Protecting cryptographic keys on client platforms using virtualization and raw disk image access

Sujit Sanjeev
Goldman Sachs Group Inc.

Jatin Lodhia
Google Inc.

Raghunathan Srinivasan
Arizona State University
raghunathansrinivas@gmail.com

Partha Dasgupta
Arizona State University
Tempe, USA
partha@asu.edu

Abstract— **Software cryptosystems face the challenge of secure key management. Recent trends in breaking cryptosystems suggest that it is easier to steal the cryptographic keys from unsecure systems than to break the algorithm itself, a prominent example of such an attack is the cracking of the HD-DVD encryption. This paper presents two methods to hide cryptographic keys in an unsecure machine. The first method uses virtualization to isolate the sections of memory that contain cryptographic keys from an untrusted guest operating system (OS). Virtualization is an effective method to provide isolation between trusted and un-trusted components of a system. This work makes the Virtual Machine Monitor (VMM) as a cryptographic service provider for guest OS. The second method provides techniques to securely retrieve and store keys in secondary storage. The information about key storage and retrieval is stored inside the application binary. On execution this section retrieves the key from secondary storage.**

Keywords-component; Key hiding; Secret Hiding; Virtualization; Raw disk interface; Lguest; Linux

I. INTRODUCTION

Computers are an integral component of enterprises, government establishments and consumer computing. They are used to carry out a variety of critical and sensitive computations. Secure computing models have evolved to use advanced cryptographic techniques such as digital signatures and high end encryption. Protocols such as Secure Sockets Layer (SSL) provide the consumers with a sense of security by encrypting all communications. These protocols use encryption to secure communication among machines, but provide no protection for the platforms themselves that execute them. A machine can get compromised in a variety of different ways. A user may knowingly or unknowingly execute unverified applications on the computing platform. It is estimated that fault density in programs can range from 2 – 75 per 1000 LOC [1]. In addition OS consists of many device drivers, which have error rates much higher than kernel code [2]. Malware writers attempt to exploit existing bugs to compromise machines in an attempt to steal secret data. Secret data may be bank account information, credit card information, social security numbers, passwords, and so on. Such information can be stored and exchanged securely using cryptography. Browsers use digital certificates which have keys embedded in them. Software like Truecrypt [3] and BitLocker [4] allow users to store data in an encrypted form on secondary storage. Storing encrypted data on computers or secondary storage merely changes the vulnerable point in the system. Instead of targeting the data directly, malware writers can target the encryption keys. A breach of trust in the machine can lead to a malware stealing keys. The use of cryptography in computer software has evolved to a major extent and the stakes are high for the users if their trust is compromised.

Keys can be stored on external tamper resistant hardware devices such as smart cards [5]. The device has a processing chip that can perform cryptographic tasks such as encryption and digital signatures. Though hardware based cryptosystems provide better security, they are not feasible for all classes of end users due to various deployment issues, hence software based key hiding systems are required. Software based techniques operate within the realms of the OS and resort to techniques like scattering, key stores and so on. In any of these systems, the cryptographic key has to be brought into system memory for any crypto operation. This can provide an attacker a window of opportunity to steal keys. Once a key is stolen, all encrypted operations using this key can be compromised. The most notable example of this kind of attack is the cracking of HD-DVD encryption [6, 7].

This paper presents two methods to preventing malware from stealing keys from computing platforms. The first involves the use of virtualization where the property of memory isolation is used to separate key storage from system memory. With this model, all applications execute in the guest OS, while software cryptosystems run in a separate and secure hardware protection domain. This provides higher levels of isolation from the untrusted parts of the system. The key is stored in the Virtual Machine Monitor (VMM), outside the domain of the guest OS. All cryptographic operations are performed by the VMM, the guest OS requests crypto operations by executing the appropriate hypercalls. As long as the VMM is not compromised by the guest, the keys stay secure and protected from the guest.

The second method uses accesses to raw disk image for hiding keys on secondary storage media. To effectively hide the key on secondary storage, the keys are directly written by an application to the unused sectors of files on the secondary storage without using the file system calls of the OS. The key bits are scattered all over in a pattern which is unique to every installation of the cryptographic application. This information is generated during installation time and is hard coded into to application binary as a key fetch block. The key fetch block is padded with 'junk' instructions; additionally the location of the key fetch block is different in each installation. This ensures that an attacker cannot use information gained by analyzing one instance of the application to steal keys from another installation of the application. It must be noted that hardware methods to acquire cryptographic keys are difficult to protect against as they can provide the exact contents of the RAM and even the HDD image. The works presented in this paper only protect against methods based in software to steal keys.

The rest of this work is organized as follows, section 2 discusses the related work in the area of key storage and key stealing, section 3 discusses threat model for both methods, section 4 describes the implementation of VMM based key storage, section 5 describes disk secondary storage based key hiding, section 6 concludes the paper.

II. RELATED WORK

A cryptographic key has to be stored somewhere in memory prior to the crypto operation. Due to this it is difficult to prevent attacks that analyze memory layout to steal keys. It has been shown in practice that without secure clearing of data, information from kernel memory can survive periods of days even if the system is in continuous use [8], in addition many regions in the system do not respect the lifetime sensitivity of the secret data [9]. This can allow pattern matching techniques to easily extract secret keys. One such method performs reverse engineering including reconstruction of process lists and data structures within the process, which hold cryptographic keys [10]. Since all binary images of a program are the same, an attacker can extract information about programs from one binary and use it to attack binaries installed on other machines. Another method to locate cryptographic keys in large data sets uses the property that the keys have higher entropy than other data in programs [11]. The same work also describes a method for finding RSA secret keys stored on hard drives. Physical memory forensics such as *Volatools* [12] allow attackers to extract encrypted material from memory dumps even after it is encrypted using commercially available software TrueCrypt. This is achieved by analyzing the symbols exported by *Device Mapper* kernel module. These symbols are parsed to find kernel objects and symbols recursively till the *master_key* member of *CRYPTO_INFO* object is found. In [13] a comprehensive list of various techniques to extract information from memory dumps is presented. The work presents that it is difficult to extract data from a raw snapshot of RAM, however intelligent analysis such as inspecting for keys/secrets at address offsets of well known programs can lead to better results.

Centralized key storage employs techniques where the generation, storage, distribution, revocation and management throughout the lifetime of the key happens on a single central machine. This offers advantages such as ease of backup, recovery, and securing a single machine secures the keys. One such commercially available solution is Thales keyAuthority [14]. In Distributed key storage, keys are stored on different machines. The key management system takes care of using the disparate keys to achieve the required cryptographic function. One such system developed by IBM is the Distributed Key Management System (DKMS) [15]. Reducing the number of key copies present in memory is another method to protect cryptographic keys from memory disclosure attacks [16]. It avoids caching of the key by the operating system and also disallows swapping of the memory area of the key. The authors conclude that though the number of copies of keys present is reduced, once a sufficiently large memory dump is obtained; there are high chances that it would contain the single copy of key. They suggest that in order to eliminate leakage of key via memory disclosure attacks, consumers would have to resort to special hardware devices.

The authors in [17] present the concept of networked cryptographic devices which are resilient to capture. It is a server aided protocol in which the networked nature of the device is exploited. The cryptographic operations are performed by interactions with a remote server. In this way, even if the networked device where the cryptographic operations are performed is compromised, the attacker cannot derive the entire key from it. This scheme is useful for large firms, but not useful for end users who may not have access to networked cryptographic devices. All or nothing transforms have also been used to store the transform of the key in memory [18]. This uses the concept that the actual key is not stored directly in memory, only its transform is stored, and the transform is such that by obtaining partial bits of the transform, an adversary cannot determine the actual key. The drawback with this scheme is that for performing the cryptographic operation, the key has to be eventually brought to memory, and malware can access the key at that instant.

III. THREAT MODEL

A. *Threat Model for VMM based key storage*

The VMM and the Host OS will not interact with the outside world directly, and only the Guest OS will do this operation. The main functionality of the Host OS is to run the virtual machines and support the VMM. This allows the VMM and the Host OS to be the trusted components in the system. This includes the assumption that the attacker cannot modify the VMM by tampering with or replacing the CPU, memory or disk. The guest operating system is assumed to be vulnerable; it could be mounted with attacks such as rootkits, keyloggers, and so on. It runs all the end user applications such as web, ftp and email servers, which are

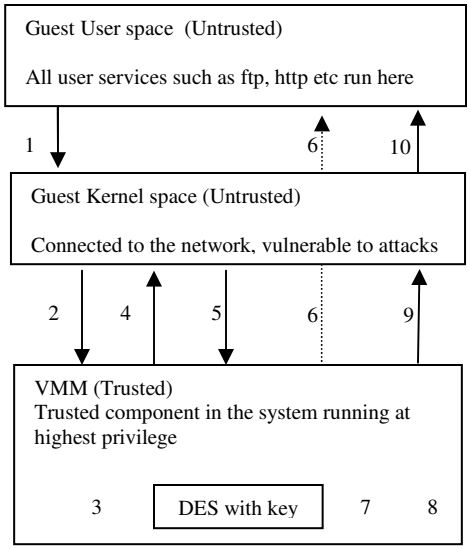


Figure 1. Key storage and attestation model

vulnerable to standard attacks. Given the vulnerability, an attacker may attempt to obtain the physical memory dump of the guest to observe cryptographic keys.

B. Threat model for disk image based key storage

The installation of the key storage software happens on a clean computer. If there is already the presence of a virus or a root kit on the machine, then the virus may be able to hinder the installation process, make a patch on the software being installed or obtain the key directly from the installer as the installer program is not secure. This will render the software ineffective. Once the software is installed the computer can get infected with malware. Attackers will attempt offline analysis of the software for finding its vulnerabilities and information about how/where it stores the keys and will use this information to crack any other installation. Any malware attempting to break the system does processing locally only, i.e., it does not send entire memory/disk to remote site. Finally, no extra hardware is available for securing the key.

IV. VMM BASED KEY STORAGE

Memory isolation is a key property of virtualization leveraged to safely store cryptographic keys in this solution. This provides a feature that programs running inside a guest VM cannot access or modify the data and programs running in the VMM or other guest VMs. All cryptographic routines used to perform encryption and decryption are provided by the VMM to the guest OS. The VMM receives the plain text or the encrypted cipher text from the guest OS and returns the encrypted or plain text correspondingly. The cryptographic key is never released to the guest OS. This prevents attackers from obtaining information about the key using analysis of the guest OS. This scheme also implements a remote attestation scheme where the VMM verifies whether the application requesting the cryptographic procedure is a legitimate program or not. In addition to preventing malicious program from requesting cryptographic operations, this also thwarts attacks like code injection where a malicious program can inject code on the guest OS component which normally requests the cryptographic operations.

The attestation process is divided into two steps, in the first step the VMM injects a Trusted Helper Program (THP) in the kernel of the guest OS from which the cryptographic request originated. The THP locates the memory and pages of the user space program which requests the cryptographic operation. This model ensures that only the designated program to execute cryptographic operations requests it. The key used to perform the cryptographic operations is stored in the hypervisor module along with the crypto routines. Figure 1 shows the pictorial representation of each of the steps involved in this scheme. In step 1 the guest user space program issues requests for cryptographic operations such as encryption, decryption, and signatures by executing a new system call '*security*'. The data on which the cryptographic operation is to be performed is passed as a parameter to the call. The system call executes a software interrupt which causes the execution to context switch from the guest user space to the guest kernel space. In step 2, the guest kernel forwards the cryptographic request to the trusted VMM using hypercalls. The hypercall executes a software interrupt which causes the control to be transferred to the hypercall handler. In step 3, the secure VMM receives the hypercall to perform cryptographic operations and in response generates code to be injected ($\mathbf{C}_i$) on the running guest kernel.

$\mathbf{C}_i$ is responsible for obtaining the guest physical address where the user space program that executed the security system call is present. It is also responsible to get the user space process's pages into memory, in case they are swapped out. Generating $\mathbf{C}_i$ ensures that it can be randomized for every attestation instance ensuring that the operations performed by it cannot be predicted by any malicious logic. In step 4 the guest kernel executes $\mathbf{C}_i$ which returns the address of the page where the user space program resides. Control is transferred back to the VMM in step 5. In step 6 the secure VMM reads the contents of the user space program directly from memory using the address obtained in step 4. In step 7 the secure VMM computes and compares the hash value of the memory content to pre-computed hash values obtained from the original image of the program. In step 8 the VMM performs the requested operation using the cryptographic key. In steps 9 the results are written back to the guest kernel, which writes the result into the user space program in step s10.

Implementation: The VMM was implemented using Lguest which is a linux based hypervisor. The Linux kernel used for implementing the framework was 2.6.23-rc9 on Ubuntu. The same kernel build was used to create the guest and the host OS. QEMU was used to install Ubuntu on a disk image for the guest OS. A new entry '.long sys_security' was added in the '/usr/src/Linux/arch/i386/kernel/syscall_table.S' to create a system call 'security' for the guest kernel. Similarly, the

TABLE I. OPERATING TIMES ON GUEST OS

Time for one round of encryption (micro sec)	Time for injection and attestation (micro sec)
31	9

'unistd.h' file was changed to show the new security call '#define __NR_security 325', and the number of system calls was changed to 326. In 'Linux/syscalls.h' a line 'asmlinkage int sys_security (char *input, char *output, int operation)' was added to declare the system call. The security system call was implemented to invoke two hypercalls, hcall(LHCALL_ENCRYPT,__pa(k),__pa(q),0), hcall(LHCALL_DECRYPT,__pa(k),__pa(r),0). __pa is a kernel routine used to convert virtual addresses to actual physic al addresses. This macro has to be applied to the pointers being passed since they actually point to addresses within the guest address space and hence the VMM needs the actual address location to read and write to that address.

TABLE II. OPERATING TIMES ON HOST OS

Time for one round of encryption (micro sec)	Time for crypto operations (micro sec)	Host communication (micro sec)
8	5	3

The hypercall transfers control from the guest kernel to the VMM. The hypercall uses the kernel routines copy_to_user and copy_from_user to perform data transfer on these addresses. The two hypercalls were implemented to perform the major cryptographic API operations invoked. crypto_alloc_ablkcipher initializes a cipher transform for DES. crypto_ablkcipher_setkey sets the key to be used during the operation, the key is declared in the VMM. crypto_ablkcipher_encrypt and crypto_ablkcipher_decrypt are calls to the encryption and decryption routines of the cryptographic API framework which in turn invokes the DES encrypt/decrypt routines. crypto_free_ablkcipher and ablkcipher_request_free are used to perform the cleanup operations. The guest kernel is created with a empty space routine filled with 'nop', when the guest kernel makes the cryptographic hypercall, the hypervisor injects code at this location. The injected code provides the address of the user space program that called the 'security' system call to the hypervisor. The hypervisor obtains the memory addresses and reads the contents, hashes them, and compares it to a pre computed hash value to determine if the user program executing the cryptographic system call is the designated application to perform it. If the program is indeed the legitimate program, the hypervisor performs the required cryptographic operation and returns the results to the guest kernel, which in turn returns the results to the guest user land application.

The system was implemented on machine running on a Intel® Pentium® dual core processor with 1GB Physical RAM. The time taken for cryptographic operations on the Host OS is shown in Table 1. The data encrypted was 8 bytes long, and the algorithm used was DES. One round of encryption on the Host OS takes 8 microseconds, the actual cryptographic operational time was measured to be 5 micro seconds, and hence the communication and data overhead inside the Host is estimated to be 3 microseconds. Table 2 shows the times as measured by the guest process. Given that one round of cryptographic operations take 5 micro sec on the Host, and the time for attestation and code injection is measured as 9 micro seconds, the estimated kernel switches and hypercalls can be estimated to be 17 micro seconds (31 – (9+5)). This is understandably higher than the host, since guest systems are slower due to mechanisms such as shadow page tables.

V. STORING KEYS IN SECONDARY STORAGE MEDIA

This section presents a technique to store encryption keys in secondary storage media without the use of external hardware and without using any password based encryptions schemes. This scheme cannot prevent attacks that steal keys from RAM. RAM based attacks are addressed in the virtualization based key storage approach previously described. This scheme is aimed at preventing attacks which identify location of keys in the secondary storage and steal them. To mitigate the problem of keys being stolen from the running copy of a process (as in HD-DVD encryption), this scheme incorporates randomization of the entire binary ($\mathfrak{p}_{crypto}$) which performs the cryptographic operation. The location inside $\mathfrak{p}_{crypto}$ where the key is stored is modified for every installation of $\mathfrak{p}_{crypto}$. This implies that the location of the key will be revealed to the attacker (Mallory) only if the system is compromised, however, it still prevents Mallory from using this information to steal keys from a different installation of $\mathfrak{p}_{crypto}$. This serves to decrease the 'return of investment' for Mallory, as the entire analysis must be performed on every instance of attack. In conventional off the shelf binaries every copy of the binary is the same, which makes stealing of keys simpler for Mallory. Mallory has to analyze the structure of the off the shelf binary in a controlled environment and determine which locations in its memory cryptographic keys get loaded. Once this is determined, this information can be used to steal keys off every other installation of the binary.

The cryptographic keys are themselves stored in the unused portions of the last sectors of files on the file system without using the file open system call of the operating system. Within this storage space the key bits are scattered all over in a pattern which is unique to every installation of the software. The pattern is generated during installation of $\mathfrak{p}_{crypto}$. The keys are written to the raw disk using the '/dev/' interface in Linux. The key is scattered in a storage area in a random pattern which is different for each installation, the storage area is called *scattered array*. A special block of code is generated during installation to retrieve the key stored in the *scattered array*. The execution of this code results in the location of the *scattered array* being generated. The locations are then used to retrieve the key. The block of

code generating the location of the *scattered array* is called the *storage fetch block*. The portion of code generating the location information is called the *bit-address fetching block*. Thus the root of trust here is the hard coding of fetching information (both about the location of the storage on disk and the exact pattern of scattering in the area designated for storage) in the code section of the executable. To make sure that there is no underlying pattern matching the fetching codes for different installations; these blocks of codes are generated randomly and are padded with random "junk" code for obfuscation. Additionally, the location of this fetching code within the software is different for each installation which makes it even more difficult to generate an attack scenario which encompasses multiple installations of $\mathfrak{p}_{crypto}$. To make sure that the attacker cannot exploit the software by hijacking the fetching code to execute in a controlled environment, we use *self attestation* inside the *bit-address fetching block*.

Fig. 2 shows the structure of the modified binary after the installation process is complete. The installation suite ($\mathfrak{l}_s$) generates the locations on the raw disk to store the *scattered array*. $\mathfrak{l}_s$ next generates the *storage fetch block* and inserts it inside $\mathfrak{p}_{crypto}$. $\mathfrak{l}_s$ also places a call to the *storage fetch block* to generate the required inode/sector information where the scattered array is present. The scattered array is fetched to the program memory prior to cryptographic operations when the *bit-address fetching block* is executed. Since the *bit-address fetching block* can generate the location of the key, Mallory will attempt to locate this block. The location of the *bit-address fetching block* is obfuscated by removing it from sequential flow, the block is reached after the execution of a JMP statement whose target is generated by a block of code. At the end of the block, the location jumps back to the cryptographic procedure.

There are two possible techniques to store the keys on the disk. The first method tried out during this research was to remove a block from the free list of OS file system and use it for storage. However, utilities like *fcsk* found out orphan disk blocks and put them back in the free list. Due to this, no claim can be made to how long these files will remain pristine. Instead, this research implemented a technique of storing keys on the last disk block sector of some files. The OS always allocates a fixed chunk of disk area to be used for a newly created file. It does not reclaim the space beyond the end of file (EOF). The space beyond EOF on this sector can be used to store the key. In order to make sure that this file is never changed or appended, special files are created during the installation of the software and the user is asked to assign names for these files.

We do not assign software generated random names to the files containing the key. This is because random software generated names either have patterns which can be easily detected or are too random to give away the fact that it is not human readable. As the user knows that this file will be used to store the keys, it can be assumed that the user will not modify it. By permuting characters present in the names given by the user, the installation suite generates many

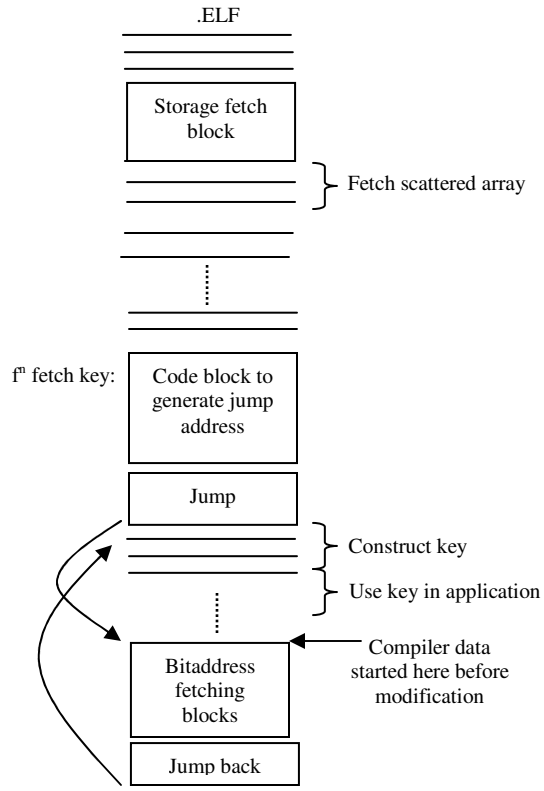
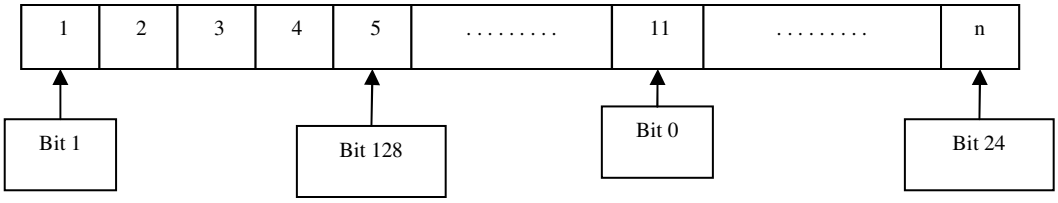


Fig 2. Structure of modified application $\mathfrak{p}_{crypto}$

dummy files (in the order of hundreds) in addition to the key storing file. This will make it difficult for the malware to know which file has the key even if the malware is able to track the disk reads executed by the software. As the names of the key bearing files which were given by the user, are a subset of the permutation, it is difficult to distinguish the user made file from the rest of the machine made files.

Each file created is filled with random data to allow 200 to 400 bytes of space left in the last blocks of the file after the EOF. This space is used to store the cryptographic keys. Figure 3 shows an instance of a scattered array. The location of a bit of the key in the scattered array is termed as the *bit-address* in the rest of this work. Every bit of the key has a *bit-address* associated with it which forms the *bit-address pattern*. The bit-address pattern is unique to every installation of the key storage system. If bit 0 of the key is present at the 11^{th} position of the scattered array then the bit-address of bit 0 is 11. The bit address pattern for the figure will be 11, 1, …, n (24^{th} bit), …, 5 (bit 128), and so on.

The bit-address pattern cannot be stored directly as it may result in Mallory obtaining the pattern. Instead of storing the bit-address pattern, a *bit-address fetching block* is created which generates the address of each bit at run time. One logical block of code is generated for every bit-address to be calculated during installation. Hence if there are 128 bits then we have 128 *bit-address fetching block*s. The bit-address is generated prior to generating the fetching block



Key Bit 0 is stored at location 11
Key Bit 1 is stored at location 1
Key Bit 24 is stored at location n
.....
Key Bit 128 is stored at location 5

Fig. 3. Scattered Array

for the address. This means that the fetching block must evaluate to a specific value. This is achieved by generating an initial value and then performing a series of random arithmetic operations on it till the desired bit-address value is generated. This *bit-address fetching block* is made using different machine level arithmetic operations. If the *bit-address fetching block* is located at the same location in every instance of installation of $\mathfrak{p}_{crypto}$, an attacker can figure out the address from one instance of the installation and break the secret. To avoid this, the *bit-address fetching block* is moved around during installation. In addition, the 'junk' code present inside it serves to modify its size and location.

Implementation: The execution of bit-address fetching block results in the formation of the bit-address in a predetermined register/memory address which is then used for accessing the bit from the scattered array. Construction of the block is done by first setting a *min* and a *max* bound for the number of operations to generate an address. $\aleph$ operations generate a bit address such that $min \leq \aleph \leq max$, where $\aleph$ is not a fixed number. Each operation is one of seven basic mathematical operations (ADD, SUBTRACT, MULTIPLY, DIVIDE, AND, OR, NOT). During bit address fetching block generation the values of registers and memory locations used is initialized and one operation is chosen to be performed. The *result* of this operation is tallied against the required value of the bit-address, if the *result* differs from the required bit-address value, further operations are applied. If the *result* of these operations reaches a range of ± 5% of the desired value, and $min \leq \aleph \leq max$, the code generation is stopped and a last operation is added to make *result* becomes equal to the desired value. The last operation is either an addition or a subtraction operation on the *result*. If $\aleph < min$ mutually cancelling operations are added till the bit-address value is generated.

If $max < \aleph$, the block generation is restarted. This scheme can generate bit-addresses as well as inode number or the sector number where the scattered array is stored on the secondary storage media which in this case is the hard drive.

The location of the *bit-address fetching block* can be revealed by the presence of a jump/call instruction to this block. The target address for the call to the bit-address fetching block is generated during execution by a small block that is similar to the *bit-address fetching block* to prevent this scenario. The location of the block which calculates the target address is also randomized in the binary and padded with many junk calculations to obfuscate the calculations. To prevent any malicious code from executing the fetching blocks, self attestation is performed on the running image of the executable. The code computes hashes on sections of its process image and compares the results with the expected results hardcoded inside it. The attestation covers the fetching blocks and the application within which it is executed. A simple inline hash function is used to avoid the hash call from being observed on process tracers and tools like *objdump*. It can be considered difficult for an attacker to change the hash values stored in the binary as their locations are randomized in each installation of the system. Multiple numbers of such attestation codes are inserted in $\mathfrak{p}_{crypto}$, each attesting a different section of $\mathfrak{p}_{crypto}$.

TABLE III. DATA FROM DIFFERENT INSTALLATION INSTANCES

Position of fetch block	Location of jump to fetch block	Location of attestation block	Hash value generated
0x21B7	0xCF7	0x255D	0x145F0A
0x21B7	0xCF7	0x2317	0x67977
0x21C0	0xCBB	0x23C7	0x3ACCA
0x21C0	0xCBB	0x2542	0xF08AE
0x21CE	0xCE5	0x244E	0x4AB06
0x21CE	0xCE5	0x272E	0x13D852
0x21DA	0xCC5	0x23A3	0x30697
0x21DA	0xCC5	0x25C6	0xE5BF3
0x219D	0xCEB	0x23AD	0xA2D
0x219D	0xCEB	0x2232	0x63B7
0x21B2	0xCFD	0x2497	0xEAE06
0x21B2	0xCFD	0x21EC	0x66C3E
0x21B1	0xCF2	0x2248	0x489E0
0x21B1	0xCF2	0x2594	0x115E85

This framework was implemented on Ubuntu 8.04 Linux OS using Intel® Pentium® 4 processors executing with 1 GB of RAM. A 32 bit key was utilized for testing the framework. The key hiding application was provided 2 self-attestation blocks, and the size of each bit-address fetch block was set as 80 bytes. A scatter array size of 2Kb was used to store the keys, and the entire application generated was 14KB in length. This implementation uses a few simple obfuscation techniques to keep the implementation simple. Table 3 provides a short summary of results obtained for the scheme. It shows that even with the margin of as small as 0x50 bytes, a good measure of randomness was obtained in the system. This shows that any malware which tries to attack the application cannot do a remote analysis and use the information gained there to attach another instance of the installation of the same application.

VI. Conclusion

Cryptographic keys on systems are a target of attacks for malware writers. Malware may use a variety of techniques including digital forensics to extract keys from an end user machine. The lack of randomness in cryptographic systems in terms of memory and lack of memory isolation among various processes in an OS provides opportunities for malware to steal keys. This paper presents methods for storing keys inside a Hypervisor and in secondary storage medium. A Hypervisor isolates the memory of a guest OS from other guest systems and the Host OS/Hypervisor. The cryptographic keys are always stored in the Hypervisor, this results in a system where a compromised guest OS can perform cryptographic operations without exposing the key. The second technique effectively randomizes the location of a key inside every machine making it difficult for malware to obtain the key from secondary storage media without extensive case by case analysis. This work can be extended such that the hypervisor uses the disk hiding scheme to store its keys.

VII. Acknowledgements

This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-1011931. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

[1] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, 2002, pp. 64.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem and D. Engler, "An empirical study of operating systems errors," in Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, 2001, pp. 73-88.

[3] Web link. TrueCrypt - free open-source on-the-fly encryption. Retrieved October 4 2010. www.truecrypt.com

[4] Web link. BitLocker. Retrieved October 4 2010. http://windows.microsoft.com/en-US/windows7/products/features/bitlocker

[5] Web link. CAC: Common access card. Retrieved October 4 2010. www.cac.mil

[6] Web link, "Hi-Def DVD Security is bypassed," vol. 2009. Retrieved January 26 2007. http://news.bbc.co.uk/2/hi/6301301.stm?lsm

[7] Web link, "Hackers discover HD DVD and Blu-ray processing key - all HD titles now exposed," Retrieved February 13 2007. http://www.engadget.com/2007/02/13/hackers-discover-hd-dvd-and-blu-ray-processing-key-all-hd-t/

[8] J. Chow, B. Pfaff, T. Garfinkel and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," in 14th Conference on USENIX Security Symposium, 2005, pp. 331-346.

[9] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher and M. Rosenblum, "Understanding data lifetime via whole system simulation," in 13th USENIX Security Symposium, 2004. 2004, pp. 321-336.

[10] T. Pettersson, "Cryptographic key recovery from linux memory dumps," in Chaos Communication Camp 2007.

[11] A. Shamir and N. v. Someren, "Playing 'hide and seek'with stored keys," in Third International Conference on Financial Cryptography, 1999, pp. 118-124.

[12] A. Walters and N. Petroni, "Volatools: Integrating volatile memory forensics into the digital investigation process," in Black Hat DC, 2007.

[13] T. Vidas, "The acquisition and analysis of random access memory," Journal of Digital Forensic Practice, vol. 1, pp. 315-323, 2006.

[14] Anonymous "Thales ISS keyAuthority," vol. 2010.

[15] Anonymous "IBM security: IBM distributed key management system (DKMS)," IBM.

[16] K. Harrison and S. Xu, "Protecting cryptographic keys from memory disclosure attacks," in 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007, pp. 137-143.

[17] P. MacKenzie and M. K. Reiter, "Networked cryptographic devices resilient to capture," in IEEE Symposium on Security and Privacy, 2001, pp. 12-25.

[18] R. Canetti, Y. Dodis, S. Halevi, E. Kushilevitz and A. Sahai. Exposure-resilient functions and all-or-nothing transforms. Presented at 19th International Conference on Theory and Application of Cryptographic Techniques.