain the access of specific data.

A. *Hardware Architecture*

Secure processor leverages hardware encryption and hash tree to protect the privacy and integrity of specific data. It has two modes: secure mode and normal mode. In normal mode it executes just as a normal processor, while in secure mode, all of the data outside the chip is considered as encrypted will be decrypted as long as being fetched into the cache. Thus, only the processor is trusted, and no external device including the memory and peripherals need to be trusted, which could significantly reduce the TCB. Meanwhile, each memory update will recalculate the corresponding merkle hash tree. When the data is loaded into the cache, the integrity checking engine will recalculate the hash of the cache line to verify whether the data has been illegally modified or not. Therefore, a malicious software, e.g., the OS, cannot modify the protected data, otherwise an alarm will be triggered during hash checking. Rogers et al. proposed AISE (Address Independent Seed Encryption) and BMT (Bonsai Merkle Tree) [10] based mechanisms to implement a efficient secure processor, which introduces less than 5% performance overhead. We use these two technologies as the foundation of our design.

In this paper, we extend the secure processor with following components, as shown in table 1. The KP is unique for each processor and is considered to be safe since it is embedded inside the hardware. The corresponding public key is maintained by the manufactory, and the user is able to access the list of public key for verification. The NVR and EMR are two registers. The LRSD-Table is used to save the information of installed LRSD.

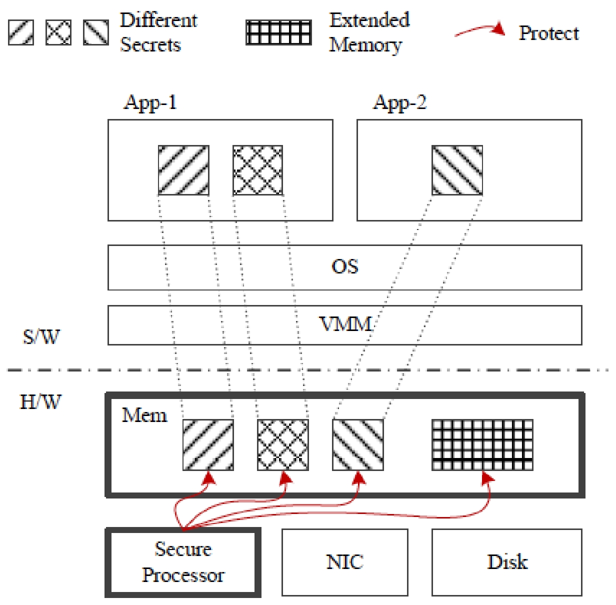

Figure 1. Example of a ONE-COLUMN figure caption.

TABLE I. Components of Secure Processor

Name	Description
K_P	Private key embedded in the processor, which is unique.
NVR	Non-Volatile Register with 128 bits.
EMR	Register saving the start address of the extended memory.
LRSD-Table	The table saved all of the installed LRSD.

B. *Extended Memory*

It is required that some data needs to be stored in non-volatile memory safely. For example, the counter that record how many times the data has already been used. One way is to save the counter directly inside the secure processor, using NVR . However, the solution is not scalable and has low performance, since the number of NVR is limited and is slow to write. In this paper, we propose a novel way to extend memory to support store data in a non-volatile and secure way.

During initialization of secure processor, the OS will assign a region of memory as extended memory, with the start address of the memory saved in EMR . The content of the memory is loaded from the disk, whose hash must be identical to the value saved in NVR . The OS is responsible to keep the consistency between the data in memory and on disk in order to tolerant machine crash. The extended memory can only be written by the protected code within the LRSD.

The process of extend memory updating is as follows: (1) the protected code writes a new value to the extend memory. (2) the processor recalculate the hash value of the memory and trigger an interrupt to notify the OS. (3) the OS will save the updated memory on the disk in the interrupt handler. It is noted that OS will save the data into a shadow copy to avoid overwriting. (4) after that, the processor write the new hash of the updated extend memory to NVR and continue to execute. Next time during initialization, the OS is responsible to load the data stored in the disk to the extend memory. The processor will check the hash of the loaded extend memory to ensure its integrity.

The design of extend memory is capable to tolerant machine crash at arbitrary time, thanks to the disk storage and the shadow copy mechanism. We consider the write operations are rare during execution, thus the performance overhead introduced is not significant.

C. *LRSD Abstraction*

An LRSD includes following components:

- Secure vector: The vector is used to install the LRSD to the secure processor.
- Code section: This section contains all of the code of the LRSD.
- Data section: This section includes all the related data.
- Metadata section: The metadata involves information about the LRSD.
- API header: The API header involves the offset of functions provided by the LRSD.

In order to install the LRSD on the secure processor, a secure vector must be used as parameter. The vector contains the encryption key of the protected LRSD, aka., K_{LRSD}. The vector itself is encrypted by the public key of the target secure processor, thus only the specific processor could decrypt the vector to get K_{LRSD}. The data of LRSD could be accessed by its own code, since the data structure of LRSD is only available to the protected code. The metadata section contains information about the LRSD, including its hash value and LRSD size. All of the code, data and metadata sections are encrypted by K_{LRSD}, while the API header is in plaintext which is used for other parts of the program to invoke functions provided by the protected LRSD. The code inside LRSD cannot call the functions outside, so it should be a closure of itself.

D. *Secure Loader*

In order to load an LRSD into the processor and memory, we introduce a component named secure loader. The loading process is as following: First, the LRSD is loaded into the memory. Then the loader invokes an instruction, lrsd_install, to install the LRSD into the processor. The processor loads the secure vector to get the K_{LRSD} , allocs a new entry in LRSD-Table to save the start address and size of the LRSD. The address range is used for the processor to automatically switch between secure and normal modes. After that, the loader will load API header so that other components can call the functions provided by LRSD.

E. *Policies Enforcement*

We leverage LRSD to implement following three policies:

Limiting the processors : As mentioned previously, an LRSD can only be installed on a secure processor if and only if the user constructs a secure vector for that processor. Otherwise the processor cannot get K_{LRSD}, and thus has no way to decrypt the data or code. Therefore, a user can control the list of processors that is capable to access LRSD by generating corresponding secure vectors for each processor.

Limiting the access times: Since the data of an LRSD is only accessible to its own code, the user can add a counter in the accessing code. As long as the code is executed, it increases the counter. When the counter has expired some threshold, the code will deny any further accessing. In order to keep the counter persistent, the processor will save the counter inside the LRSD, with the process similar to extend memory writing. Thus, even an attack use another secure vector to run the code on another processor after counter expiration, he still cannot access the data.

Limiting the period of accessing: When accessing the data in LRSD, the corresponding code can get the current time and check whether it is within a certain period. If the time is without the period, then it could deny any access. In order to ensure that the timestamp is not faked, the code should embed a public key of a trusted time server, and send a nonce to prevent replay attack. Each time when it gets a timestamp, it sends the nonce to the trusted time server, who will sign the timestamp as well as the nonce. In this way, the freshness of the timestamp could be ensured.

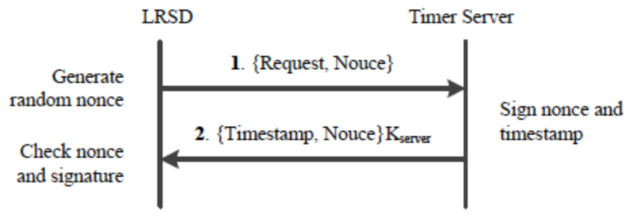

Figure 2. The time server protocol

IV. USAGE CASES OF LRSD

This section describes two potential usage cases of LRSD. The first one is using a secure coupon system and the second one is using a secure declassification of a secure.

A. *Securing Coupons using LRSD*

Providing coupon is a popular means for merchants to spread their services or goods. However, under a cloud-based e-commerce web-site, the server managing coupons may be tampered with by a cloud operator in order to gain financial advantages by providing unlimited coupons with significant lower prices to end users. For example, several mobile network operators have faced issues that some internal operators collude together to sell unlimited internet access coupons to end users, causing financial loss in the companies. Using LRSD, a network operator can secure the coupon distribution by encapsulating the coupon decision code and the coupon list as LRSDs. As the secure processor will faithfully enforce the access policies of the coupon data, even a cloud operator cannot tamper with LRSDs.

B. Secure Data Declassification

Listed companies usually need to regularly disclose their financial data to the public, which sometime will incur significant impact over the stock market. It has long been an issue that some insiders can get the financial data earlier than the supposed date. These insiders can leverage such data to take action on the stock market to make money. A potential approach to mitigating such issue is that the regulation officers can request the chief financial officer in listed companies to encapsulate their data using LRSD and specify that the LRSD can only be accessed after the disclosure date. By this means, even if the financial data encapsulated using LRSD will be spread across different parts, they cannot access such data before the designated date.

V. CONCLUSION

This paper analyzed the security threats of commodity cloud platforms. To address low trustworthiness due to unbounded trusted computing base, this paper proposed lease-regulated secret data (LRSD for short) that encapsulated secret data together with the code as well as access policies together. To ensure the security of LRSD, this paper also described a secure processor design with cryptographic logic and policy enforcement engine. Using two compelling cases, this paper demonstrated the potential usefulness of LRSD.

ACKNOWLEDGMENT

This paper is supported by 2012 National Science and Technology major projects "The next-generation broadband wireless mobile communications network", Titled "cloud computing application mode mobile Internet security research" (No. 2012ZX03002002).

REFERENCES

[1] Amazon Inc., “Amazon Elastic Compute Cloud (Amazon EC2),” http://aws.amazon.com/ec2/, 2011.

[2] D. Champagne and R. B. Lee, “Scalable architectural support for trusted software,” in IEEE Symposium on high performance computer architecture. IEEE, 2010, pp. 1–12.

[3] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, “Secureme: a hardware-software approach to full system security,” in International conference on Supercomputing. ACM, 2011, pp. 108–119.

[4] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, “Breaking up is hard to do: security and functionality in a commodity hypervisor,” in ACM Symposium on Operating Systems Principles. ACM, 2011, pp. 189–202.

[5] R. Geambasu, T. Kohno, A. A. Levy, and H. M. Levy, “Vanish: Increasing data privacy with self-destructing data.” in USENIX Security Symposium, 2009, pp. 299–316.

[6] J. Heiser and M. Nicolett, “Assessing the security risks of cloud computing,” http://www.gartner.com/DisplayDocument?id=685308, 2008

[7] R. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, “Architecture for protecting critical secrets in microprocessors,” in Proc. ISCA, 2005, pp. 2–13.

[8] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural support for copy and tamper resistant software,” in Proc. ASPLOS, 2000, pp. 168–177.

[9] D. Lie, C. a. Thekkath, and M. Horowitz, “Implementing an untrusted operating system on trusted hardware,” in Proc. SOSP, 2003.

[10] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, “Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly,” in Proc. MICRO, 2007, pp. 183–196.

[11] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, “Policysealed data: a new abstraction for building trusted cloud services,” in USENIX Security Symposium. USENIX Association, Aug. 2012.

[12] G. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, “AEGIS: architecture for tamper-evident and tamper-resistant processing,” in Proc. Supercomputing, 2003.

[13] Y. Xia, Y. Liu, and H. Chen, “Architecture support for guesttransparent vm protection from untrusted hypervisor and physical attacks.” in IEEE Symposium on high performance computer architecture, 2013, pp. 246–257.

[14] F. Zhang, J. Chen, H. Chen, and B. Zang, “Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization,” in ACM Symposium on Operating Systems Principles. ACM, 2011, pp. 203–216.