Effective Implementation of the Cell Broadband Engine™ Isolation Loader

Masana Murase
IBM Research, Tokyo Research Laboratory
1623-14 Shimotsuruma
Yamato-shi, Kanagawa-ken
242-8502 Japan
mmasana@jp.ibm.com

Kanna Shimizu
IBM Corporation, IBM Systems & Technology Group
One Rogers St, Cambridge, MA 02142
kannas@us.ibm.com

Wilfred Plouffe
IBM Research, Almaden Research Center
650 Harry Road San Jose, CA 95120
plouffe@almaden.ibm.com

Masaharu Sakamoto
IBM Research, Tokyo Research Laboratory
1623-14 Shimotsuruma
Yamato-shi, Kanagawa-ken
242-8502 Japan
sakamoto@jp.ibm.com

ABSTRACT
This paper presents the design and implementation of the Cell Broadband Engine™ (Cell/B.E.) isolation loader which is a part of the IBM Software Development Kit for Multicore Acceleration [14]. Our isolation loader is a key component in realizing secure application boot and encrypted application execution. During the application load process, the isolation loader fetches, validates, and decrypts a Synergistic Processor Element (SPE) executable, establishing a chain of trust from the hardware to the application. Since not all applications are SPE executables, we also introduce a general solution. This is a verification service framework in which all applications including system functions can be verified by the isolation loader immediately before execution.

We have applied several novel implementation techniques to the isolation loader. The countermeasure implemented in our isolation loader against the substituted-ciphertext attack is given and our staging technique to allocate contiguous working areas for applications is also introduced. The load overhead of this loader including application fetch, validation (RSA-2048/SHA-1), and decryption (RSA-2048 and AES) is less than 50 milliseconds on the 2.8 GHz IBM PowerXCell 8i processor. This overhead is reasonable compared with the 500-millisecond 2048-bit RSA signing needed by the Trusted Platform Module chips [3].

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Security and Protection

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS’09, November 9–13, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-352-5/09/11 ...$10.00.

General Terms
Design, Performance, Security

Keywords
Cell Broadband Engine™, encrypt-then-sign, isolation, multicore, substituted-ciphertext attack

1. INTRODUCTION
Although modern CPUs with multiple cores are the predominant trends for enhancing performance and power efficiency, the multi-core feature has another aspect for computer usage – isolation. In 2005, Percival [6] reported on a security flaw of Hyper-Threading Technology by which malicious software could steal secret data such as encryption keys or passwords from caches or registers. This is because two threads, one a legitimate thread handling a secret and the other a malicious thread observing the legitimate thread, can share caches and registers when they are running on the same processor core. To address such security risks, it is necessary to prevent illegal and unexpected access to shared resources. The multi-core system allows us to confine each thread to the corresponding cores with concurrent execution of those threads. In particular, if there is no dependency among threads, we can benefit from both performance and security enhancements. We assume three hardware security features to realize such a secure environment: (1) on-chip memory isolation, (2) runtime secure boot, and (3) decryption during the software boot.

The Cell Broadband Engine™ (Cell/B.E.) [4] processor is one of the multi-core processors supporting such capabilities. As introduced in [21], the security features are called the secure processing vault1, the runtime secure boot, and the hardware root of secrecy, respectively. Unlike competing security solutions, this design is unique in that even if the substituted-ciphertext attack

1 Also referred to as the Synergistic Processor Unit (SPU) isolation mode
supervisory software such as the operating system or the hypervisor is compromised, the process isolation is guaranteed. In contrast, most security architectures rely on the perpetual integrity and security of their supervisory software to protect and separate the processes. With these features, we can create an isolated and secure domain on each core which is independent of the traditional insecure domain where the operating system or the hypervisor is running.

While [21] focuses on the security hardware architecture of the Cell/B.E. processor only, we present the detailed design and implementation of our security software stack for the Cell/B.E. processor in this paper. Our work provides a secure software-based application loader minimizing the application load overhead on top of the Cell/B.E. processor. The Cell/B.E. processor validates and decrypts the Synergistic Processor Element (SPE) isolation loader first. Afterward, the authorized loader validates and decrypts a signed and encrypted application (secure application) every time the secure application is initiated. In this way, a chain-of-trust is established and maintained from the hardware layer at the bottom to the application layer. This layered structure gives us flexibility and portability for application development. The software-based application loader provides separation of the hardware and the application layer so that if one changes (for example, a different cryptographic algorithm is used), the other is not affected.

Unlike the prior approaches, we use the encrypt-then-sign policy, but not a naive one for the application signing and encryption. This policy has a large advantage in the response time to detect tampering. Since a verifier with this policy validates the target application prior to the decryption, that verifier can release the CPU resource owned by this process to another process as quickly as possible when it detects tampering. In contrast, the sign-then-encrypt policy needs both decryption and validation operations to detect tampering. Sometimes, the decryption process can be time-consuming, which occurs in our experiments. We also introduce a novel staging technique to allocate contiguous working areas for applications. This technique is much of importance when the isolated memory space is limited.

Prior research on secure-main processors such as AEGIS [22], XOM [16], and Cerium [5] also provided the same hardware security features. Unlike our approach, this work used the sign-then-encrypt policy to generate a secure application. The sign-then-encrypt policy is an effective way to cope with a substituted-ciphertext attack [9], but it takes a long time to detect tampering because both decryption and validation must be performed to detect tampering in the sign-then-encrypt approach. This approach is reasonable if a verifier is implemented in the hardware and the overhead of the cryptographic operations can be ignored. However, we need to care about the performance overhead in addition to the security if we use a software authentication layer to provide flexibility and portability of applications. The secure main processor work does not address this concern.

Prior work on secure co-processors [27, 18, 26] is another approach to implement the isolated domains. The main purpose of this approach is to validate the software stack without changing the existing main processors. Once it has been ensured that a system has not been tampered with, then the authorized supervisory software such as the operating system or the hypervisor divides the runtime environment into two or more domains: one is the secure domain, and any others are non-secure domains. Thanks to the ring protection mechanism [13] implemented in the main processor, the secure domain is guaranteed to be isolated from the other domains. Compared with the secure main-processor work, the secure co-processor approach does not care about malicious supervisory software which might be hacked or compromised. Also, these secure co-processor systems are missing the encrypted application execution. How to implement signed and encrypted application execution in an efficient way is still an open question.

The remainder of this paper is structured as follows. In Section 2, a brief introduction to the Cell/B.E. processor is given as background for this paper. In Section 3, a secure application load technique and a contiguous working area allocation technique are proposed to realize a high-performance and secure application loader. Section 4 describes the performance evaluation of the SPE isolation loader and a code verification service, which is an extension of the SPE isolation loader, is implemented in Section 5. Section 6 clarifies the difference between our approach and the existing technologies. Finally we summarize our work in Section 7.

2. CELL BROADBAND ENGINE™ OVERVIEW

The Cell/B.E. is a multiprocessor core architecture (see Figure 1) [4]. The cores are heterogeneous and there are two kinds of cores on each chip. The principal core, the 64-bit Power Processor Element (PPE), is a PowerPC processor that has the supervisory role. It is the PPE that executes the operating system and manages the allocation of most system resources, including the SPEs. The other type of core on a Cell/B.E. is the SPE consisting of a Synergistic Processor Unit (SPU) and a Direct Memory Access (DMA) engine. In the current implementation, there are 8 SPEs per chip. The SPEs are the computational workhorses: a RISC-style single instruction, multiple data (SIMD) instruction set, wide and large (128 128-bit) register files, and 256 KB of physically dedicated private memory, called the Local Store (LS), for each SPE [10]. The high bandwidth Element Interconnect Bus (EIB) connects these processor cores to each other and to the off-chip system memory and I/O.

![Figure 1: The diagram of the Cell Broadband Engine architecture](image-url)
The SPE plays a key role in the Cell/B.E. architecture. One of its distinguishing features is its private memory called the Local Store (LS). The SPE fetches instructions from the LS and loads or stores data to and from the LS. However, the LS is not a hardware-managed cache. Instead, the LS memory region is mapped in the system memory map, and software, either the software running on the PPE or the software thread executing on the SPE, is expected to explicitly transfer code and data into the LS via DMA (Direct Memory Access). The transfers can occur with any resource on the EIB, such as main memory, an LS of an other SPE, or I/O devices.

In contrast to other processor architectures, the Cell/B.E. architecture provides three features specifically for increased security: the secure processing vault, the runtime secure boot, and the hardware root of secrecy. An SPE running in a special hardware mode, the isolation mode, is effectively disengaged from the bus, and by extension, from the rest of the system. When in this mode, the LS of the SPE, which contains the application’s code and data, is locked for use only by the SPE and cannot be read or written to by any other software, even including code in the ring0 mode of the ring protection. Therefore, even if the operating system kernel or the hypervisor is hacked and controlled by an adversary, he will still not be able to steal or manipulate the data or code in the isolated SPE LS. A small area of the LS is left open and data can still be brought in or taken out via DMA. This window is needed so that even in isolation mode the SPE can continue to bring in additional data and code.

Before an application can execute on an isolated SPE, the Cell/B.E. verifies the application integrity using a key embedded in the hardware, the runtime secure boot. This authentication mechanism itself is implemented in hardware, and therefore cannot be manipulated or skipped. When isolation mode is requested for a particular application thread, the SPE can continue to bring in additional data and code.

In the PPE isolation runtime, we have the Linux kernel and libspe2 to support the interfaces to enable the isolation mode. For programming, it is easy to port a regular Cell/B.E. application to isolation mode, since the change visible to the application programmers is just setting one flag. The two programming models are compared later in this section.

The most important part is the SPE isolation runtime. The SPE isolation runtime consists of three layers: the hardware authentication layer, the software authentication layer, and the application layer. The hardware authentication layer includes the cryptographic algorithms to verify and decrypt the upper layer, the SPE isolation loader. The software authentication layer includes the cryptographic algorithms to verify and decrypt the SPE secure application. The SPE secure application running on top of the layered model is invoked after a chain-of-trust is established from the hardware authentication layer to the application layer.

The software layer provides separation of the hardware and the application layer so that if one changes (for example, a different cryptographic algorithm is used), the other is not affected. In addition, it is easy to change signing algorithms, encryption algorithms or keys implemented in the SPE isolation loader depending on application requirements. Application programmers can benefit from this layered structure for portability, flexibility, and security in their applications.

Instead of supporting the encrypted application execution, obfuscation techniques [7, 24] can be applied to hide the application secrets embedded in the application images from malicious parties. However, these are not fundamental solutions. Therefore, we recommend encrypted application execution by the SPE isolation loader.

3. SPE ISOLATION LOADER DETAILS

First, the introduction of our software stack for the Cell/B.E. isolation environment is given in this section. With our software stack, application programmers can easily program an isolated application with the same programming paradigm as the regular Cell/B.E. applications. Then, several novel design and implementation methods used in the SPE isolation loader are described.

3.1 Software Stack

Figure 2 depicts our software stack consisting of the build tools, the PPE isolation runtime and the SPE isolation runtime. The ‘spu-isolated-app’ is one of the build tools used to sign and encrypt an ELF binary image for the SPEs. We call a new binary image produced by the spu-isolated-app in such a way as an SPE secure application. Note that the SPE secure application in the signed and encrypted form is structured in the original binary format. This format is described in Section 3.3.

The SPE isolation runtime consists of three layers: the hardware authentication layer, the software authentication layer, and the application layer. The hardware authentication layer includes the cryptographic algorithms to verify and decrypt the upper layer, the SPE isolation loader. The software authentication layer includes the cryptographic algorithms to verify and decrypt the SPE secure application. The SPE secure application running on top of the layered model is invoked after a chain-of-trust is established from the hardware authentication layer to the application layer.

The software layer provides separation of the hardware and the application layer so that if one changes (for example, a different cryptographic algorithm is used), the other is not affected. In addition, it is easy to change signing algorithms, encryption algorithms or keys implemented in the SPE isolation loader depending on application requirements. Application programmers can benefit from this layered structure for portability, flexibility, and security in their applications.

Instead of supporting the encrypted application execution, obfuscation techniques [7, 24] can be applied to hide the application secrets embedded in the application images from malicious parties. However, these are not fundamental solutions. Therefore, we recommend encrypted application execution by the SPE isolation loader.

3.2 Comparison of Programming Models

In this section, we compare the two programming models, the regular Cell/B.E. programming model and the secure Cell/B.E. programming model. Because of the heterogeneity of the Cell/B.E., application developers generally write two types of source code: one for the PPE and the other for the SPEs. Since the instruction set of the PPE is different from that of the SPEs, it is necessary to generate both PPE and SPE executables that are dispatched to the appropriate cores. For the simplicity of application execution, both binaries are integrated into one binary by a linker, ppu-ld provided as part of the GNU toolchain [11].
It is important to allow Cell/B.E. users to develop secure SPE applications with the same programming paradigm as used for regular Cell/B.E. applications. Figure 3 compares the two programming models. The left of the figure shows the execution flow of the regular Cell/B.E. application, while the execution flow of the SPE secure applications is illustrated on the right.

In the regular case, here is how the SPE application is started. After a PPE executable is initiated in Steps 1 and 2, libspe2 and the Linux kernel launch the SPE executable embedded in the PPE executable (Figure 3 A(i)). In Step 3, the SPE executable is transferred from the system memory to the target LS (Figure 3 A(ii)). In Step 4, the Linux kernel writes an SPU-Run request to the specific register. In Step 5, the SPE executable starts from its entry point (Figure 3 A(iii)).

In contrast, in the isolation case, although the binary image format of the SPE secure application is different from the regular SPE application, the overall execution flow is similar to the regular flow. In Steps 1 and 2, the PPE-SPE integrated application uses libspe2 and the Linux kernel to start the SPU isolation mode (Figure 3 B(i)). In contrast to the regular case, the hardware fetches the SPE isolation loader first after the mode is switched to the SPU isolation mode. At the same time, the hardware validates and decrypts the SPE isolation loader in Step 3 (Figure 3 B(ii)). In Steps 4 and 5, the validated SPE isolation loader fetches, validates and decrypts the SPE secure application (Figure 3 B(iii)). After that, as Figure 3 B(iv) shows, only the validated SPE secure application starts running in Step 6.

### 3.3 SPE Secure Application Format and Its Transformation

An SPE secure application has the special format, because we need information about the integrity check and the encrypted part in the application validation and decryption process. Similar to the previous work on [8], the binary format of an SPE secure application is illustrated on the right side of Figure 4. The spu-isolated-app tool transforms the SPU ELF binary format (SPE executable) to this binary format.

As the left hand side of Figure 4 shows, spu-isolated-app extracts all of the code and data segments including the relocatable segments and read-only segments from the SPU ELF binary in the first step. Since we sign and encrypt the final memory map image, the bss section is also extracted from the SPE-ELF executable and is filled with zeros. Then spu-isolated-app encrypts the user-specified ELF segment (ALL, any one of the segments, or none) with AES-128/CBC mode. The reason why we provide application programmers with several options to encrypt ELF segments is to meet application requirements. Some applications do not need the code encryption, but they require the code verification, while some applications require both. During this process, an AES 128 key is required. Although it is possible for the user to supply an AES 128 key, the current spu-isolated-app automatically and randomly generates this key each time this tool is invoked. Note that spu-isolated-app does not allow weak keys or semi-weak keys in this process.

After the executable encryption, spu-isolated-app encrypts the previously used AES key with the SPE isolation loader’s encryption key. The spu-isolated-app tool supports RSA-1024, RSA-1536, and RSA-2048 in this AES key encryption. The Cell/B.E. Security SDK provides an RSA 2048-bit public key as the SPE isolation loader’s encryption key. Thus, spu-isolated-app will encrypt the AES key with RSA-2048. Now that the encryption of the executable and the AES key encryption are finished, spu-isolated-app formats the encrypted data by attaching the secure application header and the encryption section header.

Afterward, spu-isolated-app will generate the digital signature. It calculates hash values from offset 0 of the binary image to the end of the encrypted key section. We use the SHA-1 algorithm for this purpose. To generate a digital signature, spu-isolate-app encrypts the SHA-1 hash values with a user-specified RSA private key. Supported algorithms in the digital signature generation are RSA-1024, RSA-1536 and RSA-2048 as in the AES key encryption. Finally, spu-isolated-app produces the image on the right side of Figure 4 by attaching the signature section header, the digital signature and the user certificate associated with the RSA private key used in the digital signature generation. In the current implementation, the size of the SPE secure application is a multiple of 16 bytes, since the DMA engine in each SPE requires such a size.

### 3.4 Secure Application Loading

As explained in the previous section, spu-isolated-app builds an SPE secure application with an encrypt-then-sign polynomial.
icy. Compared with a sign-then-encrypt policy, this policy has a performance advantage. If the loader detects a verification error in the encrypt-then-sign case, then the loader does only the RSA and SHA-1 operations. In the sign-then-encrypt case, the loader should perform the RSA and SHA-1 operations after decrypting the application. Sometimes, the decryption process can be time-consuming. The cryptographic performance of the SPE isolation loader is discussed in Section 4.

However, [9] pointed out that encrypt-then-sign has a security flaw. In particular, it is possible to replace the signature if a malicious party has a legitimate signing key. This attack is the 'substituted-ciphertext attack'. The substituted-ciphertext attack is a fatal problem for SPE secure applications.

Let us assume that spu-isolated-app only does encrypt-then-sign to build a secure application:

1. key generation: $K_{A_{Alice}}$
2. application encryption: $K_{A_{Alice}}(S)$
3. key encryption: $(K_{A_{Alice}})_{loader}$
4. application hash calculation: $HASH(\text{APP})$
5. signature generation: $[HASH(\text{APP})]_{C_{Alice}}$

In Steps 1 and 2, spu-isolated-app generates an AES key, $K_{A_{Alice}}$, and encrypts the application secrets, $S$, with this key. In Step 3, it encrypts $K_{A_{Alice}}$ with the SPE isolation loader encryption key (an RSA public key). Then spu-isolated-app calculates the SHA-1 hash values, $HASH(\text{APP})$ for the application image, where $\text{APP}$ includes $K_{A_{Alice}}(S)$ and $(K_{A_{Alice}})_{loader}$, and then signs it with a private key associated with Alice’s digital certificate, $C_{Alice}$ in Step 5.

Figure 5 A(i)) shows the final output of spu-isolated-app. When the SPE secure application A is launched, the SPE isolation loader performs the validation and decryption operations in the reverse order of these build steps (Figure 5 A(ii)).

Now Mallory who has his legitimate certificate, $C_{Mallory}$ issued by the SPE isolation loader, can attempt a substituted-ciphertext attack against the SPE secure application A. He
imports $KA_{Alice}(S)$ and $KA_{Alice}$ loader into his application image, the SPE secure application $A'$ and signs this application with a private key associated with $C_{Mallory}$. Finally, Mallory gets the SPE secure application $A'$ containing his original code, $KA_{Alice}(S)$, $KA_{Alice}$ loader, $[HASH(APP)]_{C_{Mallory}}$ and $C_{Mallory}$ (Figure 5 B(ii)).

When this application is launched, the SPE isolation loader will begin to validate the integrity of $C_{Mallory}$ as in Figure 5 B(ii). Since $C_{Mallory}$ is a valid certificate, the loader is going to decrypt $[HASH(APP)]_{C_{Mallory}}$ with the public key in $C_{Mallory}$. After that, the loader validates the integrity of the SPE secure application $A'$ by comparing the expected hash values in the digital signature with $HASH(APP')$. This validation process also succeeds, because Mallory signed the SPE secure application $A'$ with his valid certificate. Thus, the SPE isolation loader goes to the next step to decrypt $KA_{Alice}$ loader, which is decipherable only by the SPE isolation loader. After the decryption of $KA_{Alice}$ loader, the loader decrypts $KA_{Alice}(S)$ with Alice’s key, $KA_{Alice}$ retrieved in the previous step. Then the loader will give the control to Mallory’s code, thus revealing Alice’s secrets to Mallory.

![Figure 5: Substituted-ciphertext attack against an SPE secure application](image)

To solve this problem, we need more steps for spu-isolated-app:

1. certificate hash calculation: $HASH(C_{Alice})$
2. key generation: $KA_{Alice}$ to encrypt/decrypt the application image
3. application encryption: $KA_{Alice}(S)$
4. session key generation (XOR): $KA_{Alice} \oplus HASH(C_{Alice})$
5. key encryption: $\{KA_{Alice} \oplus HASH(C_{Alice})\}$ loader
6. application hash calculation: $HASH(APP)$
7. signature generation: $[HASH(APP)]_{C_{Alice}}$

The most important part is coupling the hash values of Alice’s certificate and Alice’s application encryption/decryption key (in Step 1 and Step 4). In Step 1, spu-isolated-app calculates the hash value of $C_{Alice}$. Steps 2 and 3 in this new version are identical to Steps 1 and 2 in the previous vulnerable version. In Step 4, spu-isolated-app uses an XOR operation between $KA_{Alice}$ and $HASH(C_{Alice})$ to generate a session key, and encrypts this session key with the SPE isolation loader encryption key. In steps 6 and 7, spu-isolated-app generates a digital signature for the SPE secure application $A$, $[HASH(APP)]_{C_{Alice}}$. Finally, spu-isolated-app outputs the final image of the secure application $A$ in Figure 6 A(i).

When the application runs, the SPE isolation loader validates the integrity of $C_{Alice}$ and the integrity of the application image in the first stage. Then it decrypts $\{KA_{Alice} \oplus HASH(C_{Alice})\}$ loader with its private key associated with the SPE isolation loader encryption key to retrieve a session key $(KA_{Alice} \oplus HASH(C_{Alice}))$. Now the SPE isolation loader can extract $KA_{Alice}$ by calculating the hash value of $C_{Alice}$. Next the SPE isolation loader can successfully decrypt $KA_{Alice}(S)$ with $KA_{Alice}$ extracted at the previous stage (Figure 6 A(ii)).

With this algorithm, even if Mallory performs the substituted-ciphertext attack against this SPE secure application image, he can no longer read $S$. Since Mallory cannot generate $\{KA_{Alice} \oplus HASH(C_{Mallory})\}$ loader, he can only attach $\{KA_{Alice} \oplus HASH(C_{Mallory})\}$ loader to his application image. The possible application images he generates are shown in Figure 6 B(i) and Figure 6 C(i). In the case of Figure 6 B, the SPE isolation loader will detect the tampering, because the digital signature, $[HASH(APP')]_{C_{Mallory}}$ is not signed with $C_{Mallory}$. The loader stops the application execution at this point (6 B(ii)).

In Figure 6 C, both of the validation processes for $C_{Mallory}$ and $[HASH(APP')]_{C_{Mallory}}$ succeed as shown in Figure 6 C(ii). This is because $C_{Mallory}$ was issued by the SPE isolation loader and $HASH(APP')$ is signed with $C_{Mallory}$. However, in the decryption phase, although the SPE isolation loader can retrieve $KA_{Alice} \oplus HASH(C_{Alice})$ using the RSA decryption, it cannot extract $KA_{Alice}$ correctly. The SPE isolation loader performs $KA_{Alice} \oplus HASH(C_{Mallory}) \oplus HASH(C_{Mallory})$. Therefore, Alice’s secrets, $S$, are protected from the substituted-ciphertext attack. This method works properly as long as the loader’s private key for the RSA decryption or the private key associated with $C_{Alice}$ are not compromised. Note that our countermeasure has one restriction. A session key should not be weak or semi-weak, or Malory could easily retrieve $KA_{Alice}$, because he could find a session key and calculate $HASH(C_{Alice})$. Our build tool can avoid this by generating a new $KA_{Alice}$ with a session key that is not weak or semi-weak.

### 3.5 Staging in the SPE Isolation Loader

In contrast to staging implemented in a boot loader like GRUB [12], our staging method does not seek to expand the functions of the boot loader. Instead, our staging method compacts the SPE isolation loader and allocates large working areas for SPE secure applications.

Figure 7 depicts the staging technique implemented in the SPE isolation loader. There are three stages in our loader: stage0, stage1, and stage2. Unlike the boot loader, those three stages are concatenated into a single image. After this image is loaded into the isolated LS, stage0 relocates both stage1 and stage2 from the end of stage0 to the top of the reserved area to be in application stack space (Figure 7(iii)). Afterward, stage0 jumps to stage1, and then stage1 fetches an SPE secure application and overwrites the SPE isolation loader image initially loaded by the hardware, as Figure 7
Figure 6: XOR countermeasure against the substituted-ciphertext attack

illustrates. Stage1 validates and decrypts the SPE secure application as previously described. If the SPE secure application is not compromised, stage1 jumps to stage2. Stage2 is a cleanup process to clear the LS and the registers or other hardware resources that are used by the SPE isolation loader. In particular, the SPE isolation loader should never divulge its secrets such as an RSA private key to decrypt the application encryption key. We need to fill such areas with zeros. Stage2 finally allocates the application stack and jumps to the entry point of the SPE secure application. Note that stage2 has no secrets and it is erased as the application stack grows. We implemented both the SPE isolation loader compaction and contiguous working area allocation in this way.

Figure 7: Staging technique to compact the loader and allocate the contiguous working area

4. EXPERIMENT

We evaluated the performance of the application load process on a 2.8 GHz isolation-enabled IBM PowerXCell 8i processor [2]. Although this processor improves the performance of double-precision floating-point calculation at the hardware level in comparison with the Cell/B.E. processor, we did not use this new high-performance feature in this experiment. The instruction sets and hardware block diagrams are compatible with the Cell/B.E. processor. This experiment seeks to evaluate the feasibility and the efficiency of our SPE isolation loader implementation. In these experiments, we measured the performance of three types of applications: signed-only, signed/partially-encrypted, and signed/full-encrypted.

The application loading time of the SPE isolation loader was evaluated by varying the application executable size (text segment and data segment) from 20 KB to 80 KB. Figure 8 gives details of the time consumed by the SPE isolation loader during the application load. The values are average of 1000 times application load, and all standard deviations were less than 0.077% of the mean. The light gray bars are the load times of the signed-only SPE secure applications. The dark gray bars are the load times of the signed/partially-encrypted SPE secure applications. In this experiment, we encrypted 50% of the application executable in the partial-encryption. The black bars are the load times of the signed/full-encrypted SPE secure applications. For the signed-only application execution, it just takes less than 6.5 milliseconds to load the SPE secure application. This means that the SPE isolation loader with the encrypt-then-sign policy can detect tampering in 6.5 milliseconds. Users can re-allocate the isolated SPE core owned by this process to another process with minimal overhead comparing to the sign-then-encrypt policy. Note that the vulnerability in the encrypt-then-sign policy was fixed in Section 3.4. We have achieved both resource-efficiency and security in the implementation of the SPE isolation loader. Overall, the load overhead, about 50 milliseconds consumed by the SPE application fetch, RSA-2048/SHA-1 validation, and RSA-2048 and AES-128/CBC mode decryption processes is reasonable. It takes 500 milliseconds for the 2048-bit RSA sign process with the existing Trusted Platform Module (TPM) hardware [3].

According to the graph in Figure 8, the decryption process is much more expensive than the validation process. Comparing the load performance of the partially-encrypted applications to that of the full-encrypted application, there is not much of a performance difference. This indicates that the RSA decryption to retrieve an application decryption
key is a more time-consuming task than the AES decryption of the application image, since there is no relationship between the encrypted size and the load time. If we could gain more performance in the RSA decryption part, using the sign-then-encrypt policy would be feasible.

Figure 8: The performance of the SPE isolation loader on the 2.8 GHz IBM PowerXCell 8i processor

5. APPLICATION: CODE VERIFICATION SERVICE

Since not all applications can be ported to SPE executables, this paper also introduces a general solution. We have extended the code validation part of the SPE isolation loader and prototyped the code verification service as an SPE secure application. The code verification service validates a program residing on the system memory and returns the validation results to users. A 2048-bit RSA public key used to decrypt the digital signature of the target file is embed-

The streaming hash calculation enables the code verification service to obtain hash values for an application image that is larger than an isolated LS.

We prototyped this code verification service on top of the SPE isolation runtime. Since this service is based on the SPE isolation loader, RSA-2048 and SHA-1 are used for the validation process. Table 5 details the performance of our code verification service on the 2.8 GHz IBM PowerXCell processor. We chose several applications which are generally used on Linux systems. Note that shared libraries dynamically linked to executables are excluded in the footprint. As for the experiment of Firefox, we signed the primary executable (61 KB image size) which is directly invoked by a script, /usr/bin/firefox, but didn’t sign Firefox plugins. The time shown in the table is consumed from Step 3 to Step 8 in Figure 9. The values are averages of 1,000 verification requests and all standard deviations were less than 0.57% of the mean. In all applications, the time consumed by the validation failure case is longer than that consumed by the validation success case. This is because we optimized the code verification service for validation success by adding branch hints in the source.

In the current prototype, the verification service only returns the results of the file verification, but it is possible to implement other security features such as attestation and secure storage implemented in secure hardware like TPM. It is noteworthy that such services can be protected from tampering, behavior eavesdropping, and hijacking by the hardware isolation facility. Implementing such services in software provides benefits for users in terms of not only flexibility, but also security, because they are upgradable if the

and extract the expected hash values from the decrypted digital signature.

6. V: calculate the current hash values in a streaming manner: the target file is divided into several blocks and the code verification service calculates the hash value for each block. This operation is iterated until reaching the end of the file.

7. V: compare the current hash values and expected values

8. V — R: return the results

Figure 9: Steps of the code verification service
cryptographic algorithms or cryptographic keys are compromised.

6. RELATED WORK

In this section, we compare and clarify the differences between our SPE isolation loader and existing secure loaders.

6.1 Software Verifier

Tripwire [15] is a file validation program that can collaborate with the application loader at application load time. A user should register the hash values of the target applications into the database of Tripwire in advance. Tripwire compares the current hash values of the target applications and the values stored in its database as the system is running. If Tripwire detects a compromised application, it alerts the user that the system may be compromised. To protect Tripwire’s database, Tripwire encrypts its database with a passphrase specified by the user, which reduces the risk of using a bad database. However, when users install new security-sensitive applications or apply patches, they must update their database with new hash values. In contrast, the SPE isolation loader does not use a database to store the expected application hash values. The expected hash values are signed with keys and the signed values are attached to each application as digital signatures. The SPE isolation loader can validate the integrity of the SPE secure applications by using the digital signatures.

6.2 Secure Co-Processors

The Trusted Computing Group (TCG) defines the specification of a TPM chip [23], which is a secure co-processor. One of the security functions in TPM is direct anonymous attestation (DAA). Tripwire authenticates a client machine locally, while DAA is based on remote authentication. The reason is that expected hash values as well as installed programs in a machine can be compromised at the same time. Thus, the client machine to be verified only measures the hash values of the current system. The remote validation server compares the expected hash values with the hash values of the current status as sent from the client, and sends the validation results back to the client. The measured hash values are stored in Platform Configuration Registers (PCRs) that are only accessible from the TPM hardware. The TPM hardware signs the values in PCRs with its private key called the attestation key. On the server side, the validation program verifies the signed hash values and checks each hash value. In contrast, our approach is self-contained and there is no need for a validation server. Although application programmers should attach the certificate and the digital signature to the SPE secure applications, from the end-users’ point of view, the users do not need to set up any databases or remote validation servers to run their secure applications.

Similar to our work, Flicker [17] provides isolated execution for security-sensitive applications by using hardware security features, namely, late launch and attestation. When a user requests the late launch, the current execution environment is paused by a processor supporting the late launch. Intel and AMD offer such capabilities in their Trusted Execution Technology and Secure Virtual Machine extensions respectively. Then the late launch allows us to load a security-sensitive application at an arbitrary time with hardware protection against software-based attacks. It disables any access from other software including supervisory software to the memory space used by the security-sensitive application. During this load process, the processor causes the TPM to reset the specific PCRs and to store the hash values for the security-sensitive application into PCR 17. A trusted third party can validate the integrity of this application by verifying the signed hash values obtained from PCR 17. After the termination of the security-sensitive application, Flicker erases all of the secret data existing in the system memory and resumes the previous execution environment. The main difference between Flicker and our work is the application boot mechanism. Flicker supports the trusted boot, in contrast, we support the secure boot. Note that Flicker does not support encrypted application execution.

NGSCB [18] is also based on the TPM chip, and Dyad [26] and IBM 4758 [27] proposed original secure co-processors for the application validation and decryption. In those systems, the main processor and the secure co-processor collaborate with each other to provide a secure domain. The secure co-processors have secure local memory which is not accessible from any software or from other hardware. It is useful to store decryption keys or other secrets in such a secure memory. The advantage of co-processor solutions are easy setup of the secure domains by attaching co-processors and installing software stacks for them. Compared with the SPE isolation loader, the software stacks for IBM 4758 and NGSCB do not support encrypted application execution. Also, in our approach, the SPE isolation loader validates the integrity of a security-sensitive application every time when it is initiated, because we address attacks during the system runtime. Dyad, IBM 4758, and NGSCB perform the secure boot only at the boot time. In Dyad, since the applications are encrypted with the hardware root key, application developers must build new binaries for other machines with these different hardware keys to execute their applications on those machines. It is impractical to distribute applications tightly coupled with specific hardware. At the same time, our layered structure enables application developers to make their programs independent of hardware crypto-

<table>
<thead>
<tr>
<th>Application</th>
<th>Original Application Footprint (bytes)</th>
<th>Signed Application Footprint (bytes)</th>
<th>Validation Time (milliseconds)</th>
<th>Validation Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>la</td>
<td>127,736</td>
<td>128,192</td>
<td>1.4766</td>
<td>2.7367</td>
</tr>
<tr>
<td>ssh</td>
<td>404,932</td>
<td>405,376</td>
<td>2.513</td>
<td>6.5108</td>
</tr>
<tr>
<td>xterm</td>
<td>373,996</td>
<td>374,464</td>
<td>2.3974</td>
<td>6.09</td>
</tr>
<tr>
<td>Xorg</td>
<td>2,062,188</td>
<td>2,062,656</td>
<td>8.7097</td>
<td>29.074</td>
</tr>
<tr>
<td>firefox</td>
<td>62,328</td>
<td>62,784</td>
<td>1.232</td>
<td>1.846</td>
</tr>
</tbody>
</table>

Table 1: The performance of the code verification service for each application.
graphic algorithms, although they are dependent on the SPE isolation loader’s cryptographic algorithms. The software authentication layer can hide the hardware changes, which provides portability and flexibility for their applications.

6.3 Secure Main-processors

In XOM [16] and Cerium [5], secure and trusted software validates the integrity of each application whenever data is transferred to a CPU. When a CPU needs to send any computed results back to the main memory, the secure and trusted software encrypts them prior to the data transfers. Both XOM and Cerium assume that secure and trusted software has the highest privilege level, since neither system trusts the commodity operating systems. To protect secure and trusted software against tampering (e.g., a DMA attack [19] that can directly attack the supervisory software), the whole image of that program must stay in the cache or in local memory in the CPU, as in our approach. However, [20] pointed out that XOM has a security flaw. It is possible for hackers to use replay attacks against the encrypted data and instructions from a XOM processor.

AEGIS [22] is a system to realize secure application loading based on a secure boot. It also supports encrypted application execution similar to the SPE isolation loader. In contrast to our approach, AEGIS does not utilize a runtime secure boot. Instead, they embed hash values of a runtime system, in particular for a secure small kernel into an application image at build time. When an application is launched, secure and trusted software computes the current hash values of the running hypervisor, kernel and the target application. It subsequently validates the target application by comparing the embedded hash values with the current ones. However, in this approach, programmers must rebuild their applications every time security patches for the operating systems or hypervisors are released, because the hash values of those programs will be changed. In contrast, we establish a chain-of-trust with the runtime secure boot with the layered structure. In our approach, the hash values of the underlying software are not required for the application build process.

In XOM, Cerium, and AEGIS, when secrets are transferred from the caches of a secure CPU to the main memory, secure and trusted software or hardware automatically encrypts them with their hard-coded cryptographic algorithms. Although application programmers are freed from taking care of the secure secrets handling, it is difficult to change the encryption algorithms according to application requests. In contrast, we allow application developers to program cryptographic algorithms protecting the secrets to be transferred from the isolated LS to the main memory. This provides flexibility in programming secure applications according to users’ requirements. Our software stack also provides a high-performance cryptographic user library, the SPE cryptographic library [1]. Application developers can easily program secure DMA transfer functions.

TrustZone [25] is also categorized in this group, but this technology is different from these solutions. In TrustZone, a new privilege level is added into an ARM processor, which realizes both a secure domain and a non-secure domain in one machine. A secure kernel, secure device drivers, and secure applications are running in the secure domain, while a traditional kernel, drivers and applications are running in the non-secure domain. It is impossible to access the secure domain resources from the non-secure domain, because TrustZone partitions both domains at the hardware level. In the secure domain, a secure boot is performed from the kernel layer to the application layer. Note that the secure kernel that has the role of loading the secure applications is validated only at the system boot time. The SPE isolation loader that has the same role is also validated by the hardware every time when an SPE secure application is launched. The main difference between our work and TrustZone is that the isolation runtime modules isolate the secure application domain from the supervisory software domain. No supervisory software is running in the same memory space in the secure application domain. When third-parties secure drivers or secure applications are concurrently running in the TrustZone secure domain, those programs might try to observe the behavior of a security-sensitive application or it might be possible for hackers to steal secret data by attacking security flaws in these third parties’ programs. Our approach reduces these security risks.

7. CONCLUSION AND FUTURE WORK

We have introduced the detailed design and implementation of an SPE isolation loader, which is a part of the IBM SDK for Multicore Acceleration. The SPE isolation loader makes security practical, extensible, and portable by exploiting the Cell/B.E. security features. The coupling technique for the digital certificate and the decryption key introduced in this paper solved the substituted-ciphertext attack that can reveal secrets to others having legitimate signing keys.

In the performance measurements, it took only 50 milliseconds for the SPE secure application load process, including RSA-2048 decryption and AES-128 decryption processes, even if the SPE secure application is large. Note that this overhead only affects the SPE secure application load. Once an SPE secure application obtains control, the application can utilize all of the hardware resources, such as the registers, LS, and channels, to obtain high performance from the SPE. Our evaluation also showed that we could implement the SPE isolation loader efficiently and securely. If we replace the RSA-2048 decryption part with a better cryptographic algorithm such as Elliptic Curve Cryptography, it would be possible to apply a sign-then-encrypt policy to our loader. The design and implementation of a sign-then-encrypt policy is future work.

In addition, we have developed a code verification service in Section 5 by extending the SPE isolation loader’s validation feature. This example demonstrated that it would be possible to realize a virtual TPM or a virtual IBM 4758 co-processor in software. Although we only showed the validation implementation in this paper, our service can also be extended as a cryptographic accelerator by integrating it with AES and RSA encryption/decryption in the SPU cryptographic library. For instance, a software-based secure cryptographic accelerator would encrypt all outgoing traffic from a cluster of machines or decrypt all incoming traffic for the cluster. The cryptographic keys never leave the secure vault with our framework. We will investigate such applications in the future.

8. ACKNOWLEDGMENTS

The authors would like to thank the management team, development team, the researchers, and the architects for...
their support and guidance. In particular, we would like to thank Mike Day, Peter Hofstee, Dan Brokenshire, Jeremy Kerr, Mark Nutter, Arnd Bergmann, Mike Kistler, Sally Tekulsky, Mike Perks, and Vladimir Zharsky. This work would not have been possible without their support.

9. REFERENCES


- Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.
- IBM, POWER, Power Architecture, PowerPC, PowerXCell are trademarks of IBM Corporation in the United States, other countries, or both.
- Other company, product, or service names may be trademarks or service marks of others. Other company, product, or service names may be trademarks or service marks of others.