Key Protection for Private Computing on Public Platforms

Thomas Morris morris@ece.msstate.edu
Electrical and Computer Engineering
Mississippi State University
Mississippi State, 39762, USA

V.S.S. Nair nair@engr.smu.edu
Computer Science and Engineering
Southern Methodist University
Dallas, 75205, USA

Abstract

Private Computing on Public Platforms (PCPP) is a new technology designed to enable secure and private execution of applications on remote, potentially hostile, public platforms. PCPP uses a host assessment to validate a host's hardware and software configuration and then uses applied encryption techniques embedded in the operating system to isolate the protected application allowing its executable code, context, and data to remain unaltered, unmonitored, and unrecorded before, during, and after execution. PCPP must secure its encryption keys to ensure that the application isolation is robust and reliable. To this end we offer a protection scheme for PCPP encryption keys. We describe our PCPP key protection methodology and how it interacts with the other PCPP building blocks to isolate encryption keys even from privileged users.

Keywords*:* application isolation, encryption key protection, private computing

1 INTRODUCTION

Distributed computing technologies which enable the use of idle processors by remote users are abundant. SETI@Home [6] is a popular distributed computing application run by the University of California at Berkeley which uses remote computers to download and process data in the search for extra terrestrial intelligence. The Globus Toolkit [7] is an open source toolkit which has been used to build many working grids which are collections of computers linked to allow the sharing of computing resources across locations. Although there have been many successes in this area there has been a lack of large-scale commercial acceptance. We believe that this trend stems from a need for better application security and privacy while applications are stored and executed on remote platforms. Private Computing on Public Platforms (PCPP) was developed as a solution to this problem. PCPP enables the secure and private use of remote platforms by first performing a host assessment to validate the hardware and software configuration of the system and then using 4 active security building blocks which assure that applications executed on the remote platforms remain unaltered, unmonitored, and unrecorded before, during, and after execution.

PCPP [4] protects applications by isolating all memory and files used by the application from all other users regardless of privilege. In order to protect PCPP applications on a remote host, PCPP must have a secure and reliable means of storing encryption keys on the remote host. We have developed a new key protection methodology which isolates PCPP encryption keys from all other processes running on the same platform.. We call this new methodology PCPP key protection. PCPP key protection uses a key cache to hold all of the keys used by the other PCPP building blocks. This key cache is tagged with

integrity information during the operating system's context switch routine, at the point immediately before a PCPP isolated process relinquishes the CPU to another process, and then encrypted with a master key, k_m. The master key is then securely stored until the PCPP isolated process regains the CPU, at which point, again during the operating systems context switch routine k_m is retrieved to decrypt the isolated process's key cache.

In this remainder of this paper we first offer a discussion of related works. We then provide an overview of PCPP. Finally, in the body of this paper we provide a detail description of our new PCPP key protection methodology including a description of its architecture and implementation, a description of how the key protection methodology defends against attacks, and detail analysis of the run time impact associated with using the PCPP key protection methodology.

2 RELATED WORKS

Chow et al. [14] define a white box attack context in which among other things an attacker has complete control of the execution environment. Chow then offers a methodology for creating decryption implementations in which a static decryption key is embedded in the decryption implementation such that other users or malicious programs on the same machine cannot learn the encryption key. Embedding the decryption key in the decryption implementation may stop malicious users from learning the key; however, in the white box attack context, as defined by Chow, the malicious user would have no need of the decryption key since he could just use the decryption implementation as constructed to decrypt content at will. This of course, would not provide adequate protection for PCPP isolated processes.

Perkins et al. [7] hide secret keys in the heuristic solutions of NP-hard problems. Since the heuristic solution to the NP-hard problem can be solved much faster than a brute force implementation of the same problem, the key protection problem is reduced to limiting access to the chosen heuristic and knowledge of the chosen heuristic. However, in an open system we cannot reliably limit access to the chosen heuristic and as such this solution does not fit the PCPP model. Additionally, most NP-hard problems have multiple heuristics which may provide similar solutions possibly making it easy for an attacker to learn the key without knowledge of the exact heuristic.

Trusted Computing [1][2] uses a hardware device called a Trusted Platform Module (TPM) for encryption key generation, encryption key storage, and encryption. These functions of the TPM are required to be implemented such that they are tamperproof, resistant to observation, and resistant to reverse engineering. Such a hardware based solution is ideal, however, most platforms do not have TPM's on board and we desire a software only solution.

The Linux Key Retention Service (LKRS) [4] allows applications to store and retrieve key material from key rings, which are structures which point to linked lists of keys. LKRS attaches a thread key ring pointer to each thread's task structure (a structure used by Linux to hold a thread's context). Any privileged process running may dereference another process's thread key ring and traverse a set of pointers to learn the contents of the stored key. PCPP requires isolation of keys even from privileged users. As such, LKRS is inadequate for use with PCPP as a key protection system.

3 PRIVATE COMPUTING ON PUBLIC PLATFORMS

Private Computing on Public Platforms (PCPP) is an application security technology designed to protect applications operating in public computing environments. PCPP uses a host assessment made from internal and external scans of the public platform combined with 4 active security blocks which run alongside the protected application on the public platform; the executable guard, Secure Context Switch, Secure I/O, and PCPP encryption key protection, to protect the PCPP application while it executes on the public platform. The host assessment validates the public platform by first scanning it internally and externally to collect a set of platform attributes and second classify the host as a threat or non-threat using a Bayesian classifier. The executable guard is an encrypted ELF variant which securely stores the application's binary executable on the public platform and securely loads the binary executable into memory just prior to execution. Secure context switch stops eaves droppers from accessing the PCPP protected application's volatile memory by encrypting all protected application memory context when the

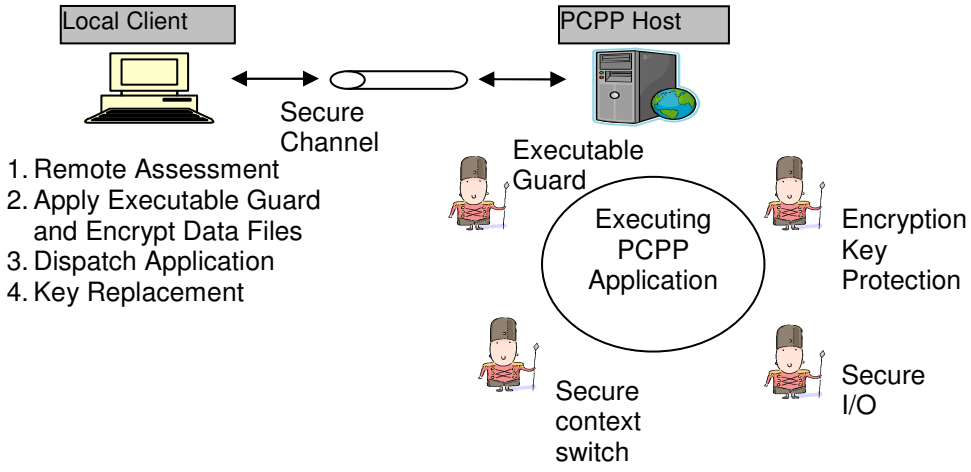


Figure 1: PCPP System Overview

application relinquishes the CPU during context switch and decrypting memory context on demand when the protected application is active. Secure I/O protects all PCPP files via decryption and encryption of all file input and output data, respectively, in the virtual file system layer of the operating system. The executable guard, secure context switch, and secure I/O all rely upon robust encryption key protection on the public platform.

4 PCPP KEY PROTECTION

PCPP key protection patches the Linux kernel scheduler's context switch routine to securely store a PCPP process's encryption keys immediately before a PCPP process relinquishes the CPU and to retrieve the stored encryption keys immediately before a PCPP process regains ownership of the CPU.

Each building block, Executable Guard (PPELF), Secure Context Switch, and Secure I/O, will have at least one encryption key and likely more than one. We use a key cache to store all of the encryption keys. The key cache uses an index to look-up and store keys as shown in Figure 2. We use a master key, k_m, to encrypt the key cache.

Using the k_m to encrypt the key cache reduces PCPP's protection burden to a single point. Building a fortress around k_m in turn protects all of the collected keys and all of the PCPP executable code and data. This fortress is a modified context switch routine which safely stores k_m during the period the PCPP protected process does not own the CPU.

The master key k_m is used every context switch by the Secure Context Switch and Secure I/O blocks. Context switches always come in pairs. If we assume the PCPP protected application is currently running and about to switch out, i.e. relinquishes control of the CPU, then during the first context switch Secure Context Switch and Secure I/O encrypt their respective data with keys stored in the key cache and then k_m is used to encrypt the key cache. The second half of the context switch pair is when the PCPP protected application switches in, i.e. regains control of the CPU, Secure Context Switch and Secure I/O use keys from the key cache to decrypt their respective data. k_m is used for exactly on context switch pair and then replaced. Each time the PCPP protected application switches out a new k_m is chosen.

Figure 3 shows a flow chart of the modified Linux context switch routine. The white boxes represent the functionality of the context switch routine before modification and the grayed boxes represent the additional functionality added to protect PCPP encryption keys. The unaltered context switch routine performs two simple steps. First, a pointer to the current processes memory map is changed to point to the memory map of the incoming process. Second, a small block of assembly code is called to swap out the CPU registers from the old process replacing them with the register values belonging to the incoming process. The PCPP updates to the context switch routine introduce two new paths through the context switch routine, one to handle outgoing PCPP processes and one to handle incoming PCPP processes.

The context switch routine is passed two pointers, *prev* (previous task) and *next (next task)*, when called, each is a pointer to a Linux task structure. The *prev* task structure holds the context belonging to the process relinquishing the CPU and the *next* task structure holds the context of the process which is gaining control of the CPU. We modified the PCPP task structure to contain a variable called *pcpp*. The *pcpp* variable is a Boolean which defaults to FALSE. When a PCPP process is launched the executable guard sets the *pcpp* variable to TRUE.

When *prev* is a PCPP process the modified context switch routine first encrypts any memory pages belonging to *prev* which are not currently in an encrypted state (this step is actually part of the Secure Context Switch building blocks and which is described in greater detail in **Error! Reference source not found.**). The encryption keys used to encrypt the context are held in a PCPP key cache which is attached to *prev's* task structure. When a PCPP process owns the CPU the key cache is in a decrypted state. It is encrypted as the PCPP is context switched out. Before encrypting the key cache we first add an integrity hash value to the key cache. The key cache is then encrypted using a master key, k_m. Finally, k_m is stored to await retrieval when the PCPP process regains control of the CPU.

When *next* is a PCPP process, the right hand path of the context switch routine is taken. First, k_m is retrieved from storage. Next, the key cache is decrypted. After the key cache is decrypted the integrity hash value, stored in the key cache when this PCPP process last relinquished the CPU, is validated. If the integrity hash value is incorrect a PCPP process shutdown commences. This shut down erases all PCPP files, overwrites all PCPP memory pages, and then kills the PCPP process. If the integrity hash value is correct the context switch will continue.

It is possible for both the previous thread and the next thread to be PCPP threads. In this case both branches of PCPP code in the context switch routine will be run, first the *prev.pcpp* branch, then the *next.pcpp* branch. In this case, the two PCPP tasks would each have separate key caches and master keys. As such encrypting *prev's* key cache and master key would not affect the retrieval of *next's* master key or decryption of *next's* key cache.

Table 1: PCPP Key Cache Integrity Hash Contents

Index	Item	Description
x_0	pcppcs_start	Physical address of the first instruction in PCPP context switch code
x_1	pcppcs_end	Physical address of the last instruction in PCPP context switch code

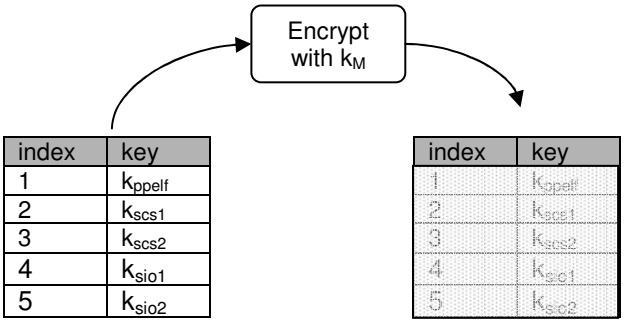


Figure 2: Encryption Key Cache

x_2	pcppcshash	Hash of the PCPP context switch code
x_3	keycache_loc	physical address of the PCPP key cache
x_4	tgid	process ID of the protected application
x_5	instrptr	physical address of random location in PCPP process's intructions
x_6	instrhash	hash of random 32 bytes from instruction memory

The integrity hash is a single value generated from the items listed in Table 1. Starting with the first value, x_0, we xor each value with the previous hash and then hash that intermediate result. The resulting hash will change if any of the items from Table 1 changes. Equation 1 illustrates the process for creating the integrity hash mathematically.

$$i_{n+1} = h(x_n \oplus i_n) \quad (1)$$

First, we store the starting and ending physical addresses of the PCPP context switch code (the grey boxes from Figure 3). These are used to confirm, after decrypting the key cache, when the PCPP process regains control of the CPU, that the PCPP context switch code has not moved. Since, this code is in the kernel address space it cannot be paged out and moved, it should stay in the same location permanently. The next integrity item is a hash of the PCPP context switch code. This item is used to confirm that the PCPP context switch code has not changed since the PCPP process last relinquished the CPU. The next item is physical address of the key cache itself. The key cache physical address is used to confirm that the key cache has not moved since the PCPP process last relinquished the CPU. The next item is the process ID of the protected process. The process ID is used to confirm that the process decrypting the key cache matches the process which previously encrypted it. The next item is the address of a random location in the PCPP process's instruction code. This address is the start location for hash of 32 random bytes from the PCPP process's instruction code. This hash is used as a spot check to confirm the protected process's instructions were not altered. If the integrity hash value found in the key cache after decryption does not the match the hash computed at the time of decryption the integrity

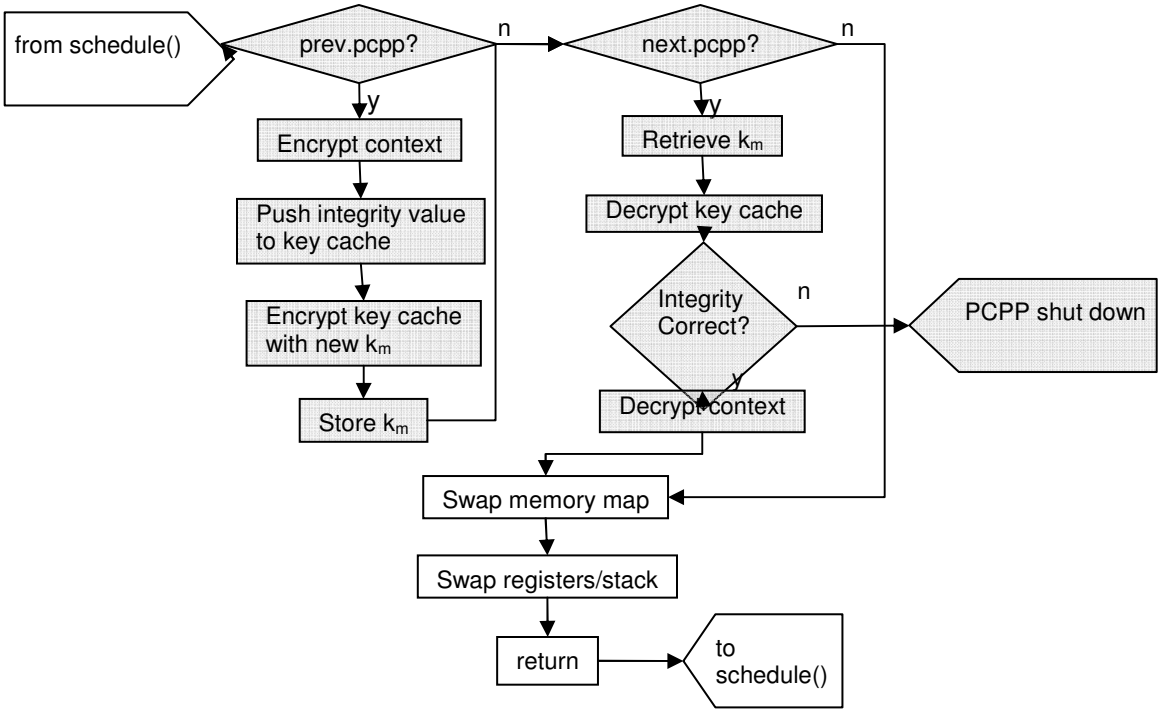


Figure 3: PCPP Context Switch Flowchart

check fails and the PCPP process is safely killed.

We developed two similar methods for storing k_m. Neither stores k_m directly, both store k_m indirectly via knowledge of the 1st and last members of a hash chain. k_m is the $(n\text{-}1)^{th}$ member of a n-length hash chain such as that shown in equation 1, where n is the number of successive hashes computed to find $h_n(h_0)$. We store h_0 and h_n as indicators for how to retrieve k_m. Retrieving k_m requires computing the hash chain from the starting value h_0 until h_n is reached and then setting $k_m = h_{n-1}$.

$$h_n(h_0) = h(h(...h(h_0))) \quad (2)$$

4.1 HMAC BASED HASH CHAINS

Protecting the master key is a difficult problem. In order to retrieve the key after storage we must also store some information on how to retrieve it. However, when the PCPP process relinquishes the CPU a foe may access this same information and couple that with knowledge of our hash chain implementation to recreate the hash chain. It is not possible to store a secret we can use which absolutely cannot be used by a foe. The SHA1/MD5 method stores h_0, h_n, and the hash type, which are then used to retrieve the key. We have shown in section via our LKRS exploit that privileged users can easily access the memory contents of other processes. This makes storing h_0, h_n, and the hash type in the clear dangerous. However, to hide these or encrypt them we would need a new secret and that secret would need to be stored. There is always a last secret which must be stored where a foe may conceivably find it. We developed to our HMAC [13] based hash chains to make it more difficult, though, we cannot make it impossible.

To make retrieving the master key more difficult for a foe we desire several properties. First, we prefer to choose from many hash algorithms. Second, we prefer to construct the hash implementations in such a way that any foe finds it much easier to try to use our hash implementations rather than build his own implementations and simply use his to create a copy of our hash chains.

Since, HMAC is a keyed hash we can use different keys to generate separate implementations. In fact, we build many separate HMAC functions with predefined keys embedded in the HMAC implementation executable code. Since, the HMAC algorithm implementations are distinct and built with different keys; they meet the requirement of many different hash choices.

The keys used for each HMAC implementation are defined as constants in their respective implementations and copied into memory as the first step when the HMAC implementation is called. Because the keys are constants and too large to copy with a single *mov* assembly instruction they are stored in parts in the executable code and copied into registers when the HMAC implementation is called. After the HMAC is computed we overwrite these registers with zero. Storing the HMAC keys in parts as constants in the executable code, rather than in the processes task structure or some other more accessible location serves to make it more difficult for a foe to steal the HMAC keys, though definitely not impossible.

Ideally, the HMAC implementations could be further obfuscated to make even more difficult for a foe to find the keys in the HMAC executable code. Currently, the keys are stored as 4, 32-bit constants and loaded into registers with 4 *mov* calls. Finding, them would be somewhat difficult, but certainly not impossible. If the code were obfuscated to diffuse the keys further into the HMAC implementation it would become more difficult for a foe to find the HMAC keys. This would serve to encourage a foe to attempt to run our HMAC implementation rather than try to steal the HMAC keys and run offline.

It is important that the HMAC implementations only be used for one PCPP application run. Re-use of the same HMAC implementations with the same embedded keys would give foes extra time to learn the HMAC keys. The HMAC implementations can be derived on the PCPP local client and sent to the PCPP remote host during application launch or they can be created on the remote host loaded into memory when the PCPP application launches.

Equation 3 describes the ordinary HMAC. Ordinary HMAC takes a single key and xor's that key with separate ipad and opad values.

$$hmac(x) = h(k \oplus opad, h(m, k \oplus ipad)) \quad 3$$

We found it simpler to dimply use two separate keys rather than create one key and then XOR twice for use in the HMAC. Our HMAC is shown in equation 4.

$$hmac(x) = h(k_1, h(m, k_2)) \quad 4$$

We first choose h_0 using a random number generator. We also use the random number generator to choose the length of the hash chain and to choose the underlying hash algorithm used to compute the HMAC this chain is derived from. The length of the hash chain must be greater than 16 and less than 128 links. Our current implementation chooses between MD5 and SHA1, though other hash algorithms would be acceptable as long as they meet HMAC requirements.

We store h_0, h_n, and the hash type in the PCPP processes task structure. The hash type now indicates both the underlying hash algorithm and which HMAC implementation to use. The h_0, h_n, and the hash type values are still stored in the clear when the PCPP process relinquishes control of the CPU. As mentioned above, k_1 and k_2 are embedded in the HMAC executable code. When the PCPP task regains control of the CPU h_0, h_n, and the hash type are used to call the appropriate HMAC implementation to recreate the hash chain and derive k_m.

4.2 TRANSFERRING KEYS FROM THE LOCAL CLIENT TO THE REMOTE HOST

In addition to safe storage of encryption keys on the remote host, the initial master key and key cache must be securely transmitted to the remote host from the local client.

The PCPP host must run a server which receives the PCPP application executable, any files sent with the executable, a hash chain, and an encrypted key cache from the local client. Once the server receives the complete launch packet it launches the application on the remote host. This PCPP server must use SSL to encrypt the connection between the local client and the remote host. It must also be protected by all the PCPP building blocks like any other PCPP application to ensure the PCPP servers executable and data remain unaltered, unmonitored, and unrecorded.

Table 2: Initial Key Transfer

Index	Who	Step
1	Remote Host	Send HMAC keys over SSL secure channel
2	Local Client	Create key cache: stores initial encryption keys for PPELF and any files sent with the executable
3	Local Client	Create initial master key (k_m), and hash chain
4	Local Client	Encrypt key cache
5	Local Client	Send key cache and hash chain to remote host
6	Remote Host	Store key cache and hash chain to server PCPP structure
7	Remote Host	Launch application
8	Remote Host	New process inherits key cache and hash chain from parent

Table 18 shows the steps required to securely send the initial master key and hash chain from the local client to the remote host. The executable and any files to be sent with the executable must be encrypted before being sent to the remote host. This process may take place immediately before sending the executable and files to the remote host or it may happen sometime in advance. Either way the encryption keys used for this process must be sent to the remote host with the executable and any other files. These encryption keys are placed in a key cache as described above and the key cache is encrypted with

the initial master key k_m. Immediately, before this encryption step a hash chain is built which is used both to derive and store k_m. The encrypted key cache and the hash chain are then sent to the remote host. All communication between the local client and remote host is encrypted to prevent third party eaves dropping. Once on the remote host the PCPP server stores in the hash chain and key cache in its own PCPP structure. The PCPP server then uses exec to launch the PCPP application. When the new PCPP process is launched it inherits a copy of its parent's task structure. If the parent is a PCPP process, which in this case it is, the parent's task structure will contain a PCPP structure. The parent PCPP structure contains a pointer to second PCPP structure, which is intended for copying to a child during launch. When a PCPP process's task structure is copied the copying code first searches for a child PCPP structure. If a child PCPP structure is available this structure is copied and used as the new process's PCPP structure. If the child PCPP structure does not exist, the parent's PCPP structure is used. For the case when the PCPP server is launching PCPP applications there will always be a child PCPP structure. When PCPP applications launch children they likely will not have PCPP structures and will therefore inherit the PCPP structure, complete with key cache and key chains from the parent.

The first item in Table 2 is specific to the HMAC case. Here, the PCPP server must send a pair of HMAC keys to the local client which can be used to create the hash chain.

4.3 DEFENSE AGAINST ATTACKS

We foresee three types of attacks against the PCPP key protection system. First, an attacker may attempt to jump midway into the key protection code in an attempt to bypass certain checks. Second, an attacker may attempt to learn the master key, k_m, by implementing his own key retrieval code or copying the PCPP key retrieval code to a separate location and modifying it. Finally, an attacker may attempt to copy a PCPP process's Linux task structure, the PCPP process's PCPP structure, and all or part of its encrypted memory contents to build separate task structure which is then placed on the ready to run queue for decryption by the context switch routine. In the remainder of this section we describe how the PCPP key protection system defends against these attacks.

In the first attack case a foe may attempt to jump to an intermediate point in the PCPP context switch code expecting to find a function return which will pop the foe's return address from the stack and return control to the attacking program. For example an attacker may wish to jump to the master key retrieval code expecting the master key to retrieve and then expecting a conveniently placed return to send control back to the attacking program. We stop such an attack by compiling all of the PCPP context switch code, shown in Figure 3, as inline code. This means all function calls are replaced by the compiler with a unique copy of the function placed in line with the calling code where the function call was previously. This stops the PCPP context switch code from calling functions using the *call* assembly instruction and then using a *return* mnemonic to jump back to the calling location. By doing this we avoid attackers jumping directly to the first instruction of a PCPP function and then using our own return call to jump out of the routine. By in-lining all PCPP context switch code the first encountered return instruction occurs at the end of the entire context switch routine. As such anytime an attacker jumps to any instruction in the PCPP context switch routine it must run through to the end of the PCPP context switch routine. Jumping into the context switch code without properly setting pointers to a previous task and a next task will cause the operating system to lock up, ceasing all activity on the machine until a reboot is performed. If an attacker manages to properly create pointers to a previous and next task the result would still almost definitely be a locked up system. In-lining stops foe's from jumping into the context switch routine expecting to return out before a context switch.

The second attack scenario involves an attacker either copying the PCPP master key retrieval code to a separate location or using his own hash implementation, with a pilfered hash chain initialization value, end value, and hash type, to retrieve the master key. Copying the key retrieval code and executing it in another location is difficult but possible. Since the code is in-lined there will be no symbols in the instruction code pointing to the starting address of the key retrieval code. Also, there will be no obvious way to know where the key retrieval code ends. If a foe finds the start and end of the key retrieval code he will then need to copy it and then add code to protect the integrity of his own codes registers by identifying all registers used in the copied code and pushing these to the stack at the beginning of the

copied code and popping them from the stack when returning to his own code. If all of this is done the copied code could be used.

Instead of copying the key retrieval code to separate location an attacker may choose to only copy the hash chain initialization value, the hash chain end value, and the hash type value from the victim process's PCPP structure. We make this very difficult by using HMAC hash chains with HMAC keys embedded in the HMAC implementations as constants and by using an HMAC algorithm based upon two keys rather than the standard HMAC which is based upon just one. To successfully use his own HMAC code first the attacker would need to implement an HMAC which mimics the PCPP HMAC behavior. This would not be overly difficult since the code could be copied from the PCPP installation. Next, the foe would need to steal the HMAC keys. Since these are embedded in the HMAC executable code which is in-lined in the rest of the PCPP context switch code these key may be difficult to find. Furthermore, if the HMAC implementations were obfuscated to hide or diffuse the keys in the HMAC implementation finding these keys would be all the more difficult.

Both copying the key retrieval code and executing it elsewhere and copying just the inputs to the key retrieval code and deriving the master key from a separate hash implementation are possible attacks. However, both are considerably difficult. If an attacker does manage to retrieve k_m he may then proceed to decrypt the key cache. In this case he will learn the values of the protected keys in the key cache. To limit the damage in case this does happen we choose a new encryption key each time an isolated page is encrypted in the demand encryption/decryption algorithm. By doing this we limit the useful life of keys stolen by an attacker. We also change k_m each time a PCPP context switch relinquishes the CPU.

The last attack vector involves an attacker copying all or part of a PCPP process's context to a separate process's memory space and then attempting to use the PCPP context switch code to decrypt the copied PCPP memory but still switch CPU control to a program chosen by the attacker. If an attacker attempts to build a Linux task structure which points to a set of PCPP data pages but replaces the instruction memory pages with a separate program the context switch code will decrypt the PCPP data pages as the attacker desires. However, it will also attempt to decrypt the replaced instructions with a key from the key cache. Two things can happen to the instructions at this point. If the instructions were not encrypted, or encrypted with a different key than the one found in the key cache, the instructions would be mangled by the decryption step and consequently the integrity check would fail causing the PCPP shutdown process to run. If the attacker managed to encrypt his replacement program pages with the same key used for the actual PCPP application then the replacement program would decrypt correctly. However, the integrity check would still fail because the hash of the random 32 bytes of instruction code will not match the hash pushed into the key cache when the PCPP application most recently relinquished the CPU.

4.4 PCPP KEY PROTECTION PERFORMANCE OVERHEAD

There is a run time performance overhead associated with using the PCPP Key protection system. This overhead is limited to increasing the time required to complete a context switch. There is no performance degradation outside the context switch routine. We measure overhead for two configurations of the PCPP key protection system. First, we measured the overhead for a system which stores the master key with SHA1 and MD5 hash chains. Second, we measured the overhead for a system which stored the master key with HMAC based hash chains.

Figure 4 shows the context switch run time overhead for 4 PCPP key protection configurations. For all of the measurements we built a small routine which solely performed the key protection steps. The steps when context switching out include: create new master key, create integrity value, add integrity values to key cache, and encrypt the key cache. The steps when context switching in include: retrieve the master key, decrypt key cache, calculate integrity values, and validate the integrity values in the key cache. To accurately measure the time to perform these steps we built a loop which perform each step in sequence. We then timed this loop while it executed 100,000 times. The time for one loop iteration was the total measured time over 100,000. The numbers are the time for 1 context switch. Since the key retrieval operations and key storage operations are close to mirrors of one another we divide the time for one iteration by 2 to get the overhead for 1 context switch.

The measurements for Figure 4 were performed on a workstation running Linux kernel 2.6.2.20 [9] with an AMD CPU clocked at 3 GHz.

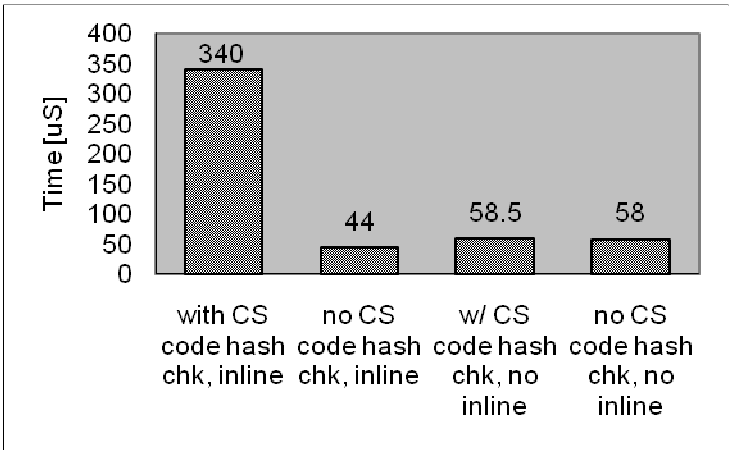


Figure 4: Context Switch Overhead from Various Key Protection Configurations

The first pair of results in left most position of Figure 4 show the run time overhead when all PCPP context switch code is compiled with the inline attribute set and with the use of a hash of all of the PCPP context switch code in the key cache to confirm the context switch code did not change since key cache encryption. The overhead for this configuration is large for both the MD5/SHA1 version and the HMAC version. This overhead is dominated by the time to compute the hash of the in-lined PCPP context switch code. Analysis of this code shows that the in-lined version of the measure code listed above was over 54K bytes. The second set of bars in Figure 4 show the impact of removing the hash of the PCPP context switch code. With this hash removed the overhead drops from 340uS to 44uS.

The next two data sets in Figure 4 show the impact of compiling the PCPP context switch code without the in-line attribute. For both cases the context switch overhead is similar to, but slightly higher than, the overhead for the case which used in-lined code but skipped the hash check of the context switch code. We expect these versions to be slightly slower than the in-lined version since the overhead of dealing with function calls is added into these results.

We conclude from this that in-lining the context switch code causes the code to grow significantly in size. This by itself would generally be acceptable since most hosts would have plenty of memory to accommodate a larger context switch routine. However, the larger in-lined code size does lead to slow hash times for validating the PCPP context switch code. We have explained the necessity of this validation step above and feel it is a required step to ensure the security of the PCPP key protection system.

We derived a set of equations to predict overall performance impact of using the PCPP key protection system.

$$t_{kprot} = e\mathrm{T} + \eta t_{out} + \eta t_{in} \qquad 5$$

Equation 5 provides an approximation of the run time required to run an application with Secure Context Switch enabled. Equation 5 starts by representing the amount of time a program needs to run in terms of context switch time slices, or epochs, e, times the period of 1 epoch, T. As such $e\mathrm{T}$ is the amount of run-time the program would require if it were running without any interruption, i.e. in an environment free of context switches. We represent this basic run-time as $e\mathrm{T}$ because e provides a minimum number of T length time slices required to run the program. All other terms in equation 5 are overhead associated with context switching.

The second term of equation 5, ηt_{out}, adds the context switch time for η context switch out(s), i.e. relinquishing the CPU. The third term of equation 5, ηt_{in}, adds the context switch time for η context switch in(s), i.e. regaining control of the CPU. In the second and third terms η is the adjusted number of times slices used by the application. We adjust η to account for the time needed for an ordinary context switch, extra time during context switch out to generate a new master key hash chain, calculate and store integrity information, and encrypt the key cache, and extra time during context switch in to retrieve the master key, calculate and check integrity information, and decrypt the key cache.

Table 3: Key Protection Overhead Variable Definitions

Variable	Description
t_{kprot}	Run time with PCPP key protection
t_{norm}	run-time without PCPP key protection
e	number of complete times slices needed to complete execution
T	period of one time slice
η	adjusted number of times slices after adding PCPP key protection overhead
t_{cs}	Context switch time without PCPP key protection
t_{in}	time for protected application to context switch in
t_{out}	time for protected application to context switch out
t_{key}	time to retrieve/store master key, encrypt/decrypt key cache, and perform integrity checks

$$t_{out} = t_{in} = t_{cs} + t_{key} \quad (6)$$

Equation 6 shows the relationship between t_{in} and t_{out}. For our PCPP key protection implementation t_{in} and t_{out} are set to equal one another because the code for context switching in mirrors the code for context switching out.

$$\eta = \left(\frac{2et_{cs} + 2et_{key}}{\mathrm{T}} \right) + e \quad (7)$$

Equation 7 estimates the adjusted number of time slices a PCPP key protected process will require. Equation 7 adds the minimum number of uninterrupted time slices, *e*, to the number of extra time slices resulting from the extra time required for basic context switching and extra time required for encrypting and decrypting protected pages during the context switch. The first term of equation 7 calculates the extra times slices required for context switching by first summing the time for 2 ordinary context switches and 2 master key retrievals/stores and then dividing by the period of one time slice, T. We use 2 context switch times and 2 master key retrieval/store times because context switches come in pairs, one at the beginning of the time slice and one at the end of the time slice.

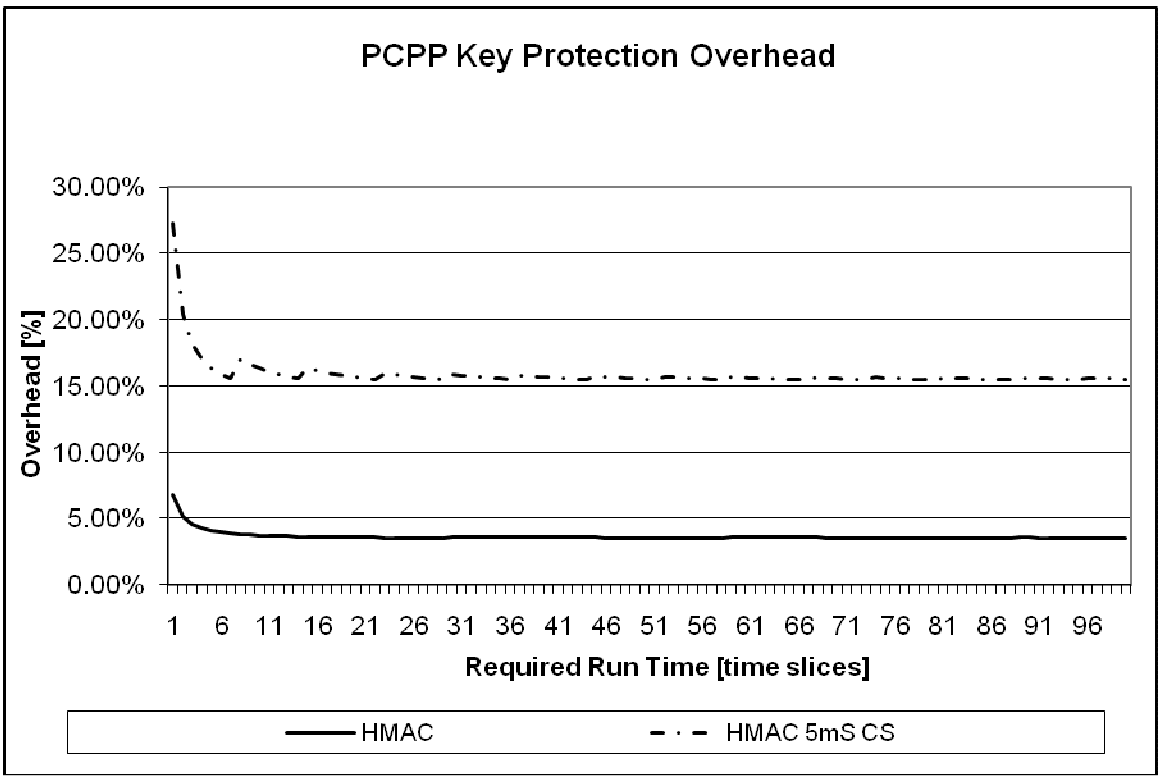
$$t_{norm} = e\mathrm{T} + \eta t_{out} + \eta t_{in} \quad (8)$$

Equation 8, which computes t_{norm}, looks just like equation 5. The only difference is the definitions of η, for the number of adjusted context switches, and the definitions for t_{in} and t_{out}

$$t_{in} = t_{out} = t_{cs} \quad (9)$$

Equation 9 defines t_{in} and t_{out}. Since there is no master key retrieval/storage, no integrity checks, and no key cache encryption/decryption required, the t_{key} term is removed relative to equation 6.

Figure 5: Predicted PCPP Key Protection Overhead



$$\eta = \left(\frac{2t_{cs}}{\mathrm{T}} \right) + e \qquad 10$$

Equation 10 shows the definition of η for the non-PCPP key protection case. Since, without PCPP key protection there is no master key retrieval/storage, no integrity checks, and no key cache encryption/decryption this term is removed relative to equation 7.

$$kprot\_overhead = \frac{t_{kprot}}{t_{norm}} \qquad 11$$

Equation 11 defines the overhead for PCPP key protected applications when compared to the same application running without PCPP key protection.

PCPP key protected applications when compared to the same application running without PCPP key protection.
We used equations 5-11 to plot the predicted overhead associated with using PCPP key protection. Figure 5 shows the predicted run time overhead for the HMAC implementations with in-lined code and all integrity checks in place. We plotted two curves. One set based upon the assumption of a 20mS context switch period, meaning our application runs in 20mS uninterrupted segments. The second set of curves show the predicted overhead for a 5mS context switch period. The chart shows that as our application is allowed to run in longer uninterrupted segments the overhead from context switching decreases. For the 5mS case we see overhead ranging from 10-15%. For the 20mS case we see overhead ranging from 3-5%. While this overhead is significant, we believe it is acceptable for the improved key protection gained by using the PCPP key protection system.

When running our key protection code with actual programs we see overheads which vary widely from job to job but tend to stay in the 2-15% range. This matches the ranges seen in Figure 5.

The overwhelming majority of the key protection overhead comes from hashing the inline code section which we use to ensure the integrity of the key protection code and our PCPP context switch code. Improving the run time performance of the key protection code can be done by reducing the size of the hashed inline code section. One might choose to hash less code, thereby choosing to guard the integrity of less code.

5 CONCLUSIONS

In this paper we presented a modified Linux context switch routine which encrypts a PCPP key cache and with a master key, k_m, when a PCPP process relinquishes control of the CPU. After encryption of the key cache k_m is securely stored. When the PCPP process regains control of the CPU, k_m is retrieved and then the PCPP key cache is decrypted. Before allowing a PCPP process to resume ownership of the CPU integrity information stored in the key cache when the PCPP process relinquished ownership of the CPU is validated. k_m is not stored as plaintext on the host platform, rather k_m is the n-1th member of a hash chain in which the root of the hash chain and the nth element of the hash chain are stored. The hash algorithm is chosen on the local client randomly and sent to the host platform during application launch.

While our PCPP key protection system is still vulnerable to other privileged processes stealing k_m when the PCPP process is not running this is made computationally difficult throught the use of HMAC based hash chains and the integrity information validation. Also, coupling the PCPP key protection mechanism with our Secure Context Switch technology limits the useful life k_m. We believe our PCPP key protection methodology is a considerable improvement over the Linux Key Retention Service. Overall, we believe that our PCPP key protection system is a significant improvement over other software based key protection systems.

REFERENCES

[1] Marchesini, J., Smith, S., Wild, O., MacDonald, R., Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear, Dartmouth Computer Science Technical Report TR2003-476, ftp://ftp.cs.dartmouth.edu/TR/TR2003-476.pdf

[2] Trusted Computing Group Fact Sheet, https://www.trustedcomputinggroup.org/about/FACTSHEET_revised_may_07.pdf

[3] Felten, E.W., Understanding Trusted Computing: Will its benefits outweigh its drawbacks?, IEEE Security and Privacy Magazine, Volume 1, Issue 3, May-June, 2003

[4] Morris, T. Nair, V.S.S. Private Computing on Public Platforms: Portable Application Security. Submitted to Wiley InterScience Journal of Wireless Communications and Mobile Computing. (to appear)

[5] Kumar A., Chopdekar S., Getting Started with the Linux key retention service, http://www.ibm.com/developerworks/linux/library/l-key-retention.html

[6] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. 2002. SETI@home: an experiment in public-resource computing. *Communications of the ACM* 45, 11 (Nov. 2002), 56-61.

[7] Foster, I. Globus Toolkit Version 4: Software for Service-Oriented Systems**.** IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2005.

[8] Perkins, G., Bhattacharya, P., An Encryption Scheme for Limited k-time Access to Digital Media, IEEE Transactions on Consumer Electronics, Volume: 49, Issue: 1, Feb. 2003

[9] The Linux Kernel Archives, http://www.kernel.org/

[10] Barak, B. and Halevi, S. 2005. A model and architecture for pseudo-random generation with applications to /dev/random. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA, November 07 - 11, 2005). CCS '05. ACM, New York, NY, 203-212.

[11] D. Eastlake and P. Jones. RFC 3174. US Secure Hash Algorithm 1 (SHA1). http://www.faqs.org/rfcs/rfc3174.html

[12] R. Rivest. RFC 1321. The MD5 Message-Digest Algorithm. http://www.faqs.org/rfcs/rfc1321.html

[13] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104. HMAC: Keyed-Hashing for Message Authentication. http://www.faqs.org/rfcs/rfc2104.html

[14] Chow, S. Eisen, P. Johnson, H. Van Oorschot, P. A White-Box DES Implementation for DRM Applications. Digital Rights Management. Springer-Verlag LNCS 2696, pp 1-15, 2002.