



# The Cell Broadband Engine processor security architecture

## Hardware security solutions

Level: Intermediate

[Kanna Shimizu](mailto:kannas@us.ibm.com) ([kannas@us.ibm.com](mailto:kannas@us.ibm.com)), Security Architect, IBM Systems and Technology Group

17 Apr 2006

The unrelenting evolution toward an even more open and connected computing infrastructure requires robust security to thrive. Learn how the Cell Broadband Engine™ processor's security architecture is uniquely suited for the challenges of this digital future.

As computers and consumer electronics devices become more connected, platform security becomes increasingly important for everyone from consumers to businesses. For consumers, privacy of data such as credit card numbers and social security numbers have always been of concern, but now new technologies such as voice-over-IP and personal video blogs bring new privacy concerns. And for entertainment content owners, piracy is a major concern as they move toward a virtual form of TV and movie content delivery (see [Resources](#)).

Within this context, the Cell Broadband Engine (Cell BE) (see [Resources](#) for more references) offers a processor security architecture that provides a robust foundation for the platform. Until now, because most processor architectures did not provide any security features, security architects relied on software-implemented approaches to provide protection. However, protecting software with software has a fundamental flaw in that the software with the protector role can be compromised as well. Therefore, processor hardware, which is intrinsically less vulnerable than software, needs to be re-thought and re-architected to support the security of the platform. The Cell BE architecture is designed with this goal in mind. Because its designers were given the rare opportunity to design a processor from the ground-up, security is an integral part of the processor architecture and not an after-thought.

With the confusing array of security solutions available in the marketplace, it is helpful to clarify what attack model a design is intended to protect against. Although, the Cell BE processor does have defenses against physical attacks, the architecture's main focus is software-based attacks. These attacks can be unleashed simply by executing software code, and often times, the code is available for free from an Internet Web site. In contrast, physical attacks require obtaining extra hardware (such as a mod-chip), or expensive measuring equipment and also require skill in opening up the system to make the necessary changes. Because it is much easier for an individual to copy a software-based attack than a physical attack, a software-based attack will clearly become more widespread and hence more devastating when it is discovered.

Furthermore, unlike physical attacks which require physical proximity to the target, software attacks can also be unleashed against a user by an external attacker through the platform's connectivity. Therefore, with software-based attacks, the user can be either the adversary (where the user's software manipulation leads to pirating of content), or the victim (where a virus exposes the user's private data). By providing features that can help thwart both classes of software-based attacks, the Cell BE security design provides a valuable solution to an increasingly challenging problem.

The architecture's main strength is its ability to allow an application to protect itself using the hardware security features instead of the conventional method of solely relying on the operating system or other supervisory software for protection. Therefore, if the operating system is compromised by an attack, the hardware security features can still protect the application and its valuable data. As an analogy, consider the protection the supervisory software provides as the castle's moat and the Cell BE security hardware features as the locked safe inside the castle. To achieve this, the Cell BE security architecture offers three core features. By using the



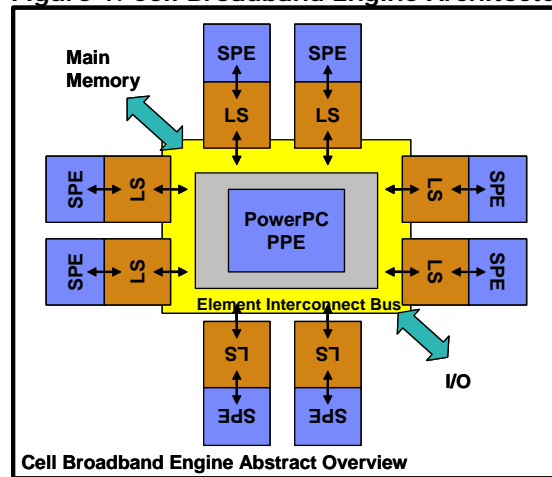
hardware *Secure Processing Vault* feature, an application can execute isolated from the rest of the software environment. With the *Runtime Secure Boot* feature, an application can run a check on itself before it is executed to verify that it has not be modified and compromised. If a modification in the application is detected, the application execution can be stopped. Secure Boot is normally done only at power-on time, but the Cell BE processor can Secure Boot an application thread multiple times during runtime. And finally, with the *Hardware Root of Secrecy* feature, unlocking of secrets such as various keys can be protected using the robustness of hardware. A fourth feature, a hardware random number generator, is also relevant to some security issues. The following section gives a brief overview of the Cell BE architecture so you can better see the framework in which these features are provided.

## Architectural overview of the Cell Broadband Engine processor

The Cell BE processor is a multiprocessor core design with nine processor cores. The principal core, the 64-bit Power Processor Element (PPE), is a PowerPC® processor assuming a supervisory role. The eight Synergistic Processor Element (SPE) cores are the computational workhorses: they are well suited for compute-intensive graphics and video calculations.

The SPE plays a key role in the Cell BE security architecture. One of its distinguishing features is its 256Kbytes of physically dedicated, on-chip private memory called the Local Store (LS). Before an application is started on the SPE, its code and data are placed in the LS, and once the transfer is complete, the SPE is kick-started, and starts fetching instructions and data directly from the LS using an LS address. The SPE has a Direct Memory Access (DMA) engine which transfers data into the LS from other resources in the system such as main memory, I/O devices, and LS of other SPEs. Therefore, on one side, the LS is read from and written to by the SPE processor, and on the other side the LS receives and services reads and write requests from other agents on the bus.

Figure 1. Cell Broadband Engine Architecture Overview



## The security features of the Cell BE processor

The following is a summarized list of the key security features of the Cell BE architecture:

- Secure processing vault
- Runtime secure boot
- Hardware root of secrecy

## The Secure Processing Vault

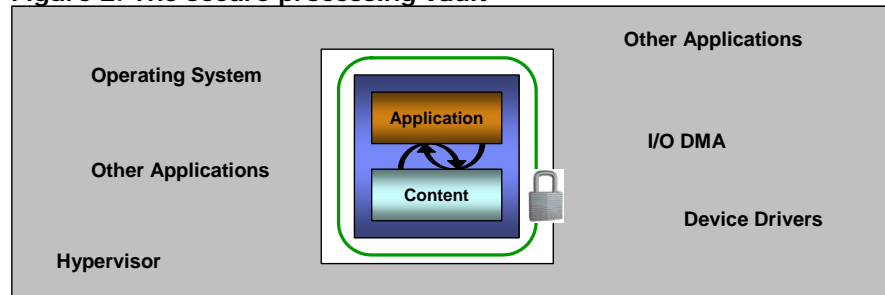
Goal: isolation of an application

To achieve a secure platform, a processing environment must exist where a single application can execute isolated from all other executing software threads in the system. The Cell BE processor's



Vault can provide such an environment. Within the vault, the execution of the application and its data cannot be manipulated or observed -- the hardware design prevents other applications from doing so. For example, digital movie content can be decrypted in, and played from, the vault without the danger of the content being compromised.

**Figure 2. The secure processing vault**

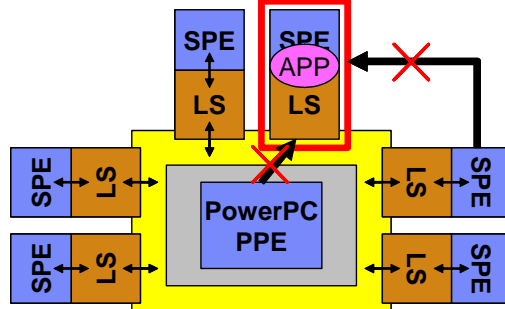


The goal of isolating a process thread is not new; however, in contrast to the hardware-based method, existing approaches have used software to enforce the separation. The operating system or the hypervisor (also known as the virtual machine monitor -- the layer of software with the most authority in a virtualized system) has the responsibility of separating processes. For example, the operating system would ensure that the memory location of the high-value digital content is protected from reads and writes from non-authorized processes. The problem with this approach is that if an adversary takes control of the operating system or the hypervisor, all bets are off. The adversary can use the operating system to change the permissions for the memory area it is trying to break into, or simply use the operating system to read the memory location since the operating system can read any memory location in most systems. In fact, this is why the operating system (or root) is usually the target for hackers and viruses. An adversary will look for a weakness in the operating system design, such as a buffer overflow vulnerability (see [Resources](#)), exploit this hole to gain control of it, and then execute operations that only the operating system has privileges to do. Within this kind of environment, sensitive data can be easily copied by the adversary-controlled operating system because the memory protection for that data no longer has any effect. The same argument would hold for a virtualized machine where a hypervisor controls the memory accesses of different processes. If the hypervisor is compromised, whatever protection mechanisms it is intended to provide will not matter anymore. The fundamental problem with existing approaches is that they rely on software to provide the isolation, but at the same time software can be manipulated by an adversary. A better approach is for the hardware design to isolate the process in such a way that the software cannot override the isolation, and this is precisely what the Cell BE processor's Vault provides.

The Vault is implemented as an SPE running in a special mode where it has effectively disengaged itself from the bus, and by extension, the rest of the system. When in this mode, the SPE's LS, which contains the application's code and data, is locked up for the SPE's use only and cannot be read or written to by any other software. Control mechanisms which are usually available for supervisory processes to administrate over the SPE are disabled. In fact, once the SPE is isolated, the only external action possible is to cancel its task, whereby all information in the LS and SPE is erased before external access is re-enabled. From the hardware perspective, when an SPE is in this isolation mode, the SPE processor's access to the LS remains the same, while on the other side of the LS (the bus side), external accesses are blocked. Thus, all LS read and write requests originating from units on the bus such as the PPE, other SPEs, and the I/O have no effect on the locked-up region of the LS. However, an area of the isolated SPE's LS is left open to data transfers to and from other units on the bus for communication purposes. The application running on the isolated SPE is responsible for ensuring that the data coming through the open communication area of its LS is safe. Also, consistent with the idea that the cores execute independently, any number of SPEs can be in isolation mode at any given time.



Figure 3. The application inside an isolated SPE cannot be observed or modified



All of this is accomplished exclusively by hardware means; no software, in the form of setting protection bits in an address translation table for example, is involved in the process. Because of this hardware isolation, even the operating system and the hypervisor cannot access the locked up LS or take control of the SPE core. Therefore, a hacker who has gained root or hypervisor privileges is not a threat to an application executing on an isolated SPE. The supervisory privileges will not enable him to control the application, nor will it allow him to read or write the memory used by it. The execution flow and the data of the isolated application are safe. A hotel analogy clarifies this security model; the hotel manager (PPE) allocates a room (SPE) for a guest (application). The guest can lock the room from the inside; the hotel manager, and other guests, cannot peek into the room. However, the hotel manager can kick the guest out.

### Runtime secure boot

Goal: verification of an application

The Vault protects an application from other software which might have been modified or compromised. However, that still leaves open the question of what happens if the application itself has been modified. For example, an adversary can modify the application so that when the application accesses valuable data within the Secure Vault, it copies the data to outside of the Vault into an openly accessible area. Such a modification needs to be detected so that the application is not executed. One counter-measure might be to design a software-implemented loader which does an authentication check on the application and only executes it when the authentication succeeds. However, the loader might be modified so that it does not check for authentication correctly and allows compromised code to execute within the Vault. Or, an adversary might circumvent the loader entirely, and the authentication step is skipped. A hardware solution is needed so that the authentication step is consistently and correctly executed.

Due to the malleability of software, it is generally believed that the root of an authentication scheme must be implemented in hardware. If the root can be trusted, then the entity authenticated by the root can be trusted, and so on as the chain of trust expands. Based on this philosophy, Secure Boot is a technique whereby during power-on time, from the first BIOS code that is executed to the operating system code, the code modules go through a cryptographic-based authentication check. Although there are a variety of flavors of this, one way is to have a small boot module be authenticated by the hardware using a hardware key (the hardware root of trust), and then this module is now entrusted to authenticate the operating system. If the authentication of the boot module or the operating system fails, the booting process is halted, but otherwise, the booting process is allowed to happen normally. The idea is that since the first software to execute on the chip was authenticated by the hardware, and all succeeding software code is verified by the code that launched it, the chain of authentication ensures that all software on the system is indirectly or directly verified by the hardware root of trust at power-on time.

The drawback of this approach is it assumes that checking for compromises in the software at power-on time is enough. It does not protect against software compromises that happen after power-on time. However, most software-based attacks happen during runtime, and if this happens, the chain of authentication breaks, and any software that is launched after that time can not necessarily be trusted.

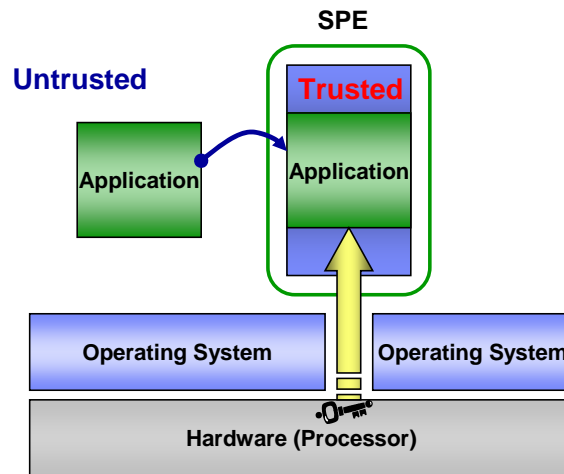
The Cell BE processor addresses this problem with its Runtime Secure Boot feature. It lets an application secure boot from the hardware an arbitrary number of times during runtime. Thus,



even if other software in the system has been compromised in the past, a single application thread can still be robustly checked independently. In essence, the application can renew its trustworthiness as many times as needed even as the system stays running longer and gets more stale. Specifically, a hardware implemented authentication mechanism uses a hardware key to verify that the application has not been modified, and the authentication is based on a cryptographic algorithm.

This runtime secure boot, in fact, is tightly coupled with an SPE entering isolation mode. An application must go through the hardware authentication step before it can execute on an isolated SPE. When isolation mode is requested, first, the previous thread is stopped and cancelled. Then, the hardware will automatically start fetching the application into the LS, and the hardware will verify the integrity of the application. If the integrity check fails, the application will not be executed. The check can fail for one of two reasons. The application might have been modified within memory or storage. Then, the assumption is that the functionality might have changed and it cannot be trusted anymore. Or, the writer of the application does not know the cryptographic secret that is needed for a successful authentication. Otherwise, if the authentication check is successful, the hardware will automatically kick-start the application's execution in isolation mode. Because the hardware controls all of these steps, the verification of the application's integrity cannot be skipped or manipulated and will happen consistently and correctly.

**Figure 4. The runtime secure boot feature**



1. Isolation mode is initiated.
2. SPE processor state is cleared, initialized.
3. Application is fetched in and checked by the hardware authenticator
  - based on a hardware key and cryptographic algorithm
4. Integrity check fails; execution stopped
  - Application was tampered
5. Check succeeds; will kick-start the application's execution in isolation mode.

## Hardware Root of Secrecy

One of the most important aspects of system security is how keys are managed. Keys are the linchpin for system security and data protection. Applications use them to encrypt data in system memory, to decrypt movies, or to establish a secure communication channel. If the keys can be easily exposed, the entire security scheme falls apart.

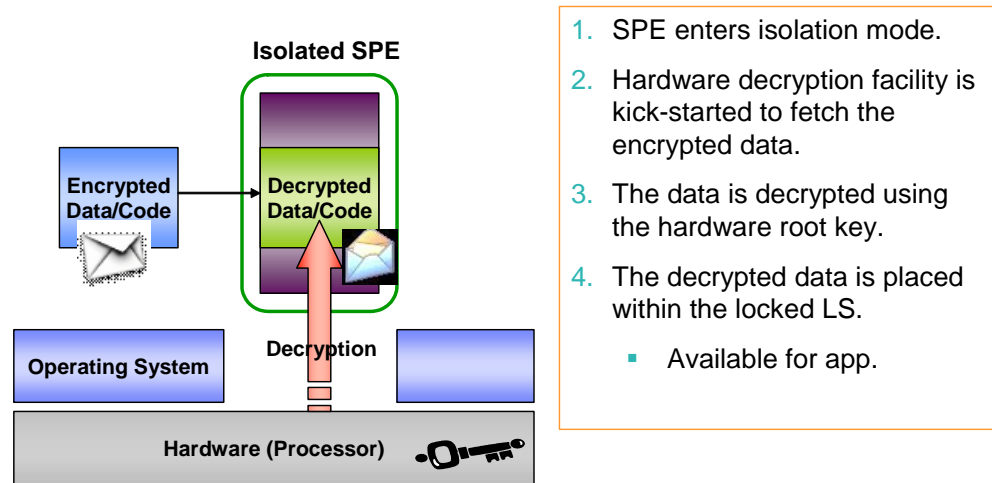
Despite their critical role, keys are usually stored in plain text form in storage. Ideally, instead of in this naked state, the keys will be sealed in an envelope (in other words, encrypted) when in storage, and only unsealed when given to an application that has been authenticated. However, this implies that another key is used for the sealing and unsealing (in other words, for encrypting and decrypting the first key); how is this key stored? Eventually, there must be a key that is not encrypted, and because this is the key that is at the root of all unsealings, we will refer to it as the *root key*.

Because of the root key's importance in keeping all other keys hidden, it must be robustly protected. The Cell BE processor accomplishes this with its Hardware Root of Secrecy. The root key is embedded in the hardware, and you cannot access it with software means; only a hardware



decryption facility has access to it. This makes it much more difficult for software to be somehow manipulated so that the root key is exposed, and of course, the hardware functionality cannot be changed so that the key is exposed.

**Figure 5. Hardware Root of Secrecy**



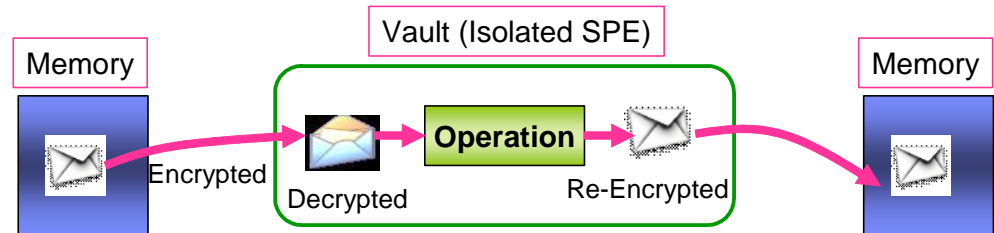
Furthermore, the activation of the hardware decryption using this root key is tightly integrated with the SPE isolation mode. When an SPE enters isolation mode, the hardware decryption facility is kick-started to fetch the encrypted data into the isolated SPE and decrypt the data using the hardware root key. The decrypted data is placed within the protected Local Store and is available for an isolated SPE application to use. In fact, the decryption based on the root key can only happen within an isolated SPE and not outside of it; no access to the root key is available, by hardware or software means, from a non-isolated SPE or the PPE. First, this implies that a system designer can force all data decryptions by the root key to happen within the protected environment of the Secure Processing Vault; the keys unsealed by the root key will always be placed (at least initially) in the Vault only. Second, only applications that have successfully passed the Runtime Secure Boot authentication are given access to the keys unsealed by the root key. Any software that might have been adversely modified will not be given access to the unsealed keys. Because the foundation of this control is grounded in both the Runtime Secure Boot and Hardware Root of Secrecy features, the process is more resistant to manipulation than with a pure software-controlled access mechanism.

Another advantage of this feature answers the question, what prevents an adversary from taking an application intended to run within the Vault and executing it outside of the Vault? The answer is to encrypt a portion of the application code using the hardware root key. Because the code is encrypted, it cannot be captured and directly executed on a regular, non-isolated SPE. The code needs to be decrypted, and therefore has to execute within the Vault where it can be decrypted by the root key. This reassures the application writer that a particular application will only execute within a Secure Processing Vault.



## Usage models

Figure 6. Encrypt-in, Encrypt-out usage model



➔ Data Flow (example: content video data, passwords, credit card numbers, social security numbers, any data with piracy or privacy concerns)

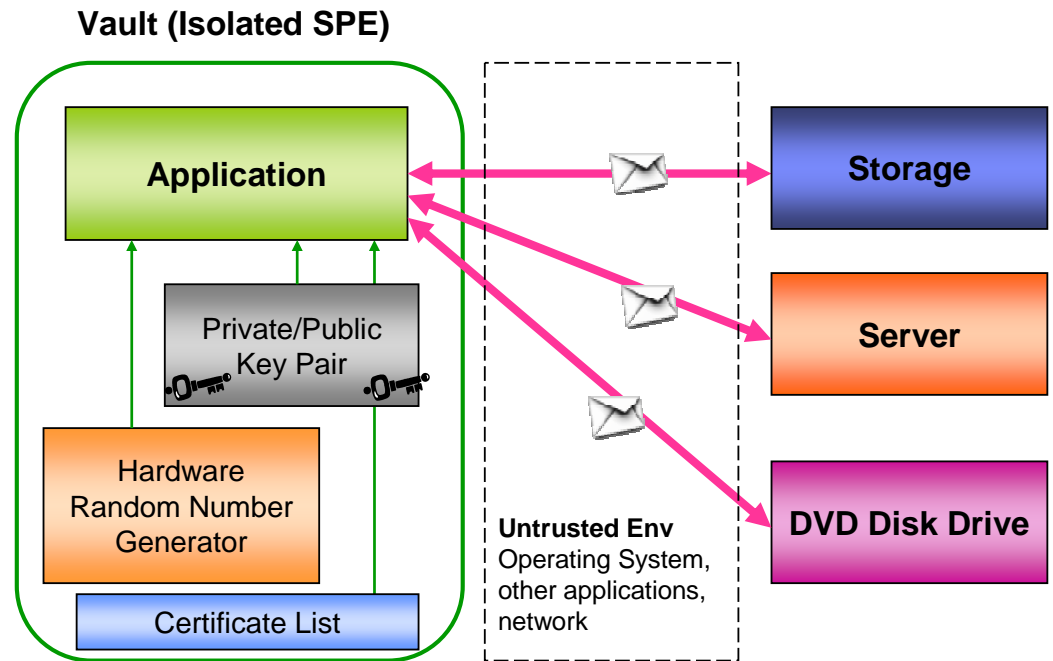
- The data is in its plaintext form only within the Vault, never in main memory.
- The only code that has access to the plaintext data is authenticated by hw.
- The keys used for encrypting/decrypting are hidden from the system.

The Secure Processing Vault is best exploited by the Encrypt-in, Encrypt-out usage model where the incoming data is decrypted, an operation is done on the data, and the data is re-encrypted before it is placed outside of the Vault. In this model, the data is in its vulnerable, plain text form only within the Secure Processing Vault; the only code that has access to this plain text data is authenticated through the Runtime Secure Boot; and the keys used for decryption and encryption are hidden from the system using the Hardware Root of Secrecy. With this usage model of the Vault, existing system functions such as file operations and network operations can be used as is without sacrificing on security. Because the data is already encrypted by the time it is accessed as a payload to these operations, even if these system operations are somehow compromised, the secrecy and authenticity of the data can be ensured.

These system functions that are outside of the Vault are treated as part of the untrusted environment, and traditional cryptographic-based methods are used for the Vaulted application to securely communicate to itself at a later time (for storage functions), to a server on the network (for network functions), or to another device in the system (for I/O functions). In addition to the usual use of a public and private key pair (see [Resources](#)) and a certificate revocation list (a list of identities that have been revoked and should not be trusted), the secure, authenticated communication is achieved by the three core security features and also an on-chip hardware random number generator. For example, the Vault feature can ensure that the process of authenticating its communication partner is not manipulated by an adversary. The runtime secure boot can protect the certificate revocation list (think of this as the list of bad guys) from modification (a particular bad guy might be removed from the list, for example). The hardware root of secrecy can ensure that the private key is not exposed by an attacker. (If the private key is obtained, all communication addressed to the private key owner will be exposed to the attacker). The hardware random number generator protects against replay attacks (see [Resources](#)) by marking the current communication with a time stamp. A replay attack is where an adversary takes an old communication message and sends it again through the unsecured communication channel. Because the authentication protocol will verify that the message is authentic, a robust time stamping feature is the only way for the communication partners to realize that there is a man-in-the-middle attack happening.



Figure 7. Secure Authenticated Communication over an insecure channel



## Resources

- [Participate in the discussion forum.](#)
- The article [Smashing the Stack for Fun and Profit](#) explores some of the ways that buffer attacks work.
- Wikipedia has good articles on [replay attacks](#) and [public-key cryptography](#).
- **References**
  - The April 10, 2006 *Wall Street Journal* discussed Disney's plans to offer TV shows on the web, on page A1.
  - The *IBM Journal of Research and Development*, Volume 49, No. 4/5, discussed the Cell architecture in substantial detail, on pages 589-604.
  - The March 11, 2006 Economist article *Hackers go home* appeared on page 54, and discussed issues relating to content protection.
  - *Advances in Cryptology*, 2001, printed the paper *Revocation and Tracing Schemes for Stateless Receivers*, pages 41-62.

Kanna Shimizu is the Security Architect for the Cell Broadband Engine processor and Next Generation Computing Systems at IBM. She holds a B.S. in Electrical Engineering from California Institute of Technology, a M.Sc. in Computer Science from University of Oxford, and a Ph.D. in Electrical Engineering from Stanford University. Her interests include secure computing, content protection, formal methods, and financial mathematics.