USENIX Association

Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems

Lihue, Hawaii, USA
May 18–21, 2003

Certifying Program Execution with Secure Processors

Benjie Chen Robert Morris
MIT Laboratory for Computer Science
{*benjie,rtm*}*@lcs.mit.edu*

Abstract

Cerium is a trusted computing architecture that protects a program's execution from being tampered while the program is running. Cerium uses a physically tamper-resistant CPU and a μ-kernel to protect programs from each other and from hardware attacks. The μ-kernel partitions programs into separate address spaces, and the CPU applies memory protection to ensure that programs can only use their own data; the CPU traps to the μ-kernel when loading or evicting a cache line, and the μ-kernel cryptographically authenticates and copy-protects each program's instructions and data when they are stored in the untrusted off-chip DRAM. The Cerium CPU signs certificates that securely identify the CPU and its manufacturer, the BIOS and boot loader, the μ-kernel, the running program, and any data the program wants signed. These certificates tell a user what program executed and what hardware and software environment surrounded the program, which are key facts in deciding whether to trust a program's output.

1 Introduction

Although research on the use of tamper-resistant hardware has been in progress for nearly 15 years [10, 11, 12, 7, 5], public concerns for issues such as copy protection and secure remote execution and the recent push in commodity secure hardware [2] suggest that the benefits of using secure hardware is now exceeding its overhead in complexity, performance, and cost. This paper describes a trusted computing architecture, Cerium, that uses a secure processor to protect a program's execution, so that a user can detect tampering of the program's instructions, data, and control-flow while the program is running.

This paper considers the following computation model. A user runs a program on a computer outside the user's control. The computer runs the program and presents the user with an output. The user wants to know if the output is in fact produced by an un-tampered execution of the user's program. We call this computation model *tamper-evident execution*. Tamper-evident execution enables many new useful applications. For example, a project that depends on distributed computation, such as SETI@home [1], can use tamper-evident execution to check that results returned by participants are produced by the appropriate SETI@home software.

The goal of Cerium is to support tamper-evident execution while facing strong adversaries. At the user level, Cerium should expose malicious users forging results of other users' programs without running them. At the system level, Cerium should expose buggy operating systems that allow malicious programs to modify the instructions and data of other programs. At the hardware level, Cerium should detect hardware attacks that tamper with a program's data while they are stored in memory, such as attacks on the DRAM or memory bus. Such strong adversaries prevent us from using software only techniques (e.g. Palladium [3] and TCPA [2]) to implement tamper-evident execution.

Cerium is designed to be open and flexible. Cerium does not limit which operating system or programs can run on a computer. Instead, Cerium tells a user what program executed and what hardware and software environment surrounded the program, so the user can decide whether to trust the program's output. This is in contrast to a more controlled and restrictive approach taken by some related systems [12, 7]. The IBM 4758 system [7], for example, provides a secure computing platform by allowing only operating systems and programs from trusted entities to run inside a secure co-processor. The co-processor establishes trust with a new entity (e.g. a bank) if other entities the co-processor already trusts (e.g. the manufacturer) vouch for the new entity. Thus, if a user wants to use the co-processor to run a program, the user must first establish trusts with several entities.

Nevertheless, this paper proposes an architecture that borrows several ideas from these systems. At the hardware level, Cerium relies on a 4758-like physically tamper-resistant CPU with a built-in private key. Unlike the 4758 co-processor, the Cerium CPU is the main processor in a computer and does not contain internal non-volatile storage. The Cerium CPU caches a portion of

a running program's instructions and data in its internal, trusted, cache. The remaining portions reside in untrusted external memory. Like Dyad [12], Cerium runs a μ-kernel in the secure CPU. The kernel's instructions and its crucial data are pinned inside the secure CPU's cache, so they cannot be tampered with. User-level processes that implement traditional OS abstractions (e.g. Mach servers) and virtualized operating systems (e.g. Windows running in VMWare) complete the μ-kernel-like operating system.

The Cerium CPU and the μ-kernel cooperate to protection programs from each other and hardware attacks. The μ-kernel partitions programs into separate address spaces, and the CPU applies conventional memory protection to prevent a program from issuing instructions that access or affect another program's data (cached or not). The CPU traps to the kernel when loading or evicting a cache line, and the kernel's trap handler cryptographically authenticates and copy-protects each program's instructions and data when they are stored in untrusted external memory. This technique allows the kernel to detect tampering of data stored off-chip.

The Cerium CPU reports what program is running and what hardware and software environment surrounds a program through certificates signed with the CPU's private key. The μ-kernel keeps a signature of the running program and includes the signature in the certificate. With a certificate, a user can detect if a different program binary was executed or if the computer is using a buggy kernel that cannot be trusted to protect the user's program.

Remaining sections of the paper describe system goals, related work, design, and applications.

2 System Goals

The main goal of the Cerium architecture is to provide tamper-evident execution of programs. Cerium protects a program's instructions, data, and control-flow during the program's execution so that they cannot be tampered by hardware attacks or other programs undetected. Another goal of Cerium is to allow trusted and untrusted operating system and processes to co-exist, all on the same CPU. Cerium reports the hardware and software environment to users so they can decide if the output of a program is in fact produced by that program. To help understand the design requirements, we describe a few examples.

Secure Remote Execution: Distributed execution of CPU-intensive programs can increase performance. Projects such as SETI@home [1] tap CPU cycles on idle computers scattered throughout the Internet. A problem with this computing model is that users cannot easily verify results obtained from an untrusted computer. A malicious user can, for example, return forged results without running the program. Cerium solves this problem by allowing a user to verify that an output is in fact produced by the user's program.

Copy Protection: Content distributors can use Cerium to enforce certain copyright restrictions. For example, an e-book's author can require customers to use Cerium, and distribute copies of the author's book so that each copy can only be viewed on a computer with a particular software configuration (i.e. with a given BIOS, boot loader, μ-kernel, and media player). A distributor can discover a Cerium computer's configuration from a certificate signed by the computer's secure CPU.

Secure Terminal: Users frequently check their e-mail by connecting to remote servers using untrusted terminals (e.g. at an Internet cafe). Although using tools such as `ssh` or SSL-based web login prevents passive adversaries sniffing data on a network, it does not prevent a bogus software on the untrusted terminal from stealing data. Cerium enables more secure login from untrusted terminals using the Cerium architecture. A user's trusted server authenticates the login software and the terminal's operating system, to make sure that the login software and the operating system appear on a list of software known not to steal data. As a result, the terminal approaches the safety of the user's own laptop.

3 Related Work

Previous research in secure processors and co-processors makes the use of a tamper-resistant CPU realistic. μABYSS [10], Citadel [11], and the IBM 4758 secure co-processor [7] place CPU, DRAM, battery-backed RAM, and FLASH ROM in a physically tamper-resistant package such that any tamper attempt causes secrets stored in the DRAM or battery-backed RAM to be erased. AEGIS [4] uses a processor that ties a secret to the statistical variations in the delays of gates and wires in the processor; an attack on the processor causes a chance in the processor's physical property, and therefore the secret. While no provably tamper-proof system exists, we believe current practices in building secure processors make physical attacks difficult and costly.

Dyad [12] and the IBM 4758 system [7] use tamper-resistant co-processors to provide trusted computing environments. Both systems allow only software from entities the co-processor trusts to run inside the co-

processor. The co-processor boots in stages, starting with the BIOS stored in the ROM. The software at each stage self checks its integrity against a signature stored in the co-processor's non-volatile memory. Each stage also authenticates the software for the next stage. Trusted entities install and maintain the software and their signatures. The co-processor establishes trust with an entity (e.g. a bank) if other entities the co-processor already trusts (e.g. the manufacturer) vouch for it. A trusted program running on the co-processor can also store encrypted data on external memory or disk. An advantage of the co-processor approach is that a user can connect a trusted co-processor using PCMCIA or USB; the user does not have to trust the microprocessor or the co-processor inside the computer the user is using. On the other hand, to write a program for a co-processor, the programmer must first establish trusts with several entities. Cerium provides a more open and flexible computing base. The Cerium architecture allows anyone to write operating systems and programs (buggy or not) for the secure CPU.

XOM [5] and AEGIS [8] also use physically tamper-resistant processors to support tamper and copy-evident computing. XOM and AEGIS do not to trust the operating system to protect programs from each other. Instead, the secure processor partitions cache entries and memory pages of different programs and the operating system in hardware/firmware. In contrast, Cerium depends on a μ-kernel to partition programs into separate address spaces and to authenticate and copy-protect each program's instructions and data when they are stored in untrusted external memory. Cerium securely identifies the μ-kernel, so a user can decide if the μ-kernel can be trusted to protect the user's program.

TCPA [2] promises to provide a trusted computing platform. TCPA uses a secure co-processor to store secrets and to perform cryptographic operations, but runs programs on a conventional microprocessor. Consequently, attacks on the DRAM and memory bus can alter the execution of a program undetected. Like Cerium, TCPA computes signatures of the system software as it boots up, and uses these signatures to enforce copy-protection. Palladium [3] is a software architecture that uses TCPA hardware. Palladium uses a small μ-kernel to manage applications that require security, much like Cerium.

4 Cerium

Cerium protects a program's execution using several techniques. Cerium relies on a physically tamper-resistant CPU with a built-in private key. The CPU runs all the software a computer uses. The CPU's tamper-resistant package protects a program's instructions and data from hardware attacks when they reside in the CPU's internal cache. A μ-kernel partitions programs into separate address spaces, and the CPU applies conventional memory protection to prevent a program from issuing instructions that affect data in another address space. The CPU traps to the kernel when loading or evicting a cache line, so the kernel can use cryptographic techniques to detect tampering of data stored off-chip. Upon request, Cerium tells a user what program is running, and what hardware and software environment surrounds the program, so the user can decide whether to trust the output of a program.

4.1 Tamper-Resistant CPU

The main component of Cerium is a tamper-resistant CPU, packaged in a way that physical attempts to read or modify information inside the CPU cannot succeed easily [4, 7, 10]. The CPU's tamper-resistant package protects its internal components, such as registers and cache, from hardware attacks. The CPU's internal cache is big enough (e.g. tens of megabytes) to contain a μ-kernel and the working set of data for most programs, but not big enough to contain whole programs. Programs that require more memory use untrusted external memory. The CPU traps to a kernel when evicting or loading a cache line, so the kernel trap handler can protect data stored in external memory (see Section 4.2).

Each CPU has a corresponding public-private key pair. The private key is hidden in the CPU and never revealed to anyone else (including software that runs on the CPU). The CPU reveals the public key in a *CPU certificate* signed by the manufacturer. A user trusts a CPU if the CPU certificate is signed by a trusted manufacturer. A CPU uses the private key to sign and decrypt data, and users use the public key to verify signatures and encrypt data for the CPU.

4.2 Operating System

At the software level, Cerium uses a μ-kernel to create and schedule processes onto the CPU and to handle traps and interrupts. The kernel runs in privileged mode; it can read or write all physical memory locations. Using page tables, the kernel partitions user-level processes into separate address spaces. The CPU applies conventional memory protection to prevent a program from issuing instructions that access or affect data in another address space. The page table of each address space is protected so only the kernel can change it. To prevent tampering of

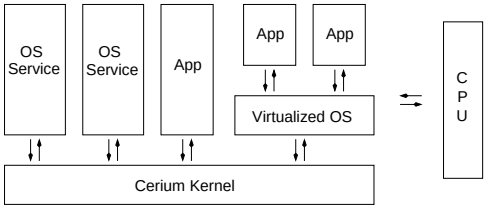

Figure 1: The organization of a Cerium system.

the kernel, the kernel text and some of its data reside in the secure CPU's cache and cannot be evicted.

The secure CPU traps to the μ-kernel when evicting data from its cache to external memory, or loading data from external memory into its cache. The kernel's trap handler decides, on a per-program basis, if and how the data should be protected. A program may ask the kernel to only authenticate or to both authenticate and copy-protect its instructions and data whenever they leave the secure CPU's internal cache. This technique is similar to the cryptographic paging technique used in Dyad [12].

The overhead of taking a trap on every cache event depends on a program's memory access pattern and miss rate, the cost of cryptographic operations, and the level of security a program demands. We believe that an increase in the size of a CPU's cache and using hardware-assisted cryptographic operations decrease the overhead. Furthermore, only programs that require integrity and/or privay protection take on these costs. We are currently investigating this issue.

Figure 1 shows what a Cerium system would look like. The μ-kernel and some user-level servers that implement OS abstractions form a complete operating system. Users can also run virtualized operating systems (e.g. Windows running in VMWare) in user space.

4.3 Booting

Cerium reports the hardware and software configuration of a computer, so a user can decide if the hardware and software can be trusted to protect the user's program. The CPU identifies each μ-kernel by the content hash of the kernel's text and initialized data segment. If the kernel is modified before the system boots, its content hash would change. Because a kernel's text and data reside inside the tamper-resistant CPU, they cannot be changed (e.g. by a DMA device) after the system boots. We refer to the kernel's content hash as the *kernel signature*.

A Cerium computer boots in several stages. On a hardware reset, the CPU computes the content hash of the BIOS and jumps to the BIOS code. Next, the BIOS computes the content hash of the boot loader, stored in the first sector of the computer's master hard drive, and jumps to the boot loader code. Finally, the boot loader code computes the μ-kernel signature, and jumps into the μ-kernel. Each stage uses a privileged CPU instruction to compute the content hash. The instruction stores the content hash in a register inside the CPU. The registers are protected so malicious programs cannot modify their content. This booting technique is similar to that of TCPA [2].

4.4 Running a Program

A program specifies its protection policy in its program header, so the μ-kernel knows how to protect the program's instructions and data when they are stored in external memory. There are three protection policies: no protection, authentication only, and copy-protection. A program asks the kernel to only authenticate its instructions and data if they do not need to be hidden from other programs. A program can also ask the kernel to copy-protect its instructions and data, so other programs cannot read their content.

When a program starts, the kernel first computes the content hash of the program's text and initialized data segment. The kernel uses this content hash as the *program signature*. A program signature uniquely identifies the program; if the program text or initialized data values change, the content hash would change as well. This technique only correctly protects programs using statically linked libraries.

4.5 Memory Authentication

This section describes how a μ-kernel handles authentication of data stored in untrusted DRAM. The kernel divides the entire physical address space into two regions. The *off-chip* region contains program data, such as text, data, and execution states, and some kernel data, such as page tables. The *on-chip region* contains the kernel's text, initialized data segment, and some data the kernel uses to authenticate data from the off-chip region. The on-chip region is pinned inside the secure CPU's cache so they cannot be evicted to external memory. Data from the off-chip region may be stored in external memory.

The μ-kernel efficiently authenticates data stored in external memory using a Merkle tree [6, 9]. A Merkle tree is a tree of hashes. Each intermediate node in the tree contains an array of *PA, hash* pairs, where PA specifies the physical address of one of the node's children, and hash is the cryptographic content hash of that child. Each leaf node stores hashes of data in external memory,

one hash for every 4K block (the size of a cache line on the CPU; this is acceptable because the CPU contains a large cache (e.g. tens of megabytes)). The root of the tree is stored in the on-chip region of the memory, so it cannot be modified by other programs or hardware attacks.

When the CPU traps to kernel to load data from external memory, the trap handler takes as its argument the physical address of the data. The trap handler uses the physical address to index the Merkle tree and find the corresponding leaf node. The trap handler computes the content hash of the data loaded into cache, making sure that the hash matches the one stored in the leaf node. When the leaf node is loaded into the cache, the trap handler verifies its integrity using the hash stored in the node's parent. This recursive process stops at the root of the tree. When the CPU evicts data from its cache, the kernel trap handler updates the Merkle tree accordingly.

4.6 Copy Protection

The CPU and μ-kernel can copy-protect a program's instructions and data while they are stored in external memory or on disk. We now describe how Cerium copy-protects a file, which could be a program's text or data.

Each copy-protected file has a corresponding *protection profile*. The protection profile contains a symmetric encryption key and signatures of trusted BIOS programs, boot loaders, kernels, and programs. A user encrypts the plaintext file using the symmetric key in the protection profile, then encrypts the profile using the Cerium CPU's public key. To open a copy-protected file, a program issues an instruction to ask the CPU to retrieve the symmetric key from the file's encrypted protection profile. The CPU returns the symmetric key only if the protection profile contains the current software configuration (i.e. signatures of the BIOS, boot loader, kernel and program). For example, the CPU refuses to return the decryption key if a malicious kernel, whose signature does not appear in the profile, is running. This technique is also used by TCPA and Palladium [2, 3].

A shell program that loads a copy-protected program stores the symmetric key of the new program in the on-chip memory region. When the CPU traps to kernel to load or evict a cache line for the new program, the trap handler uses this key to decrypt or encrypt the cache line.

4.7 Certifying Execution

Cerium is designed to be open and flexible. Cerium allows any μ-kernel or program to run on a computer, but reports what program is running and what hardware and software environment surrounds the program. A user can then decide if the identified hardware and software can be trusted to protect the user's program.

Cerium reports a computer hardware and software configuration in an *execution certificate*. Each execution certificate contains the CPU certificate, the content hashes of the BIOS and the boot loader code, the kernel signature, the program signature, and any data the program wants signed. For example, a program may also ask the kernel for content hashes of user-level OS services the program depends on. On a system call, the Cerium kernel creates a certificate and fills in the program signature and program data. The CPU fills in the rest of the certificate and signs the certificate with its private key.

Upon receiving a certificate, a user first extracts the CPU's public key and verifies that the CPU is made by a trusted manufacturer. The user also checks the certificate's signature. Next, the user checks if the signatures of the BIOS, boot loader, and μ-kernel in the certificate appear on a list of software the user trusts. Finally, the user checks if the program signature in the certificate matches the user's program. If the user trusts the CPU to correctly compute the hash of the BIOS, the BIOS to correctly load the boot loader and compute its hash, the boot loader to correctly load the μ-kernel and compute its signature, and the μ-kernel to correctly protect the user's program, then the user can trust the output of the program identified in the certificate.

5 Application Solutions

Secure Remote Execution: A user sends a program, the program's input, and a nonce to a remote computer. The program performs computation on the remote computer and obtains a signed execution certificate. The program includes in the certificate the hash of the nonce and the program's input and output. The program output and the certificate are then sent back to the user.

The fidelity of a program's output is determined in three steps. First, the user checks if the certificate identifies a trusted CPU manufacturer and a software configuration that the user trusts to protect the user's program. Second, the user checks if the certificate identifies a program signature that matches the signature of the user's program. Finally, the user checks if the hash of the nonce, the program input, and the received output matches the hash shown in the certificate. If all three conditions hold, then the output is in fact produced by the user's program.

Copy Protection: A content distributor takes three steps to copy-protect a file. First, the distributor sends a challenge to the customer's media player, and asks the media

player for an execution certificate that includes the hash of the challenge. The distributor uses the certificate to verify that the customer's hardware and software configurations can be trusted to not leave the copy-protected file unencrypted on disk or in external memory. Second, the distributor creates a protection profile and encrypts the file using the symmetric key in the profile. The distributor encrypts the profile using the public key of the customer's Cerium CPU. Finally, the distributor sends the encrypted file and the encrypted protection profile to the media player.

The media player asks the Cerium CPU to retrieve the decryption key from the protection profile. The Cerium CPU first decrypts the profile using its private key, then checks the current software configuration to make sure that it appears in the protection profile. If this check succeeds, the CPU returns the decryption key to the media player.

When the media player decrypts the encrypted file from the content distributor, the resulting plaintext initially resides in the secure CPU's cache. If the CPU evicts a block of the plaintext to external memory, the kernel's trap handler uses a session key that the media player generated to encrypt the evicted data.

Secure remote login: A user's trusted server must authenticate the hardware and software configuration of the untrusted terminal on behalf of the user. Before a login session, the login software (e.g. a `ssh` client) obtains a nonce from the trusted server and obtains an execution certificate from the untrusted terminal that includes the nonce. The login software forwards the certificate to the user's server. To verify the certificate, the server checks that the certificate is signed by a trusted manufacturer's CPU, that the untrusted terminal has a software configuration that can be trusted, and that the nonce in the certificate matches the one the server sent to the login software. If these conditions hold, the server returns a one-time response to the login software that the user can recognize as coming from the server. The user can then use the login software knowing that it will not steal any sensitive data. This solution does not guard against attacks on the computer's input interface, such as using a camera to monitor keyboard strokes.

6 Conclusion

This paper describes Cerium, a trusted computing architecture that provides tamper-evident program execution. Cerium uses a physically tamper-resistant CPU and a μ-kernel to protect programs from each other and from hardware attacks. Cerium reports what program is running and what hardware and software environment surrounds the program, so the a user can decide whether to trust a program's output.

Acknowledgments

We thank PDOS members and Satya for their comments.

References

[1] SETI@home. http://setiathome.ssl.berkeley.edu/.

[2] TCPA. http://www.trustedcomputing.org/.

[3] A. Carroll, M. Juarez, J. Polk, and T. Leininger. Microsoft Palladium: A business overview, August 2002. Microsoft Press Release.

[4] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Controlled physical random functions. In *Proceedings of the 18th Annual Computer Security Applications Conference*, December 2002.

[5] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 168–177, 2000.

[6] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 135–150, October 2000.

[7] S. W. Smith and S.H. Weingart. Building a high-performance, programmable secure coprocessor. In *Computer Networks (Special Issue on Computer Network Security)*, volume 31, pages 831–860, April 1999.

[8] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. The AEGIS processor architecture for tamper-evident and tamper resistant processing. Technical Report LCS-TM-461, Massachusetts Institute of Technology, February 2003.

[9] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Hardware mechanisms for memory authentication. Technical Report LCS-TM-460, Massachusetts Institute of Technology, February 2003.

[10] S. Weingart. Physical security for the μABYSS system. In *Proceedings of the IEEE Computer Society Conference on Security and Privacy*, pages 38–51, 1987.

[11] S. White, S. Weingart, W. Arnold, and E. Palmer. Introduction to the Citadel architecture: security in physically exposed environments. Technical Report RC16672, IBM Thomas J. Watson Research Center, March 1991.

[12] B. Yee. *Using secure coprocessors*. PhD thesis, Carnegie Mellon University, May 1994.