Java Cryptography on KVM and its Performance and Security Optimization using HW/SW Co-design Techniques

Yusuke Matsuoka, Patrick Schaumont, Kris Tiri, and Ingrid Verbauwhede

Electrical Engineering Department
University of California, Los Angeles
Los Angeles, CA 90095

{yusuke, schaum, tiri, ingrid}@ee.ucla.edu

ABSTRACT

This paper describes a design approach to include and optimize Java based cryptographic applications into resource limited embedded devices.

For easy prototyping and to be platform independent, the security applications are first developed in Java. Two Java cryptographic libraries, the Bouncy Castle API and the IAIK API are ported to a real embedded device for cost and performance evaluation. It requires 0.88Mbytes to 1.2Mbytes in the KVM footprint size and a few milliseconds to run secret key algorithms and message digests on a typical embedded device.

In a second step, the performance critical components of the security applications are moved to hardware acceleration units. The GEZEL design environment is used for the hardware modeling and the co-simulation between software on KVM and the hardware co-processor. Moving the AES algorithm from the SH3-DSP microprocessor to a hardware co-processor shows a performance gain of 10.4x including the overhead in Java, C, and hardware interfaces.

Then in a third step, the security critical components are realized by means of a special dynamic differential logic (DDL) style, which makes the secure modules resistant against side channel attacks. All key related actions and cryptographic algorithms are restricted to the secure co-processor. The overall performance gain is 25x compared to a pure Java implementation.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems - *Real-time and embedded systems*; D.3.2 [**Programming Languages**]: Language Classifications - *Object-oriented languages*, *Java*

General Terms

Performance, Design, Security, Languages

Keywords

Java, Cryptography, Security, Embedded Systems, Design

1. INTRODUCTION AND MOTIVATION

The worldwide market of mobile communication is growing at a rapid pace and has overtaken wired phone communications. The applications for mobile devices become more complex and include new features and services over the network, such as online banking, e-commerce, user and server authentication, and so on. At the same time, the consumer expects longer lasting battery times, operating and standby times. It is clear that these mobile devices require low power embedded security. To provide secure communication channels, the mobile devices need to be capable of running cryptographic algorithms. The cryptographic algorithms use different types of keys to encrypt/decrypt the data, such as secret keys, public keys, session keys, and so on. The keys also have to be secure against eavesdropping and leaking. In contrast to a desktop computing environment such as an authentication server deployed in the backend of a network infrastructure, off-the-shelf types of devices are more vulnerable to the threats of eavesdropping [1][2]. Most systems do include some measures against tampering. But even with tamper-proof devices, there is the risk of leaking information through side-channels.

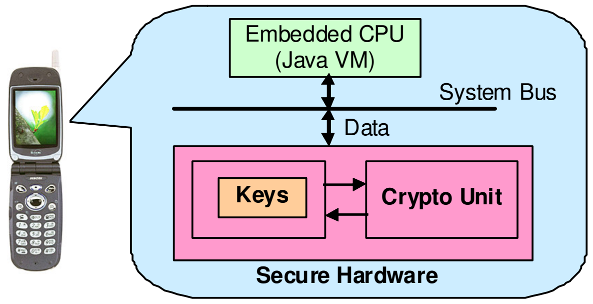


Figure 1. Architecture of Secure Embedded Device.

One solution toward these types of security threats is to deploy secure hardware next to the CPU in the system. Figure 1 depicts the concept in which the cryptographic algorithm is implemented in a separate secure hardware unit with the key storage. It has the advantage that the access, the calculation and the protection is limited to a well confined area. Adding security measures has a cost in terms of area and power and by limiting it to a smaller section of the SOC, there is an overall performance gain without a security loss.

In addition to the confidentiality of the keys and the cryptographic operations, performance of the cryptographic operations is also a crucial aspect for mobile devices in spite of its constrained computing resources. Various cryptographic applications need to be executed with feasible cost and performance. Ideally, these applications should be platform independent – i.e. the application does not need to be modified for each platform and/or for the performance and security optimization.

To satisfy these security, cost/performance, and consistency requirements, we propose a design framework for secure embedded systems for future mobile devices. The proposed method starts from a specification written in Java and proposes a gradual and systematic refinement to an implementation that consists of Java with one or more crypto co-processor acceleration units, implemented in a secure digital standard cell design technology.

We chose the KVM (K Virtual Machine) as the design platform. KVM is the implementation-level foundation for the J2ME (Java 2 Micro Edition) [3] and it has been implemented on many cellular phones and mobile communication devices all over the world [5]. However, since the security protocols are not implemented on KVM [4], most of the cellular phones only have a minimum set of security protocols such as SSL/TLS [6]. In spite of the absence of standardized methods, future mobile devices need to be capable of running various cryptographic algorithms with feasible cost and performance in a platform-independent manner. We used lightweight Java cryptographic libraries (API) on the J2ME platform to provide a fast prototyping and platform independent environment. The cost and performance of the API is evaluated in a real embedded device.

For the performance optimization, we use the GEZEL design environment which allows us to move computational intensives modules to dedicated co-processors. To avoid bugs and potential security weaknesses, the GEZEL design environment allows co-simulation of the code running on the KVM and the cryptographic co-processors in a cycle-true manner. In the third step, the secure hardware module is made side-channel resistant by implementing it in a secure digital design style.

The remainder of this paper is organized as follows. In section 2, we first evaluate the cost and performance of Java cryptographic libraries on a real embedded device. In section 3, we present a design flow for the performance optimization and introduce the design method of hardware acceleration. The performance gain is evaluated by the hardware/software co-simulation techniques. Section 4 provides a security optimization on key management of the proposed design framework against the physical attacks. Conclusions are provided in section 5.

2. COST AND PERFORMANCE OF JAVA CRYPTOGRAPHIC LIBRARIES

2.1 Cryptographic Extensions on J2ME

To evaluate the cost and performance of Java cryptographic libraries, we have chosen two sets of cryptographic libraries: the Bouncy Castle lightweight API [7] and the IAIK JCE/iSaSiLk Micro Edition API [8] as the implementation targets because of their suitability for the KVM platform and the source code availability. These libraries are built into the KVM executable at compile time and used by the user application as the Java API. The cryptographic extension of KVM is performed based on the J2ME CLDC 1.1 RI (Reference Implementation) [4].

Table 1. Cost of Cryptographic Extensions.

	CLDC 1.1 RI Original	KVM with Bouncy Castle API	KVM with IAIK API
Total # of class files	102	508 (original + 406)	169 (original + 67)
Total size of class files	91KB	589KB	286KB
KVM size	277KB	829KB	518KB

Table 1 shows the comparison of the cryptographic extensions performed on Cygwin. Note that the total size of the class files is the size of the zipped jar file. The Bouncy Castle Lightweight API consists of 406 class files, and the KVM footprint is 829Kbytes. On the other hand, the IAIK API consists of 67 class files, and the KVM footprint is 518Kbytes. Comparing to the original KVM footprint, the overhead to accommodate the Bouncy Castle API and the IAIK API is 552Kbytes and 241Kbytes, respectively. Consequently, the IAIK API requires a smaller KVM footprint, whereas the Bouncy Castle APIs supports a larger number of cryptographic algorithms.

2.2 Performance Evaluation on Embedded Device

2.2.1 *Experimental Setup*

The Intel StrongARM SA-1110 Development Board is used for the experiments and performance evaluation of the cryptographic extensions. The SA-1110 processor is a 32-bit RISC processor that can run up to 206 MHz, optimized for portable and embedded applications [9]. The SA-1110 processor has a 16Kbyte instruction cache and an 8Kbyte data cache, a memory-management unit (MMU), and read/write buffers. The board has a 64MB SDRAM with 100MHz interface bus, and a 16MB flash memory for the programming.

To provide an integrated tool chain and a debug environment for porting the KVM, the eCos OS is used. eCos is an open source, configurable, portable, and royalty-free embedded real-time operating system supported by the GNU open source development tools [10]. In the configuration tool integrated in the eCos package, we used a design template for the SA-1110 Development Board to obtain the libraries for the StrongARM processor.

By using the arm-elf-gcc cross-compiler and the library from eCos, the two KVMs with the Bouncy Castle API and the IAIK API have been built on Cygwin. The size of the KVM footprint was 1.2Mbytes for the Bouncy Castle APIs, and 0.88Mbytes for the IAIK APIs, respectively.

2.2.2 *Experimental Results*

To test the functionality and to evaluate the performance, popular cryptographic algorithms are implemented in Java. We have evaluated the AES, DES, RC2, RC4, RC5, and RC6 algorithms for symmetric encryption and decryption, and the MD2, MD4, MD5, SHA-1, SHA-1(256bit), and SHA-1(512bit) algorithms for message digest. We have also tested the SHA-1 HMAC (Hashing for Message Authentication Code) and RSA (asymmetric encryption and decryption) cryptographic algorithms. These Java applications are developed based on the example implementation provided in the Bouncy Castle API package. These examples also include test vectors from cryptographic standards such as NIST's FIPS documents or RFC documents. Table 2 shows the performance of the cryptographic algorithms running on the SA-1110 board. Note that the performance of each algorithm is the average execution time of running the algorithm once with different sets of data and key length provided in the example codes. In RSA, the decryption takes much longer than the encryption. This is because the public encryption key is much smaller than the decryption key, which is kept secret. Still both are two to three orders of magnitude slower than the symmetric key algorithms or the message digest algorithms.

Table 2. Performance of Cryptographic Algorithms on SA-1110 Board.

	Algorithm	Encryption (msec)	Decryption (msec)
Symmetric Cipher	AES	1.75	3.25
	DES	8.57	8.71
	RC2	1.75	2.00
	RC4	2.67	3.00
	RC5	1.41	1.11
	RC6	3.17	3.33
Message Digest	MD2Digest	12.71	
	MD4Digest	0.75	
	MD5Digest	0.75	
	SHA1Digest	2.20	
	SHA256Digest	5.25	
	SHA512Digest	8.25	
Other	SHA1HMAC	12.50	
	RSA	400.00	7000.00

From the results shown thus far, one can conclude that the performance of cryptographic algorithms on the KVM is feasible for real embedded systems. Although the performance is only around 100Kbits/s with a large memory footprint, the memory footprint can be reduced significantly by selecting only necessary API for a particular application. One way to improve the performance is by further tuning of the libraries and by using faster processors and memories. Another way to improve the performance is by selecting the computationally intensive routines and by implementing them on cryptographically secure co-processors.

3. PERFORMANCE OPTIMIZATION OF JAVA CRYPTOGRAPHIC APPLICATIONS

In this section, we provide the design framework for the performance optimization of cryptographic algorithms. We use the GEZEL design environment for modeling and simulating the hardware accelerator of cryptographic algorithms [12]. GEZEL consists of a specialized language that expresses the Finite State Machine and Datapath (FSMD) and its simulation environment. GEZEL also provides instruction-set co-simulation and VHDL code generation.

3.1 Design Hierarchy and Interface

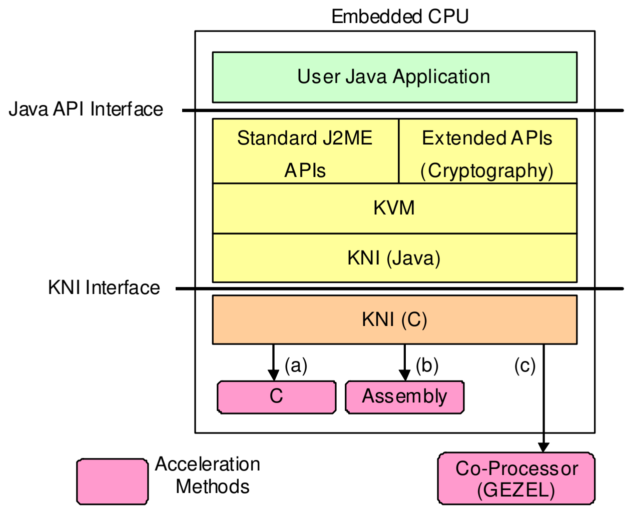


Figure 2. Design Hierarchy and Interfaces.

Figure 2 shows the concept of design hierarchy and interfaces between different languages for the performance optimization. The user application is running on top of the KVM, by using Java APIs provided by standard J2ME/CLDC class libraries and/or extended cryptographic libraries such as the Bouncy Castle APIs. This application, by default, runs directly on the processor as the native code. For the Performance optimization, we propose three methods of acceleration:

a) acceleration in C

b) acceleration in Assembly

c) acceleration in GEZEL

If the performance of the Java implementation of a cryptographic algorithm is not enough, the algorithm can be transferred from Java into C and executed via the K Native Interface (KNI) provided in J2ME/CLDC platform (Figure 2(a)). If further improvement is necessary, the algorithm can be implemented in the assembly language for the particular processor by writing inline-assembly in C source code (Figure 2(b)). The final solution for the acceleration is deploying a co-processor besides the main processor (Figure 2(c)). The co-processor can be a hardware unit which performs the algorithm in lower latency and higher throughput.

In all three cases, the proposed methods use the KNI. The KNI consists of a piece of code in Java and in C, providing a capability of passing and returning arguments. Based on this design hierarchy, the most significant merit is that the user application itself is written in Java and does not need to be modified for the performance optimization. The detail is provided later in the design example section.

3.2 Design Flow on the SH3-DSP Embedded Processor Core

3.2.1 The SH3-DSP Embedded Processor Core

The SH3-DSP embedded processor core has been chosen as a target platform of the design. SH3-DSP is a 32bit RISC microprocessor core with a DSP unit, and is also known as the core of the SH-Mobile processor [22]. The SH-Mobile processor is an application processor specialized for next-generation cellular phone communications, and has been embedded in many cellular phones all over the world, especially in Japan. We used an Instruction-Set Simulator (ISS) model of the SH3-DSP core, which uses a 16kbyte unified cache and two 8kbyte X/Y-RAM memories for DSP operations.

3.2.2 Design Flow

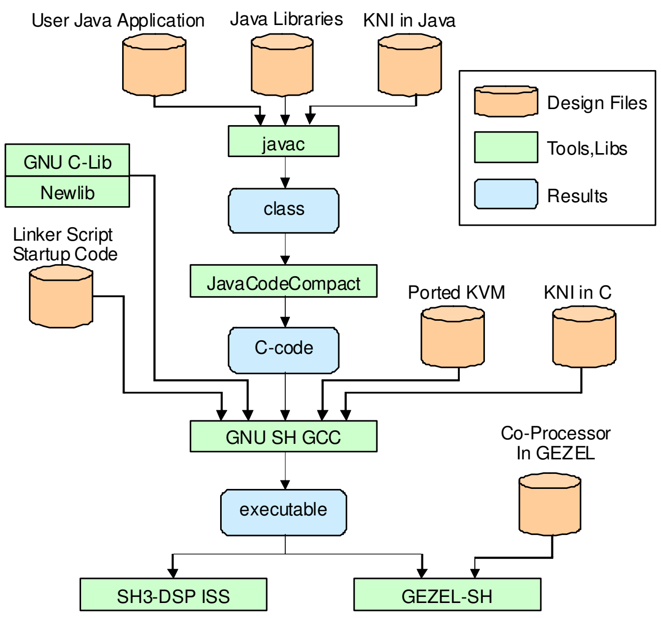
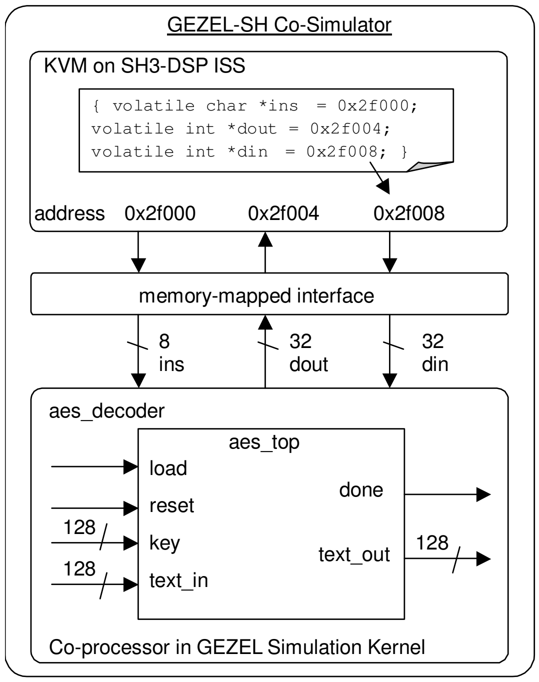


Figure 3. Design Flow.

Figure 3 shows the design flow of the optimization. The user application and libraries such as the Bouncy Castle API, and a part of KNI are written in Java, then compiled via the Java compiler (javac) and corresponding class files are generated. The JavaCodeCompact, the EmbeddedJava development tools provided by Sun, takes these class files as the input, and generates C codes (source and header file). We used the GNUSH_v0304 version of KPIT [13] cross-compiler to compile the C codes. In addition to ported KVM, KNI is also prepared in C to provide the interface between KVM and the acceleration methods described in the previous section. The GNUSH compiler takes these C codes and other configuration files (linker scripts and startup codes) and generates the executables. This binary is the input of either the SH3-DSP ISS or the GEZEL-SH simulation kernel. The SH3-DSP ISS is used for the simulation of the acceleration methods using C and assembly (as specified in Figure 2(a) and 2(b)), whereas the GEZEL-SH is used for the simulation of hardware co-processors (Figure 2(c)). The GEZEL-SH also takes a GEZEL description of the co-processor as the input, and performs the co-simulation of the software and hardware (co-processor) in a cycle true fashion.

3.3 AES Optimization Example

As a design example, the Advanced Encryption Standard (AES) algorithm has been tested in our design framework. The AES is a symmetric block cipher that can process data blocks of 128bits, using cipher keys with length of 128, 192, and 256bits as specified in Federal Information Processing Standard (FIPS) [14].



```
{ volatile char *ins  = 0x2f000;
volatile int *dout = 0x2f004;
volatile int *din  = 0x2f008; }
```

Figure 4. AES Optimization with GEZEL.

The AES is one of the most popular symmetric cipher algorithm used in cryptographic applications and tools.

As an example, we have developed an AES co-processor with a 128bit key length in GEZEL. The AES co-processor communicates with the KVM which is running on the SH3-DSP ISS via the memory-mapped interface as part of the GEZEL-SH co-simulator. Figure 4 shows the structure which corresponds to hardware acceleration in GEZEL (Figure 2(c)).

To evaluate the optimization methods, three cases were tested as shown in Figure 5 based on the design architecture described thus far. One AES encryption consists of two parts. One is the key-scheduling to prepare the round key, and the other is the actual block encryption with the subsequent round keys. In Figure 5(a), the user application calls the Java API function (`Key_J` and `Enc_J`), and the encryption is done in Java. No acceleration is performed. In Figure 5(b), the function call inside the Java API is substituted by the KNI function calls (`Key_K` and `Enc_K`) so that the acceleration can be performed in C. In Figure 5(c), the function call inside the KNI is substituted by the driver function calls (`Key_G` and `Enc_G`) so that the acceleration can be done in the GEZEL model. For both accelerations, the KNI and/or memory-mapped interface is used to pass and return the variables such as encryption key, plain text, and encrypted text.

For the evaluation, the number of cycles consumed per encryption is counted and the result is shown in Figure 6. All the numbers in the figure represent one iteration of the AES encryption (key-scheduling and encryption), starting from the Java function call in the user application. Startup overhead, such as setting up the C or Java runtime environment, is not included.

The AES encryption takes 198,741 cycles with the Bouncy Castle API (Figure 6(a)). In Figure 6(b), the AES encryption in C takes 29,008 cycles in which the overhead of Java and KNI interface is 18,763 and the actual encryption takes 10,245 cycles. Finally, the AES encryption in GEZEL takes 19,198 cycles in which the overhead of Java and KNI interface is 18,938 cycles

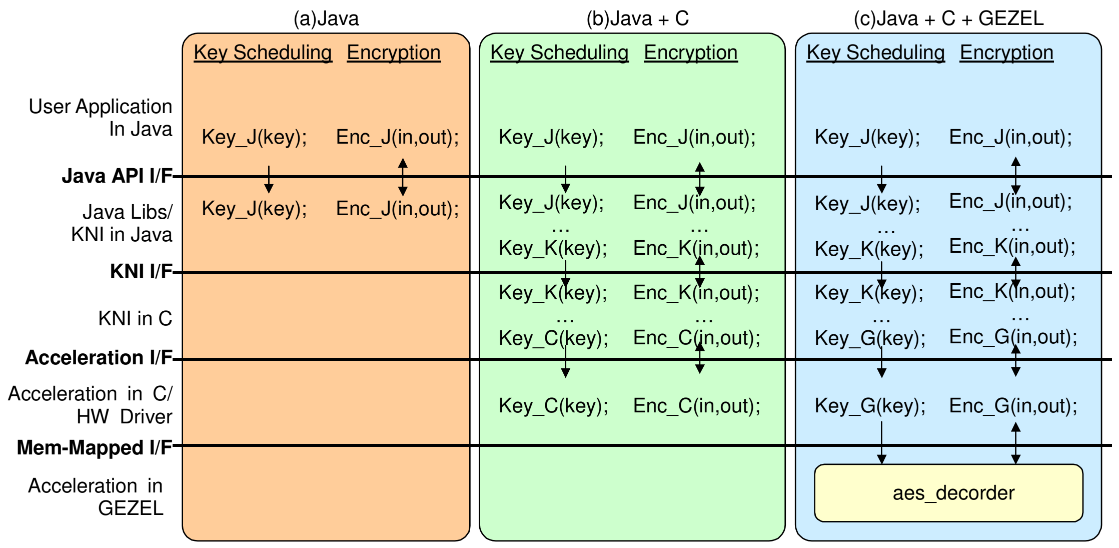


Figure 5. AES Optimization Architecture.

and the actual encryption takes only 260 cycles including the memory-mapped interface. The performance gain in going from Java to GEZEL is now 10.4x including Java and KNI overhead.

As one can see from this result, the Java and KNI overhead dominates the number of cycles consumed in an encryption and limits the overall performance gain. However, the major benefit of this design scheme is that the user application does not have to be modified for the performance optimization – i.e. the user uses the `Key_J` and the `Enc_J` function calls as specified in the Java API in all the three cases as shown in Figure 5. On the other hand, only minor modifications in the Java API and the addition of small pieces of code for the KNI are necessary for the performance optimization.

Based on this design scheme, a cryptographic application can be easily prototyped by using the cryptographic API, and the user can choose whether the performance optimization is necessary or not depending on the performance in Java. If the performance optimization is necessary, the user still has a choice of whether the optimization is done in C or in GEZEL. Thus, the user can have design flexibility in performance optimization.

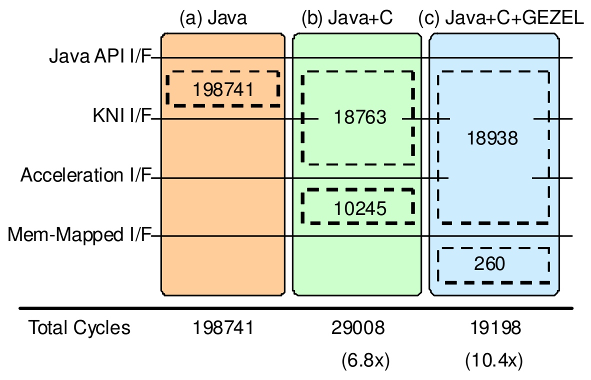


Figure 6. AES Optimization Results.

4. SECURITY OPTIMIZATION FOR KEY MANAGEMENT

4.1 Security Issues for Key Management in Java Cryptographic Extensions

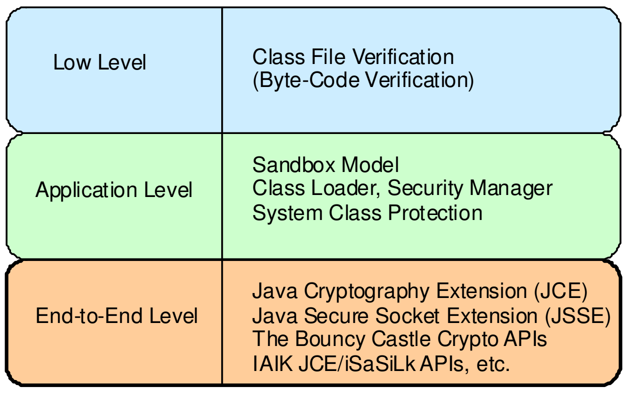

Low Level
Class File Verification (Byte-Code Verification)
Application Level
Sandbox Model
Class Loader, Security Manager
System Class Protection
End-to-End Level
Java Cryptography Extension (JCE)
Java Secure Socket Extension (JSSE)
The Bouncy Castle Crypto APIs
IAIK JCE/iSaSiLk APIs, etc.

Figure 7. Java Security Policy Levels.

Java has some built-in security features. Figure 7 shows the three different levels of security policies in Java, as described in the CLDC specification [4] and in the Java 2 Platform Security Architecture (JSA) [15]. The Java Cryptography Extension (JCE), the Java Secure Socket Extension (JSSE), and the other Cryptographic extensions such as the Bouncy Castle Crypto API and IAIK API are provided to ensure the End-to-End security in Java. End-to-end security refers to a model that guarantees that any transaction initiated on a device is protected along the entire path from the device to/from the entity providing the services for that transaction (e.g, a server located on the Internet). The End-to-End level security is not described in the CLDC specification and is assumed to be implementation-dependent. This fact – there is no standardized method for End-to-End security – motivates the need for a trustworthy implementation. This section focuses on the End-to-End level security issues of the JCE and the Bouncy Castle Crypto APIs from the key management point of view.

More specifically, we first address the lack of security in secret (symmetric) key management in JCE and the Bouncy Castle Crypto APIs for the Symmetric Key Infrastructure (SKI).

In the SKI environment, the secret key has to be protected against exposure. The management of secret keys can be categorized in the following two cases:

Case 1. Static Key: The key is unique for each device, and it will never be changed.

Case 2. Dynamic Key: The key value is changed or generated by the software or hardware as necessary – e.g. session keys.

For Case 1 (static key management), the Java Cryptography Architecture (JCA) [16] provides the key interfaces in the KeyStore class. The KeyStore class is an in-memory collection of keys and certificates for systems with I/O streams - e.g. file system. The secret key is stored in the KeyStore object via password-based encryption. Using this interface, the key is stored safely against exposure. However, in order to perform the encryption algorithm on a common architecture with the CPU and the main memory, the key has to be decrypted and loaded into the memory. This also means that the key can be easily exposed by monitoring/probing the system bus or main memory. Furthermore, the KeyStore class and the JCA's Security Tools assume the existence of a file system in the target system. This is not suitable for small embedded systems such as cellular phones in terms of the cost as mentioned in section 1. Consequently, the Bouncy Castle API has the same type of threats for Case1. Even though the key is stored in a safe place, the key has to be loaded into an object (instance) of a Java class in the main memory via the standard I/O stream of the system.

```
KeyGenerator kgen = KeyGenerator.getInstance("AES");
SecretKey skey = kgen.generateKey();
byte[] raw = skey.getEncoded();
SecretKeySpec skeySpec =
      new SecretKeySpec(raw, "AES");

Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
byte[] encrypted =
      cipher.doFinal("This is just an example".getBytes());
```

(a) JCE

```
SecureRandom sr = new SecureRandom();
AESkey = new byte [16];
sr.nextBytes(AESkey);
CipherParameters params = new KeyParameter(AESkey);

BufferedBlockCipher cipher =
      new BufferedBlockCipher(new AESFastEngine());
cipher.init(true, params);
int len = cipher.processBytes(in, 0, in.length, out, 0);
```

(b) Bouncy Castle API

Figure 8. Key Management Examples.

Figure 8 shows examples of Case 2 (dynamic key management) in JCE and the Bouncy Castle API for the key management and the AES encryption. In both cases (Figure 8(a) and (b)), the secret key is generated and stored in an object in the heap region of the virtual machine. The heap region is allocated in the main memory of the system, and the key is read from or stored into the main memory via the system bus. Therefore, as in the static key management case, the key can be easily exposed by monitoring/probing the system bus or main memory.

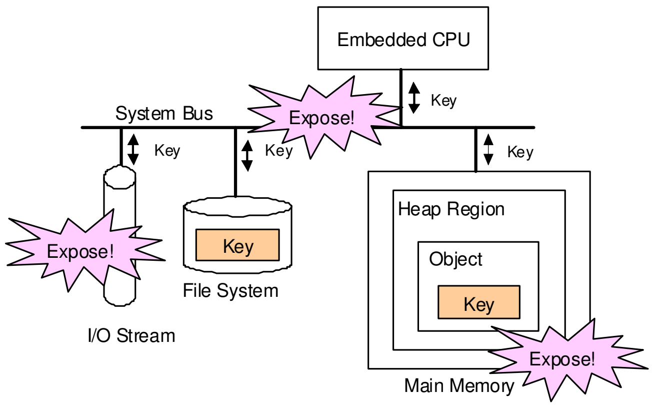


Figure 9. Threats of Secret Key Exposure.

Figure 9 depicts the threats of secret-key exposure discussed so far. The system consists of the CPU, main memory, file system, and the I/O interface to the external storage or the communication channel. The threats of exposure exist in all the interfaces between the CPU and the storage.

In addition to these types of threats, the standard CPU-based system has a fatal weakness against the so-called Side-Channel Attacks (SCAs) [17]. SCAs are a real threat for any device of which the security IC is easily observable such as smart cards and embedded devices [23][24]. One of the most powerful SCA is Differential Power Analysis (DPA) [18]. The DPA attack is non intrusive and exploits the fact that logic operations have power characteristics that depend on the input data (secret keys). The DPA employs statistical analysis to extract the information from variations in power consumption that are correlated to the secret key. As a countermeasure for DPA, a design of secure Elliptic Curve Cryptography (ECC) on VLIEW DSP processor has been proposed by the instruction-level modification such that the number of instructions and arithmetic operations are independent of the input data [25]. This type of algorithm modification needs to be done for every new crypto algorithm. Our proposed solution, by using Wave Dynamic Differential Logic (WDDL), is at a different abstraction level – i.e. logic level. Indeed if the underlying logic gates can be made such that its power signature is independent of the input data, it becomes algorithm independent. For the rest of this paper, we propose a design flow that enables the cryptographic algorithms to be computed on DPA resistant co-processors.

4.2 Proposed Approach

The key should not be generated in Java or C, nor stored in an unsafe location such as the main memory to avoid the exposure shown in Figure 9. Figure 10 shows the two implementations of the key management with the DPA resistant co-processor in terms of the Case 1 and Case 2 in the previous section.

Case 1. The static key is stored in DPA resistant hardware, and the cryptographic algorithm is executed on DPA resistant hardware. The key is not loaded into the main memory or the CPU.

Case 2. The static key management is the same as Case 1. The dynamic key is either generated by the DPA resistant crypto unit and stored in DPA resistant hardware, or is read from the I/O stream of the system in the encrypted

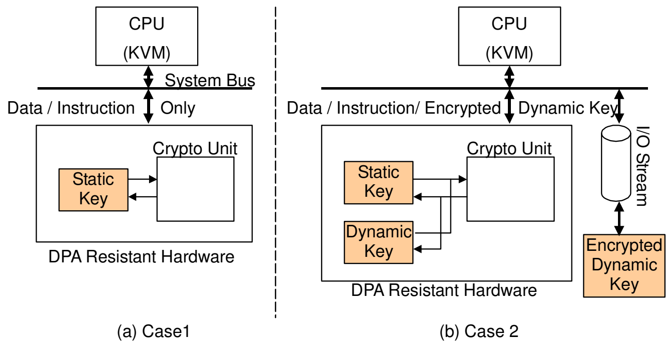


Figure 10. Key Management with DPA Resistant Co-Processor.

form. The static key can be used for the encryption of the dynamic key.

In both cases, the cryptographic algorithm is executed inside of the DPA resistant co-processor, concealing the secret key against the DPA attacks.

Figure 11 shows the design flow of the proposed approach. As described in section 3, the cryptographic algorithm is performed on the co-processor written in GEZEL. First, the co-processor is converted to the synthesizable VHDL description by the VHDL converter (fdlvhd) for GEZEL [19]. Once the VHDL code is generated, we follow the design flow proposed in [20]. Using WDDL the co-processor is made DPA resistant. This DDL style is a set of logic components that consume the same amount of charge/discharge power in all the clock cycles, and this is independent of the inputs signals. The DDL is an effective way to conceal the characteristics of power consumption related to the input data (secret key) against DPA [21]. The DDL library consists of inverter, AND-type, OR-type, and register types of logic components. The VHDL code is synthesized by a logic-synthesis tool with only a subset of the standard cell library so that they can be replaced by the DDL components later. Subsequently, the script replaces all the standard cells in the gate-level netlist with the corresponding DDL cells. The resulting DDL netlist can be put on a modified place and route tool to perform the physical layout.

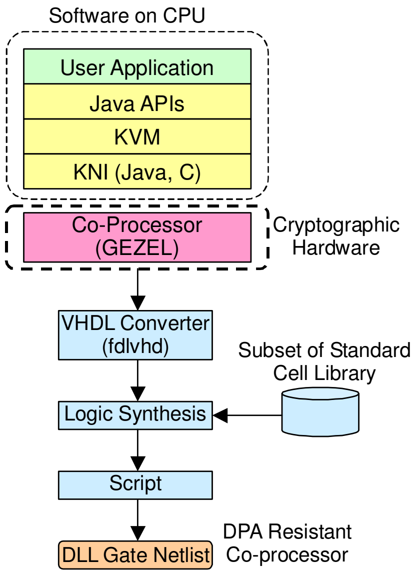


Figure 11. Security Optimization Design Flow.

4.3 Key Management Example in AES and its Performance

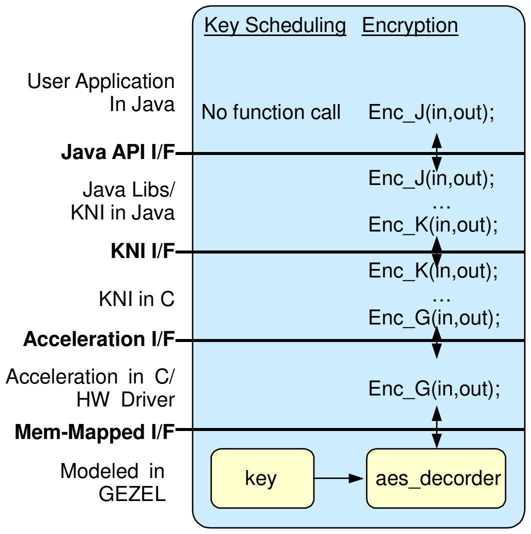


Figure 12. AES Key Management Example.

As an example of our proposed method, the AES design in section 3 has been modified such that the key is not stored in the Java or C program, and is not passed as a parameter over the interfaces. Figure 12 shows the modification in which the function call for the key scheduling has been removed from the user application. Instead, one instruction is added for the co-processor to load the secret key from secure storage. This key-load instruction is inserted right before the data-load instruction in the hardware driver function (`Enc_G`). The key storage is modeled as a register in GEZEL, and it takes 4 clock cycles to load a 128bit key.

For the architecture shown in Figure 12, the number of clock cycles consumed for one AES encryption was counted as shown in Figure 13. The number of cycles for the overhead of Java and KNI interface is reduced from 18,938 (Figure 6) to 7,642. The overall performance gain is 25x compared to the original Java implementation and the encryption speed is 2.16Mbits/sec assuming that the SH3-DSP and the AES co-processor is running at 133MHz, including the overhead of Java and KNI interface.

To evaluate the cost of the proposed security optimization, the AES co-processor (aes_decoder module in GEZEL) is

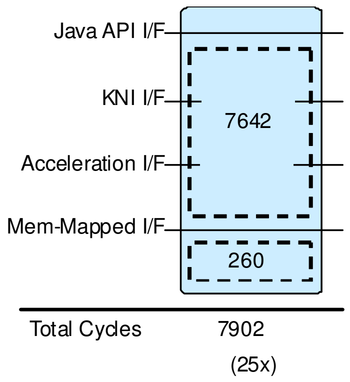


Figure 13. Optimized AES Encryption Result.

implemented in WDDL using the design flow as specified in Figure 11. The evaluation is performed with two versions of the AES co-processor, one for a fast design, and the other for a small design. This is done in the logic synthesis process. Table 3 shows the speed and area cost of the original (non-DDL) and the DDL gate-level netlist using a 0.18 mm CMOS standard cell technology.

Since WDDL consumes the same amount of charge/discharge power at all clock cycles independent of the input value, it costs more logic gates and delays for the same logical functionality. One observation from this result is that the cost of the security optimization is approximately 0.8nsec in the critical-path delay, and 1.92x to 2.33x in the area. On the other hand, the power consumption variations will be reduced by the factor of 37 to 52 by the DDL [20].

The DDL is based on a precharge and evaluation cycle. During the precharge cycle, a reset travels through the circuit. During the evaluation cycle, the actual calculations take place. Therefore new inputs can only be applied every 8.46nsec and 21.36nsec for the fast design and the small design, respectively. This overhead is still very small compared to the overall number of clock cycles since it only adds to the encryption operation and not to the communication interface which is the limiting factor in system performance.

Table 3. Speed and Area Cost of the AES Co-Processor.

		Critical Path Delay (nsec)	Area Cost (Kgates)	Reduction Factor of Power Variation
Fast Design	Non-DDL	3.44	53.2	37 to 52 [20]
	DDL	4.23 (+0.79 of non-DDL)	102.2(1.92x of non-DDL)	
Small Design	Non-DDL	9.84	26.3	
	DDL	10.68 (+0.84 of non-DDL)	61.3 (2.33x of non-DDL)	

5. CONCLUSION

This paper describes a design framework of Java cryptographic application for secure embedded systems that enables a DPA resistant implementation of co-processors and a performance improvement of the cryptographic application.

The two Java cryptographic libraries, the Bouncy Castle APIs and the IAIK APIs were first ported to a real embedded device. The cost and performance were evaluated. The cost ranged in 0.88Mbytes to 1.2Mbytes in the KVM footprint size and the performance resulted in a few milliseconds for secret key algorithms and message digests on the SA-1110 processor, running at 206MHz.

The design framework for performance optimization was then presented using the GEZEL environment for hardware modeling and for co-design between the hardware co-processor and the software on KVM. The simulation results of the AES example on the SH3-DSP microprocessor show a performance gain of 10.4x including the overhead in Java, C, and hardware interfaces.

For further performance and security optimization, a design method was introduced for secret key management in a co-processor protected against eavesdropping. The design method is based on a DDL that is resistant to DPA attacks. In the optimized architecture for key management, the overall performance gain was 25x. The AES co-processor was mapped onto the DDL in 0.18mm CMOS standard cell technology to evaluate the performance and the area overhead.

The proposed design framework has the following main benfits:

1) Fast prototyping using platform independent Java cryptographic APIs.

2) Easy customizability by providing a way to partition software and hardware to optimize for performance without modifying the application code.

3) Security enhancement by allowing design of coprocessors resistant to DPA attacks.

This paper shows that there exist systematic ways to improve the performance of embedded security. Traditionally, the HW accelerators and the embedded software routines are developed separately in an ad-hoc way. Often up-front, a decision is made of what parts will go in HW and what parts will go in SW. Our approach allows a designer to gradually move from only SW to a hardware accelerated design combined with a new WDDL style.

6. ACKNOWLEDGMENTS

The authors acknowledge the funding of NSF (Account no CCR-0310527) and UC Micro (Account no 02-079). The authors also acknowledge Renesas Technology Corp. Japan for providing the SH3-DSP Instruction Set Simulator and for funding the first author.

7. REFERENCES

[1] M. Renaudin, F. Bouesse, Ph. Proust, J. P. Tual, L. Sourgen, F. Germain, “High Security Smartcards”, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Volume I (DATE'04) pp. 228-233

[2] Bruce Schneier, “Applied Cryptography”, John Wiley & Sons, 1996 ISBN 0-471-12845-7

[3] J2ME Building Blocks for Mobile Devices - White Paper on KVM and the Connected, Limited Device Configuration (CLDC) http://java.sun.com/products/cldc/wp/KVMwp.pdf

[4] J2ME CLDC 1.1, http://java.sun.com/products/cldc/index.jsp

[5] Java Devices, http://www.microjava.com/devices

[6] Japan NTT DoCoMo's i-mode Article, http://www.peterindia.net/i-ModeView.html

[7] The Bouncy Castle Lightweight API Release 1.20, http://www.bouncycastle.org/download/lcrypto-j2me-120.tar.gz

[8] IAIK JCE and iSaSiLk APIs, http://jce.iaik.tugraz.at/download/evaluation/index.php

[9] Intel SA-1110 Processor, http://www.intel.com/design/edk/product/strongarm_edk.htm

[10] The eCos OS, http://sources.redhat.com/ecos

[11] Michael Yuan, “Enterprise J2ME: Developing Mobile Java Applications”,http://www.enterprisej2me.com/pages/enterprisej2me/book.php

[12] The GEZEL Design Environment, http://www.ee.ucla.edu/~schaum/gezel/

[13] KPIT Cummins GNU Tools & Support, http://www.kpitgnutools.com/

[14] Advanced Encryption Standard, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[15] Java 2 Platform Security Architecture, http://java.sun.com/j2se/1.4.2/docs/guide/security/

[16] Java Cryptography Architecture, http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html

[17] E. Hess, N. Janssen, B. Meyer, T. Schuetze, “Information Leakage Attacks Against Smart Card Implementations of Cryptographic Algorithms and Countermeasures – a Survey”, EUROSMART Security Conference (2000) pp.55–64

[18] P. Kocher, J. Jaffe, B. Jun, “Differential Power Analysis”, Proc. of Advances in Cryptology (1999) pp.388-397

[19] GEZEL User Manual, http://www.ee.ucla.edu/~schaum/gezel/gzldata/gezelum.pdf

[20] K. Tiri, I. Verbauwhede, “A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation”, Design Automation and Test in Europe Conference (DATE 2004) pp.246-251

[21] K. Tiri, I. Verbauwhede, “Securing Encryption Algorithms against DPA at the Logic Level: Next Generation Smart Card Technology”, Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003) pp.125–136

[22] SH-Mobile Application Processor, http://www.renesas.com/eng/products/mpumcu/shmobile/index.html

[23] P. Kocher, R. Lee, G. McGraw, A. Raghunathan and S. Ravi, “Security as a New Dimension in Embedded System Design”, Proc. of 41st Design Automation Conference (DAC 2004), 2004, pp.735-760

[24] S. Ravi, A. Raghunathan and S. Chakradhar, “Tamper Resistance Mechanisms for Secure Embedded Systems”, Proc. of 17th International Conference on VLSI Design (VLSID 2004), 2004, pp.605-610

[25] C.Gebotys, "Design of Secure Cryptography against the threat of power-attacks in DSP embedded processors", ACM Transactions on Embedded Computer Systems, Vol. 3, No. 1, February 2004