RC25287 (WAT1205-070) May 23, 2012
Computer Science

IBM Research Report

SecureBlue++: CPU Support for Secure Execution

Rick Boivie
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

SecureBlue++: CPU Support for Secure Execution

Rick Boivie, Ph.D.
IBM Distinguished Engineer
T. J. Watson Research Center
rhboivie@us.ibm.com

Introduction

IBM Research has been developing an innovative secure processor architecture that provides for verifiably, secure applications[1,2]. The architecture protects the confidentiality and integrity of information in an application so that 'other software' cannot access that information or undetectably tamper with it. In addition to protecting a secure application from attacks from "outside" the application, the architecture also protects against attempts to introduce malware "inside" the application via attacks such as buffer overflow or stack overflow attacks.

This architecture provides a foundation for providing strong end-to-end security in a network or cloud environment.

The architecture, which we call SecureBlue++, builds upon our SecureBlue secure processor technology[3] which has been used in 10's of millions of CPU chips to protect sensitive information from physical attacks. In a SecureBlue system, information is "in the clear" when it is inside the CPU chip but encrypted whenever it is outside the chip. This encryption protects the confidentiality and integrity of code and data from physical probing or physical tampering.

SecureBlue++ builds upon SecureBlue. Like SecureBlue, SecureBlue++ protects against physical attacks but SecureBlue++ uses "fine-grained" SecureBlue-like crypto protection that also protects the confidentiality and integrity of information in an application from all the other software on a system. Importantly, it does this in a way that is largely transparent to applications.

---

[1] CPU Support for Secure Executables, P. Williams, R. Boivie, Trust 2011, 4th International Conference on Trusted Computing, June 22-24, 2011, Pittsburgh, Pennsylvania

[2] SecureBlue++: CPU Support for Secure Execution, R. Boivie, P. Williams, 2nd Annual NSA Trusted Computing Conference and Exposition, September 20-22, 2011, Orlando, Florida

[3] http://www-03.ibm.com/press/us/en/pressrelease/19527.wss

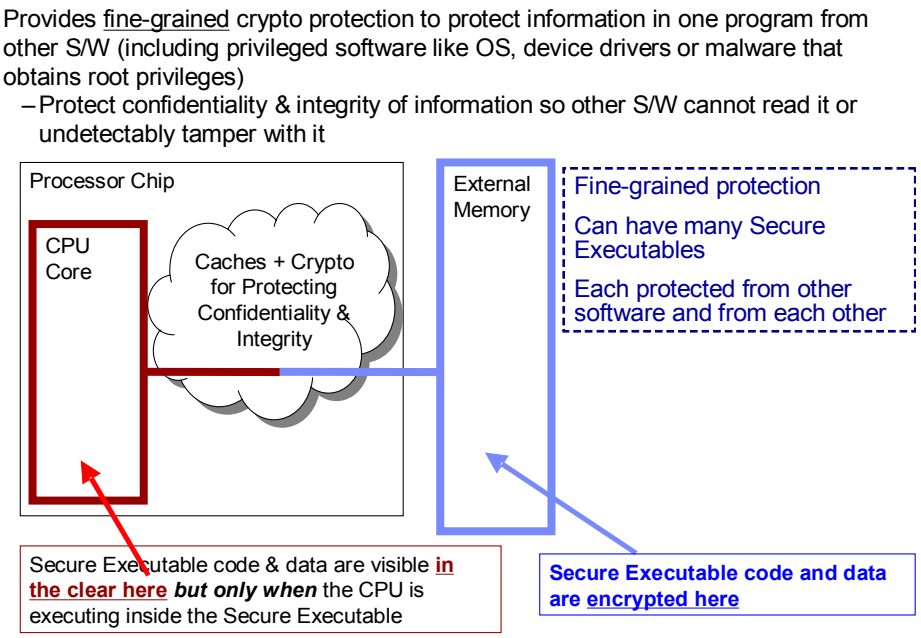

Figure 1: Secure Blue++

Technical Overview

When an application's information is outside the CPU (e.g. when the application is in the file system prior to execution; or when it's in memory or in the paging system during execution) it is encrypted under keys that are not available to any other software and an integrity tree is used to detect tampering. Integrity protection (as well as confidentiality protection) is continuous -- not just at application launch time. When an application's information is inside the CPU (e.g. in on-chip caches), it is in the clear but context labels prevent other software from accessing or tampering with that information.

Since an application's information is encrypted whenever it is outside the CPU and since other software cannot access an application's cleartext information inside the CPU, other software such as the page demon or malware can only see the encrypted form of an application's information. Thus an application's information is protected from all the other software on the system including privileged software like the operating system, device drivers or applications running with root privileges -- or malware that obtains root privileges by exploiting a vulnerability in privileged software.

Unlike systems that "measure" boot code, operating system code and device driver code at system boot time and then trust this code during execution, in our architecture, an application does not need to trust the operating system or any other software. An application in a SecureBlue++ system uses operating system services for scheduling, paging, I/O, interrupt handling etc. but it does not trust the operating system with any sensitive information. This is analogous to the way we use the Internet today when we make an online purchase via https. We use the Internet to transport our packets but we

don't give it access to any of our sensitive information. A malicious network operator may delay or drop our packets, but it cannot violate their confidentiality or integrity.

A SecureBlue++ system thus minimizes the amount of code that needs to be trusted. SecureBlue++ allows us to know with a high level of confidence that information in an application is secure without having to trust millions of lines of operating system, device driver and boot code or other applications. The only code that an application needs to trust is the code within the application itself.

As discussed above, the information in an application is cryptographically protected when the application is in a file system as well as throughout execution. This information is also protected while the application is in-transit prior to installation in the file system. Since the information in an application is always protected, we can "compile in" a root (or roots) of trust that cannot be stolen or tampered with. Since a "compiled in" private key can't be stolen, we can provide strong protection against key theft and identity spoofing. And since a "compiled in" digital certificate can't be tampered with, we can provide strong protection for a chain of trust that allows us to determine the authenticity of other entities in a network. Importantly, these roots of trust are protected from 1) an adversary that has root access to the system or 2) a malware-infected operating system.

Secure Applications (also known as Secure Executables)

In our architecture, a secure application consists of 1) a cryptographically protected region containing encrypted code and data, 2) an initial integrity tree that protects the integrity of the protected region and 3) an unprotected region that includes "communication" buffers and a loader that includes a new CPU instruction. The new instruction, 'esm', causes the CPU to 'enter secure mode' and process the encrypted code and data. Encrypted code and data is decrypted and checked for integrity in hardware as it is brought into the CPU chip (i.e. into an on-chip cache) from external memory; and data is encrypted and integrity tree values are updated as information (i.e. dirty cache lines) are written out to external memory. The operand of the esm instruction, which includes the key[4] for accessing the encrypted code and data as well as the initial value for the root of the integrity tree, is itself protected by a 'system key' that is not available to any software.

Thus other software, such as 'memory scraping'[5] malware cannot access or undetectably tamper with information in a secure application. (The esm operand is protected by a public/private key pair. The build machine encrypts the esm operand with the public key of the target machine. The private key, which might be installed in the target machine at manufacture time, is used by the esm instruction at run-time to decrypt the operand.) In addition to loading the application's encryption key and integrity root into the crypto hardware, the esm instruction also allocates an ID, the Executable-ID or EID, that the hardware and the operating system will use to refer to this 'secure executable'.

[4] Or keys

[5] SANS Institute: Pervasive Memory Scraping Among Most Dangerous of Attacks, http://www.messagingwire.com/aev-901.aspx

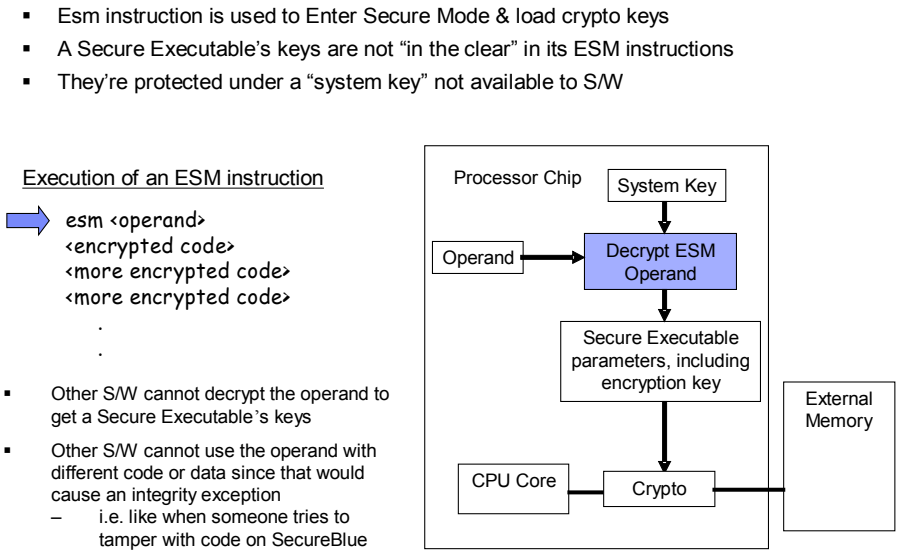

Figure 2: Enter Secure Mode Instruction

As discussed above, information is stored in the clear in on-chip caches so cryptographic operations only occur when information moves between on-chip caches and external memory. In similar fashion, the checking and updating of integrity values only occur when information moves between on-chip caches and external memory. Thus the cryptographic and integrity overhead is close to 0 when a secure application is getting cache hits.

Using the Operating System without Trusting the Operating System

As discussed above, a secure application (or secure executable) uses operating system services but does not trust the operating system with any sensitive information.

In the paging subsystem for example, the operating system moves an application's pages between memory and disk but the pages are encrypted and the operating system has no access to the encryption key(s).

Another example is in network I/O. An application uses buffers in the unprotected region to send packets to and receive packets from a remote system. Since these buffers are in the unprotected region, they are not protected by the SecureBlue++ crypto. Thus the operating system and the remote system "see" the same information that the secure application "sees". On the other hand, if the contents of a buffer were cryptographically protected by the SecureBlue++ crypto protection, the operating system and the remote system would not be able to "see" the content that the secure application "sees".

Since the CPU decrypts information that moves from a protected region in external memory into the CPU and since the CPU does not encrypt information that moves from the CPU to an unprotected region in external memory, the act of copying information

from the protected region to the unprotected region has the effect of decrypting the information. Information in this region can be sent to a remote system and the remote system will be able to read it. Similarly the act of copying information from the unprotected region to the protected region has the effect of encrypting it under the secure application's encryption key. This is useful when information is received from a remote entity.

Note that although the unprotected region is not protected by the SecureBlue++ encryption, information in that region can still be cryptographically protected. Standard communications security mechanisms such as SSL or TLS can be used to provide end-to-end security. If a message that is to be sent to a remote system is encrypted under TLS before it is moved into the unprotected region, the message will be protected in the unprotected region in the same way that it is protected while it is traveling across a network. If, at the receiving end, the message is moved to the protected region before the TLS decryption, the message will have strong protection end-to-end with no point of vulnerability along the way.

Note that the keys that a secure application uses for TLS or other communications security will be protected so other software including the operating system will not be able to access those keys or the packets protected by those keys, or undetectably tamper with either the keys or the packets. Keys inside a secure application can also be used to protect information stored in a file system, and other software including the operating system will not be able to access those keys or the contents of the files protected by those keys.

System call "wrappers" can be linked with a secure application so that the application does not need to know about the protected and unprotected regions. These "wrappers" are library functions that invoke the actual system calls and copy the contents of buffers between the protected and unprotected regions in a way that is transparent to the application while allowing the application to communicate intelligibly with remote systems.

In addition to hiding some complexity from the programmer, the wrappers also allow us to convert an existing application into a secure application in a way that is largely transparent to the application program.

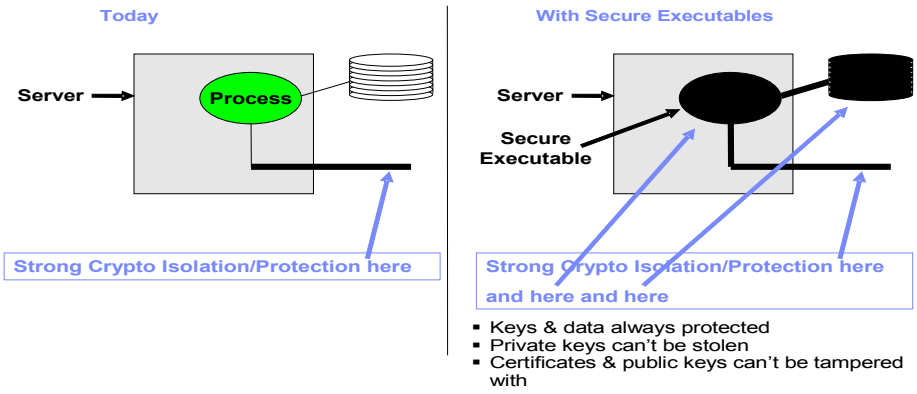

Figure 3: End-to-end protection of sensitive information with Secure Blue++

Secure Context Switching

As discussed above, the operating system, device drivers and other software do not have access to a secure executable's information. When a secure executable is interrupted, the CPU hardware securely saves certain information before the OS or device drivers get control including the contents of the general purpose registers and the secure executable's encryption key and integrity root. When the operating system gets control, it has no access to any of this information (or the contents of the secure executable's cryptographically protected 'protected region') but it does have access to the EID of the secure executable that was interrupted -- which is available in a (new) register. The operating system uses a new instruction, *restorecontext*, to securely restore and dispatch a previously suspended secure executable. The operand of this instruction is the EID of the secure executable to be dispatched[6].

Software Build Process

The make process for a secure executable / secure application doesn't require any changes to programming languages, compilers or link-editors. An application is compiled in the usual manner and then linked with the wrappers discussed above and with a loader that includes the esm instruction (with a placeholder for the esm operand). An integrity tree is built for the code and data that will comprise the protected region and the protected region is then encrypted with a randomly generated symmetric key. The symmetric key and the root of the integrity tree are then encrypted under the public key of the target system to form the operand of the esm instruction. Finally, the integrity tree is combined with the protected region and the loader to form the complete secure executable binary.

[6] A third new instruction, deletecontext, is used by the operating system at process exit to indicate to the CPU that a secure process has completed execution and that the CPU can de-allocate its EID and free any resources associated with that EID.

Protecting from "Attacks from the Inside" such as Stack and Buffer Overflow

As discussed above, SecureBlue++ can protect the confidentiality and integrity of information in an application from "external attack" so that "other software" cannot access or undetectably tamper with information inside the application. It can also protect an application from an "internal attack" in which an attacker attempts to exploit a vulnerability in an application's interfaces to get the application to execute code of the attacker's choosing via a software attack such as a buffer overflow or stack overflow attack.

As discussed above, the esm instruction enables the use of cryptographic hardware that protects the confidentiality and integrity of information in an application from "external attack". The esm instruction also establishes the address range that will be protected by the cryptographic hardware. As discussed previously, some portions of the address space (e.g. communication buffers) are not cryptographically protected.

To protect against attacks such as stack overflow and buffer overflow, the esm instruction will also set up two additional address ranges for an application. One of these will tell the CPU hardware that the address range corresponding to the application's code is, from the application's perspective, "read-only". The other will tell the hardware that the address range corresponding to the application's data, stack and heap is "no-execute". Thus if an attacker attempts to exploit a bug in an application so that the application will attempt to write into the code region or execute instructions from the data region, the attempt will fail.

As in the case of the encryption key and integrity root registers, the address range registers will be set by the CPU hardware in the execution of the esm instruction. These registers will not be accessible to software and they will be saved and restored by the CPU hardware during a context switch along with the encryption key and integrity root registers.

Protection Options

As discussed previously, SecureBlue++ can protect the confidentiality and integrity of an application's code and data. But other options are possible. While the integrity of code and data will always be important, in some cases one may not need or want cryptographic confidentiality protection -- for code or for data. A SecureBlue++ system could be designed so that

- The integrity of code and data is protected in all secure executables[7]
- The confidentiality of data is protected in secure executables that require that protection
- The confidentiality of code is protected in secure executables that require that protection.

Depending on application requirements and "make flags", secure executables could be built with various combinations of protection[8]. The particular combination that would be

[7] A SecureBlue++ system would still support "regular applications" as well as secure executables.
[8] This would involve some additions to the software build process discussed previously.

used for a given secure executable would be communicated to the CPU via the esm operand.

Deployment Considerations for Secure Executables

The esm operand of a secure executable is encrypted using the system key of the target machine. This adds some complexity to the process of building and distributing software since different target machines will normally have different system keys. The build machine could build a unique binary for each target machine. Alternatively the build machine could build and send a single binary to all the targets. This approach would leverage a special Deployment Server (DS) process on the target machine.

The DS would itself be a secure executable that protects sensitive information -- including software that it receives from a build machine.

The build machine would compile and link the software that will comprise a secure executable and securely send the software to the DS process on the target machine via standard communications security mechanisms such as SSL or TLS.

The DS will generate an encryption key for the received software and encrypt the software under this key. It will then encrypt the esm operand, which includes this encryption key and the initial integrity root, under the system key of the target system.

Thus the transmitted secure executable software is protected by standard communications security mechanisms until it is safely inside the DS secure process and it will be protected inside the DS process until it has been encrypted and customized for the particular target machine.

Once the received software and its esm operand have been encrypted, the received software is a secure executable that can be safely installed in the file system.

Thus the software is always protected from the time it leaves the build machine until it is safely installed in the file system on the target machine with no point of vulnerability along the way.

Note that the DS could also use digital certificates or public keys to validate a digital signature on received software to ensure that any software received is from a legitimate trusted party. Since the DS process is itself a security executable, the integrity of these digital certificates or public keys would be protected from tampering.

Similarly, secure executable mechanisms and digital certificates on the build machine would allow the build machine to validate the trustworthiness of the hardware of the target machine.

Current Status

We've implemented an initial prototype of a SecureBlue++-enabled CPU on our Mambo CPU simulator and we've run (a slightly modified) Linux (that uses the *restorecontext* and *deletecontext* instructions discussed above) and some simple demo applications on this prototype in a real network. In the demonstration, we show how a simple credit-card-authorization application can be built as either a "plain vanilla" application or as a secure executable using the same source code. We then demonstrate that malware can obtain sensitive information from the plain vanilla application (either while it's in the file

system prior to execution or while it's in execution), while malware cannot obtain any sensitive information from the secure executable.

Summary

We have been developing an innovative secure processor architecture that provides for verifiably, secure applications. The architecture protects the confidentiality and integrity of information in an application so that 'other software' cannot access that information or undetectably tamper with it. In addition to protecting a secure application from attacks from "outside" the application, the architecture also protects against attempts to introduce malware "inside" the application via attacks such as buffer overflow or stack overflow attacks.

This architecture minimizes the amount of code that needs to be trusted. An application uses operating system services but does not trust the operating system or device drivers with sensitive information. The architecture allows us to know with a high level of confidence that information in an application is secure without having to trust millions of lines of operating system, device driver and boot code or other applications. The only code that an application needs to trust is the code within the application itself.