Secure Context Switch for Private Computing on Public Platforms

Thomas H. Morris, *Member, IEEE*, V.S.S. Nair, *Member, IEEE*

***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 four active security building blocks which together allow an application to remain unaltered, unmonitored, and unrecorded before, during, and after execution on the public platform. In this paper we develop a key PCPP building block, Secure Context Switch (SCS), which isolates an executing application's context, i.e. its executable code, data segments, heap, and stack, using encryption techniques. Additionally, we detail our implementation of SCS and offer experimental results showing the performance impact of protecting an application with SCS.**

***Index Terms*—application isolation, private computing**

I. Introduction

Distributed computing technologies which enable the use of idle processors by remote users are abundant. SETI@Home [8] 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 the lack of commercial acceptance of distributed computing technologies 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) [3] [4] [5] [6] was developed as a solution to this problem. PCPP enables the secure and private use of remote public 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 public platforms remain unaltered, unmonitored, and unrecorded before, during, and after execution.

Secure Context Switch (SCS) is one of the four PCPP active security building blocks. SCS protects memory pages assigned an application. Specifically, SCS protects an applications executable code, data segments, stack, and heap during execution by isolating these memory pages using applied encryption techniques. SCS encrypts the aforementioned memory pages immediately before a process relinquishes the CPU during context switch. Immediately before a PCPP protected process regains control of the CPU the encrypted memory pages are decrypted. Our current SCS implementation requires a single processor architecture. On a single processor platform attackers cannot read, write, or alter the PCPP application's memory when the PCPP application itself is running. When the PCPP application relinquishes the CPU an attacker may attempt to read memory locations owned by the protected application, however, with SCS, these pages will appear encrypted to the attacker.

The body of this paper defines PCPP, defines SCS, details our implementation of SCS, and compares experimental SCS run times to non-SCS run times. Finally, we offer a section on SCS and PCPP future work.

II. Related Work

We find no related work which directly overlaps with SCS. There are other methods for isolating applications. First, SELinux [10] is a role based access control system which limits access to applications and data to users which take specified roles. Because, the number of available roles is unlimited a role can be defined for any arbitrary application. SELinux then requires that only users with specific roles may access controlled objects. We find two problems with SELinux which make it inapplicable to PCPP. First, SELinux policy can be complex and difficult to validate. Second, since SELinux policy can be changed by the root administrator on a system it is feasible that a user may gain root privilege and disable SELinux or alter its protections.

A second related work is Trusted Computing [9]. Trusted Computing isolates certain protected API of a process by running these API in a curtained memory environment. Generally, the protected API are security sensitive shared libraries such as encryption API, privilege elevation API, etc. Since these API run in curtained memory other applications are physically blocked from accessing the memory used by the protected API. The Trusted Computing specifications do not specify how vendors should implement curtained memory. As such curtained memory effectiveness as a Trusted

Manuscript received August 15, 2008.

Thomas H. Morris is with Mississippi State University, Mississippi State, MS 39762 USA. (e-mail: morris@ece.msstate.edu).

V.S.S. Nair is with Southern Methodist University, Dallas, TX 77205 USA. (e-mail: nair@engr.smu.edu).

Computing building block is unknown. Since the OS will undoubtedly contain a mechanism to load code segments into curtained memory it stands to reason that if an adversary manages to load his or her own code into curtained memory he may be able to access other areas of the curtained memory region and therefore alter, or copy sensitive memory contents belonging to the protected API. Furthermore, Trusted Computing curtained memory is only intended to isolate a subset of security critical API and has not been proposed for use in isolating all memory contents belonging to an application.

III. Private Computing On Public Platforms

A. Overview

PCPP enables remote execution of protected applications on public platforms. We define the following requirements for PCPP. First, we must achieve private computing on the public platform which we define as the PCPP protected application remaining unaltered, unmonitored, and unrecorded before, during, and after execution on the public platform. The second requirement is that PCPP must have a software only implementation, which enables a much larger set of potential public platforms. Third, PCPP is an opt-in system. PCPP will not execute applications on an idle host without that host's prior permission. Furthermore, the opt-in process allows code to be downloaded and installed on a platform before it is used. Fourth, PCPP must be able to validate a public platform before use. This ensures that the platform is capable of executing the PCPP protected application and allows PCPP to assess the relative safety of using the public platform. Fifth, PCPP provides application level protection. As such, only chosen applications are subject to its protections, and consequently only those applications are subject to its overhead. Finally, as encryption is fundamental to PCPP, all encryption keys must be robustly protected while in use on the public platform.

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 scanning it internally and externally to collect a set of platform attributes. Initially, subsets of these attributes are collected and evaluated to make a quick go-no-go decision about the platform. One key go-no-go requirement is that the operating system and all PCPP server code has not changed since the original PCPP opt-in. Next, the entire set of scanned attributes is used to classify the public platform as a threat or non-threat. We showed in [4] that a Naïve Bayesian classifier accurately classifies potential platforms as threats or non-threats by detecting trends among the collected attributes with a small execution time overhead for each use of the public platform. Public platforms are only used if they are first classified as non-threats. Validation of a platform's hardware and software configuration reduces the likelihood of attacks but does not directly mitigate risk.

An executable guard protects the application's binary executable as it is stored on the public platform and during the process of the loading the binary executable in to memory just prior to execution. Privacy protected ELF (PPELF) [5] is an implementation of an executable guard for the Linux operating system. PPELF is an extension of the ELF [1] format in which the application code and data contained in the binary executable are encrypted. PPELF uses a new binary format handler and a modified GLIBC interpreter to decrypt and load the PCPP application executable into memory. The PCPP executable is first loaded from non-volatile memory into RAM while encrypted. Only after the application is loaded into RAM is it decrypted to allow execution. The application code is never in a decrypted state in non-volatile memory (hard disk). The PPELF binary format offers a simple effective way to isolate the application executable.

Secure context switch (SCS) stops eaves droppers from reading from PCPP protected application's volatile memory. SCS isolates the PCPP application's volatile memory from other threads running on the host by modifying the Linux kernel scheduler to encrypt the volatile memory pages which store PCPP application executable code, data segments, stack, and heap when the application is context switched out and decrypts the same executable code, data segments, stack, and heap when the application is context switched in. In its current form SCS is only effective on single processor hosts. We plan to expand this to cover symmetric multi-processor hosts as future work.

Secure I/O (SIO) [6] protects all PCPP files. All files used as input to the PCPP application are required to be encrypted before sending to the public platform. Files created during execution are automatically encrypted. As with PPELF the encrypted files are never stored as plaintext in non-volatile memory. SIO modifies the POSIX read and write system call implementations to encrypt all data on writes and decrypt all read data. Additionally, SIO makes changes to the mmap system call implementation to ensure files accessed with mmap are also protected. Since the SIO encryption takes place at the system call level it is file system independent. This means copying the file to a different file system will not decrypt it, as is the case with encrypted files systems.

The PPELF binary format, secure context switch, and encrypted I/O all rely upon robust encryption key security on the public platform. PCPP encryption key protection [6] isolates PCPP encryption keys from all other processes running on the same platform, regardless of the privilege assigned to that process.

We are introducing PPELF, SCS, SIO, and Encryption Key Protection as a set of four separate papers. This paper discusses the second of the four, the Secure Context Switch (SCS).

B. Accessing another Process's Memory Contents

PCPP needs to isolate a protected application's memory from other applications running on a platform. With most operating systems, non-privileged users may not access memory which is in use by other users. However, privileged users generally may access any memory in the system. PCPP needs to isolate protected memory contents from even privileged users.

We built a small kernel module, which we call the snooper module, for use on Linux platforms to demonstrate the ability of one user to access the memory of another. The snooped module scans the task structures (a structure used by the Linux to hold a thread's context) of all tasks currently active in the system looking for a certain signature. This signature can be any attribute the task structure contains. In our case we looked for a Boolean indicating the task was a PCPP task. Once a targeted task is found the snooper module simply follows a pointer in the task structure which points to a linked list of all contiguous memory areas assigned to the targeted task. After locating the desired memory locations, accessing them requires pointing setting the page fault hander to point to the victim process's memory map and then accessing it like any other memory pointer. Alternatively, one can convert snooped virtual addresses to physical addresses manually, bypassing the page fault handler. Bypassing the page fault handler also bypasses any access permissions set for the page to be accessed. We built this first portion of our memory snooper as a kernel module so that this code would run with kernel privilege. We then added a mechanism to allow calling this code from any user process.

We called the snooper module repeatedly in a loop so that we could run until we found a PCPP process in memory to attack. When the snooper successfully found a PCPP process it simply printed a portion of that process's memory to the screen as hexadecimal code.

To use our snooper we required root privilege only to install the snooper as a kernel module. With the snooper module installed any user can monitor any process's memory contents regardless of the attacking user's privilege level and regardless of the victim's privilege level.

IV. Secure Context Switch (SCS)

A. Overview

SCS protects volatile memory assigned to a process by interceding in the context switch procedure and encrypting application volatile memory contents during context switch out and decrypting the same memory contents during context switch in. Specifically, SCS decrypts the process's instruction memory contents, data segments, stack, and heap. When run on a single processor platform the SCS effectively denies other active processes the ability to access the protected processes volatile memory.

At the end a PCPP process's allotted time slice and when the PCPP process voluntarily relinquishes the CPU the modified Linux scheduler will the operating system calls the *context_switch()* routine to switch the current memory map and CPU register contents before relinquishing CPU control to the next process. SCS intercedes in the context switch process and encrypts the memory pages holding the PCPP process's instruction memory contents, data segments, stack, and heap, before jumping to the next process. When a waiting PCPP process regains control of the CPU, SCS, again within the context switch routine, decrypts the PCPP process's instruction memory contents, data segments, stack, and heap and then allows the context switch to proceed.

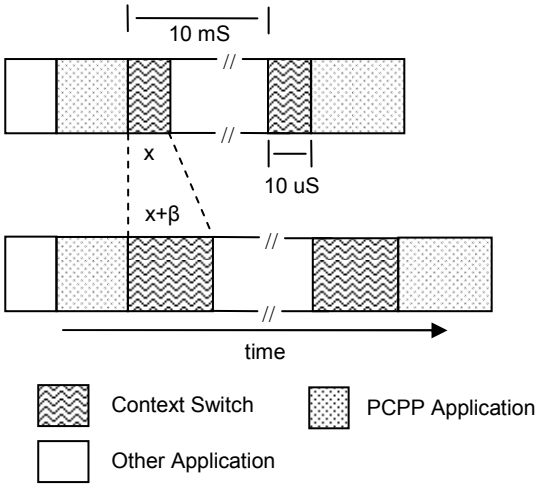

Figure 1: Secure Context Switch Overhead

B. Performance Analysis

Figure 1 shows how SCS adds to a processes overall run time. The top timing diagram in Figure 1 shows a process running on an unaltered kernel. For the unaltered timing diagram the context switch period is set at 10mS and the context switch overhead (x) is constant at 10uS. The figure does not show the context switch period to context switch overhead ratio to scale. The bottom timing diagram represents a kernel running with SCS. Since SCS must encrypt and decrypt memory pages belonging to the protected process at each context switch coincident to a PCPP process, the context switch overhead increases by β. The total run time delay for a process protected by SCS is equal to the number of times the protected process context switches in (takes ownership of the processor) and out (relinquishes the processor), which we call n, times β. In the figure the protected process context switches out 1 time, to allow an unrelated application to run, and context switches in 1 time, when regain the CPU, so n is 2 in this case. As such, the total run time increase for the PCPP application attributable to SCS is 2β. Increased context switch frequency requires more encryption and decryption of the protected processes memory contents and therefore leads to longer application run times.

β increases with the amount of memory which must be encrypted or decrypted during a context switch. As such, the performance overhead due to SCS protection increases as protected processes use more memory.

C. Experimental Evaluation

We collected empirical data to compare the run time of a

SCS protected application to that of a non-protected application. All data was gathered running a program called *matrix* which takes *m* and *n* size parameters from the command line and then dynamically allocates memory to store the *m x n* matrix. Finally, *matrix* fills the matrix by writing an index value to each location of the matrix. We chose matrix as our test subject because it easily demonstrates the affect of increasing allocated memory size on an application's run-time when using SCS protection.

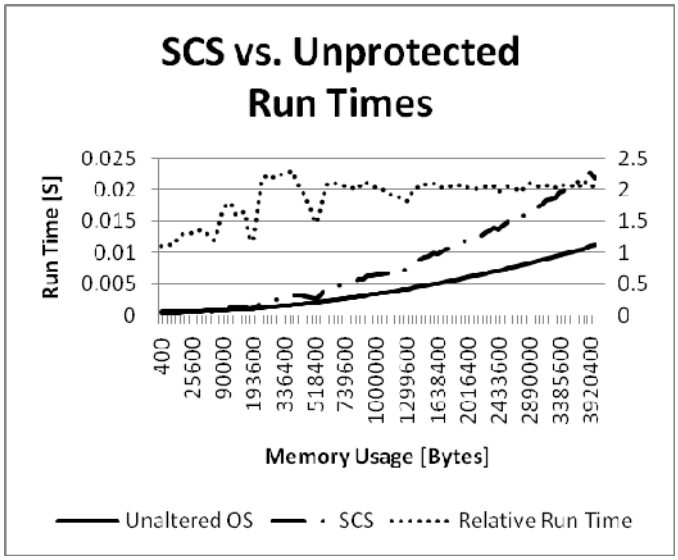

Figure 2: Measured Run Time SCS Vs. Unmodified Kernel

When running to collect data we always ran with square matrices, i.e. $m = n$. We ran *matrix* in a nested loop. The outer loop incremented the matrix size from 10 to 1000 by 10. The inner loop ran matrix 1000 times for each matrix size. In total we ran matrix 1 million times with SCS encryption enabled and 1 million times with SCS disabled for comparison. Each run was timed individually and the result was recorded in a file.

After execution we processed the run time data to prepare for plotting. First, we trimmed the run time outliers from the data. We trimmed the 2 smallest run times and 2 largest run times. We performed this trimming step because we found that some matrix size run time averages were skewed by individual runs which took orders of magnitude longer to execute than their peers. We conjecture that this occurs when a process is subjected to more than the ordinary number of context switches. This can occur if the machine becomes busy with another task. While we intended for the host to be lightly loaded while running the matrix program some interruptions still occurred. Next we averaged the remaining 996 samples for each matrix size.

Figure 2 offers a run time comparison between the SCS protected matrix application and the non-SCS version. Figure 3 shows 3 curves. The bottom two curves show the average run time for both the SCS and non-SCS runs plotted against the total memory size of the dynamically allocated matrix. The higher of the two run time curves plots the SCS run times and the lower curve shows the run times for the non-SCS runs. Both curves show increasing run-time as the size of the matrix increases. This is the expected behavior since *matrix* is a order-n complexity application. The separation between the SCS curve and the non-SCS curve, the difference in run time, also increases as the matrix grows larger. This occurs because the SCS overhead increases with the size of processes memory image while the context switch time for non-SCS applications is constant. The increasing separation between the run times is further illustrated by the top most curve of Figure 2 drawn with a dashed line, which shows the SCS run time divided by the non-SCS run time plotted against the total memory size of the dynamically allocated matrix. The Y-axis for this curve is labeled on the right hand side of the graph. For the collected data set the SCS runs range between 1.0 and 2.5 times slower than their non SCS counter parts. This ratio increases as the applications allocated memory size increases.

The results shown in Figure 2 were limited to a maximum of 4 million bytes of protected memory space. Above 4 memory bytes of memory usage the SCS overhead becomes too large and begins to overwhelm the application, eventually, causing the application to stall and never complete. This roll off limits this SCS architecture's effectiveness to protecting processes with relatively small memory usage. In [6] we offer a new architecture, which we call demand encryption/decryption, which eliminates this upper limit on the memory usage and improves SCS performance considerably.

D. Implementation Specifics

Implementing SCS requires patches to the Linux operating system. These patches occur in two primary locations. First, PCPP uses a custom binary format handler. This binary format handler loads PPELF executables into memory for execution. PPELF is our secure executable format briefly described in section *3A* (see [5] for a detailed description). In addition to loading the PPELF application the PPELF binary format handler fills certain needed data structures in the new processes task structure including setting a Boolean value to indicate the process is a PCPP process, creating an AES instance for the process, creating a key cache for the process, and finally inserting the executable's encryption key into the key cache.

In addition to the new code in the binary format handler SCS requires changes to the Linux kernel scheduler in the routine which implements the context switch. Here we insert code to encrypt the memory pages of outgoing PCPP processes and decrypt the memory pages of incoming PCPP processes. When an outgoing PCPP process is encountered, we learn this by checking for the Boolean set by the PCPP binary format handler, SCS loops through all of the memory pages assigned to the protected process encrypting pages which map the executable code and data segments, the process's heap, and the process's stack. Pointers to these memory pages are available via the application's task structure.

Since the context switch routine runs with interrupts

disabled there are several challenges when implementing this portion of SCS. First, the Linux kernel contains a large number of API which are available for use in kernel programs. However, many of these API are not available when interrupts are off since they may require the active process to sleep. Since we are working during the context switch allowing our process to sleep would hang the operating system as there would be no way to switch to another process. Furthermore, sleeping would conflict with the security requirements of SCS, since the whole intent of SCS is to encrypt/decrypt the protected processes context before another process gains control of the CPU. We were impacted in two primary areas by the lack of access to existing kernel API and processes. First, the Linux Kernel contains a built-in AES implementation. We were not able to use this AES implementation, and were therefore forced to add our own. Second, we were not able to use the Linux page fault system to convert virtual addresses to physical addresses. Instead we implemented our own virtual to physical address mapping API.

We used a counter mode implementation of AES to enable block-wise random access. The actual AES encryption and decryption routines were a modified copy of the AES implementation available in the Linux kernel. We copied and modified the AES routine to enable its use within the context switch routine. Most of changes were minor, simply removing calls to the Linux *might_sleep*. One change was significant, we were unable to use the hand optimized assembly routines which are available for our architecture, x86_64, and instead used a C based implementation. Our AES key length was 128-bits for our experiments, though 192-bit and 256-bit key lengths are supported.

We implemented SCS as a series of patches to version 2.6.20.6 [2] of the Linux operating system. We performed our experiments on a 64-bit AMD platform running at 3 GHz with 2 GB of DDR2 DRAM running at 800 MHz. During our experiments one of the two CPUs was disabled to emulate the performance on a single processor platform.

V. Future Work & Conclusion

Secure Context Switch (SCS) encrypts a process's context, i.e. its executable code, data segments, heap, and stack, when the process relinquishes control of the CPU and decrypts the context when a process regains control of the CPU. By placing this encryption/decryption step inside the operating system's context switch routine the context is isolated from any other process running on the same platform. We illustrated that SCS does result in a performance penalty which we quantified with experimental results from our working SCS implementation, which we implemented as a patch to the Linux operating system. With the implementation of SCS described here, overhead limits the practical amount of memory SCS can protect to less than 4 MB. We describe an alternative SCS architecture in [6] called demand encryption/decryption which removes this upper limit on memory usage and offers a significant speed improvement over the SCS architecture described in this paper by decrypting pages as they are accessed rather than each time the protected process regains control of the CPU.

Secure Context Switch offers a novel method for isolating application memory from other processes running on the same execution platform. SCS reduces this isolation problem down to one of protecting a set of encryption keys. Said another way, SCS encryption keys must be protected for SCS to be effective. PCPP encryption key protection [6] further reduces the problem to one of protecting a single key and then offers a solution to protecting a key on an open white box platform via a set of operating system enhancements.

In the near future, we plan to apply PCPP to various application problem areas. First, we plan to integrate PCPP protection into the Globus tool kit [7]. Next we plan to use PCPP to protect an ad-hoc networking client to ensure network traffic cannot be altered or monitored by nefarious users.

References

[1] Youngdale, E. 1995. Kernel Korner: The ELF Object File Format by Dissection. Linux J. 1995, 13es (May. 1995), 15

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

[3] Morris, T. Nair, V.S.S. PCPP: Private Computing on Public Platforms A New Paradigm in Public Computing, 2nd IEEE International Symposium on Wireless Pervasive Computing (ISWPC 2007), Feb. 2007

[4] Morris, T. Nair, V.S.S. PCPP: On Remote Host Assessment via Naive Bayesian Classification, IEEE International Parallel and Distributed Processing Symposium (IDPDS 2007), 26-30 March 2007, Pages:1-8

[5] Morris, T. Nair, V.S.S. Privacy Protected ELF for Private Computing on Public Platforms. Proceedings of the Third International Conference on Availability, Reliability, and Security (ARES2008), March 4-7, 2008.

[6] Morris, T. Private Computing on Public Platforms. Southern Methodist University. (PhD-thesis). 2008.

[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] SETI@Home, http://setiathome.berkeley.edu/

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

[10] Loscocco, P.A. Smalley, S.D. (2001). Meeting critical security objectives with Security-Enhanced Linux. In Proceedings of the 2001 Ottawa Linux Symposium