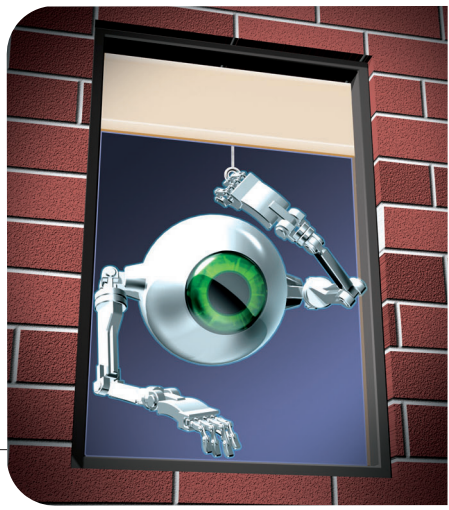
Securing Embedded Systems

A top-down, multiabstraction layer approach for embedded security design reduces the risk of security flaws, letting designers maximize security while limiting area, energy, and computation costs.



David D. Hwang, Patrick Schaumont, Kris Tiri, and Ingrid Verbauwhede
University of California, Los Angeles

The embedded systems field is growing rapidly, with devices such as cellular phones, PDAs, smart cards, and digital music players permeating society. On the horizon are futuristic technologies such as embedded network sensors and wearable computers, which promise even greater interaction between humans and machines.

As embedded devices are increasingly integrated into personal and commercial infrastructures, security becomes a paramount issue. For example, if a patient is wearing a heart-monitoring device that sends data wirelessly to a doctor, the embedded system must keep this information confidential and deliver it uncorrupted to the doctor. An embedded network sensor monitoring water quality to prevent bioterrorism must have multiple methods to detect tampering in both hardware and software, lest an attacker bypass security measures and corrupt the water supply.

The design of security for embedded systems differs from traditional security design because these systems are resource-constrained in their capacities (and consequently in their defenses) and easily accessible to adversaries at the physical layer. Embedded security can't be solved at a single security abstraction layer, but rather is a system problem spanning multiple abstraction levels. We use an embedded biometric authentication device to demonstrate the necessity of addressing all levels of the security pyramid to ensure a fully robust and secure embedded system. (See the "Related Work and Design Alternatives" sidebar for a discussion of some other solutions to the embedded systems security problem.)

Embedded design challenges

Embedded systems are essentially processor-based devices operating under resource-constrained conditions. Embedded devices include systems as diverse as automobile microcontrollers, cellular phones, smart cards, embedded network sensors, and digital cable boxes. These devices are often portable, communicate via wireless channels, and are battery-powered or otherwise energy-limited. Because these systems are often considered small computers, it's tempting to port workstation-based security techniques directly onto the devices to make them secure. However, embedded systems have characteristics that differentiate their security architecture from that of workstations and servers. We group these characteristics into two categories: resource limitation and physical accessibility.

Resource limitation

Embedded devices pose severe resource constraints on the security architecture in terms of memory, computational capacity, and energy. For example, the SmartDust node[1] is a battery-powered device possessing an 8-bit, 4-MHz CPU with 8,000 bytes of instruction flash memory, 512 bytes of RAM, and 512 bytes of EEPROM.

Clearly, such a platform severely limits potential security scenarios. In terms of memory, sophisticated public-key cryptography techniques such as RSA or elliptic-curve cryptography might simply be infeasible. Considering the device's 4-MHz computational horsepower, certain protocols could cause too much latency to be useful. Furthermore, an energy-intensive security scheme can cause the node to perish from battery exhaustion before it can perform useful work. Power dissi-

pation and other resources are infrequently major concerns when dealing with workstation-based security.

Physical accessibility

Devices such as stolen smart cards or compromised sensor nodes are easily accessible at the physical layer. This accessibility has led to several new security attacks in recent years in the areas of physical tampering and side-channel analysis. For example, Paul Kocher, Joshua Jaffe, and Benjamin Jun's differential power analysis (DPA) attack shows that an adversary can monitor a smart card's power line to extract the card's cryptographic key.[2] Such attacks might be irrelevant to workstation-based security but are extremely important in embedded system design.

An issue related to physical accessibility and portability is privacy, particularly with networked embedded systems. For example, the recent US Federal Communications Commission (FCC) mandate for enhanced 911 (the number citizens in North America call in case of emergency) requires a user's location to be available to an emergency dispatcher, often through GPS devices on cell phones. This mandate has sparked a debate between safety and privacy advocates. In addition, as camera phones have pervaded society, laws have banned phones in places such as locker rooms and courtrooms because of potential privacy violations. In general, networked embedded systems have caused many new sociological and legal issues to emerge.

Storing data on an embedded system also creates privacy concerns. On the one hand, storing sensitive information on a device rather than on multiple servers minimizes the number of locations where an attack can occur. On the flip side, small devices can be easily lost or stolen, and hence must have extra security measures built in to ensure that private data can't be compromised.

Embedded security pyramid

Because of these unique characteristics, we can't solve embedded security at a single level of abstraction—that is, it's not a point problem. Rather, embedded security is a system problem that we must solve at all abstraction levels.

The security pyramid in Figure 1 illustrates the five primary abstraction levels in an embedded system:

- *protocol* level, which includes the design of protocols to be performed on embedded devices to achieve such security goals as confidentiality, identification, data integrity, data origin authentication, and nonrepudiation;
- *algorithm* level, consisting of the design of cryptographic primitives (such as block ciphers and hash functions) and application-specific algorithms used at the protocol level;
- *architecture* level, consisting of secure hardware/software partitioning and embedded software techniques to prevent software hacks;
- *microarchitecture* level, which deals with the hardware design of the modules (the processors and coprocessors) required and specified at the architecture level; and
- *circuit* level, which requires implementing transistor-level and package-level techniques to thwart various physical-layer attacks.

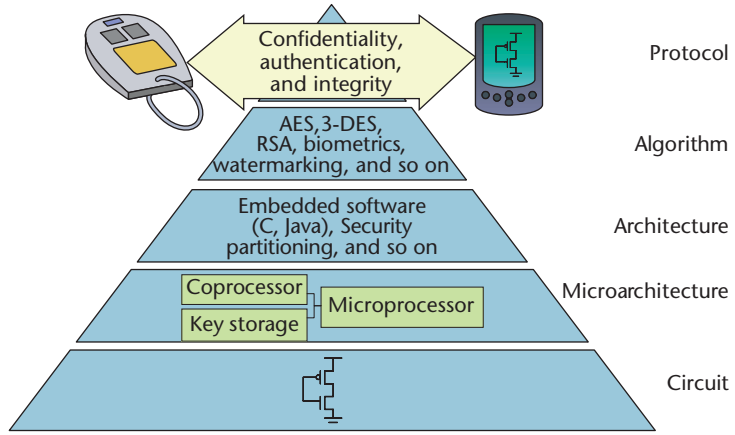


Figure 1. Embedded security pyramid. To ensure security in an embedded system, we must address the problem in all abstraction layers.

For an embedded system to be secure, every level must be secure. For example, a smart card can possess an advanced protocol applying a strong cipher; however, if the circuit design allows for side-channel attacks that can extract the key, the smart card's security is broken.

We categorize security issues into two general types: *single-level* security issues, in which the problem and the remedy are at the same abstraction level; and *translevel* security issues, in which we can fix a problem at one level only with a remedy at another (lower) level. We discover such issues by examining the subtle interrelationships between levels.

Building the pyramid

The wireless biometric authentication device shown in the upper left of Figure 1 is our embedded system's driver application. With a form factor of a keychain dongle, this device consists of a complementary metal-oxide semiconductor (CMOS) fingerprint sensor, a 32-bit reduced-instruction-set computing (RISC) microcontroller, specialized security and biometric hardware, an infrared wireless transmitter, and embedded memory. The device's primary function is to facilitate secure biometric authentication between a user and a server in applications such as intelligent keys, credit card and smart card replacement, and access control.

Figure 2a illustrates a classic server-based fingerprint verification scheme. In this method, a user enters a

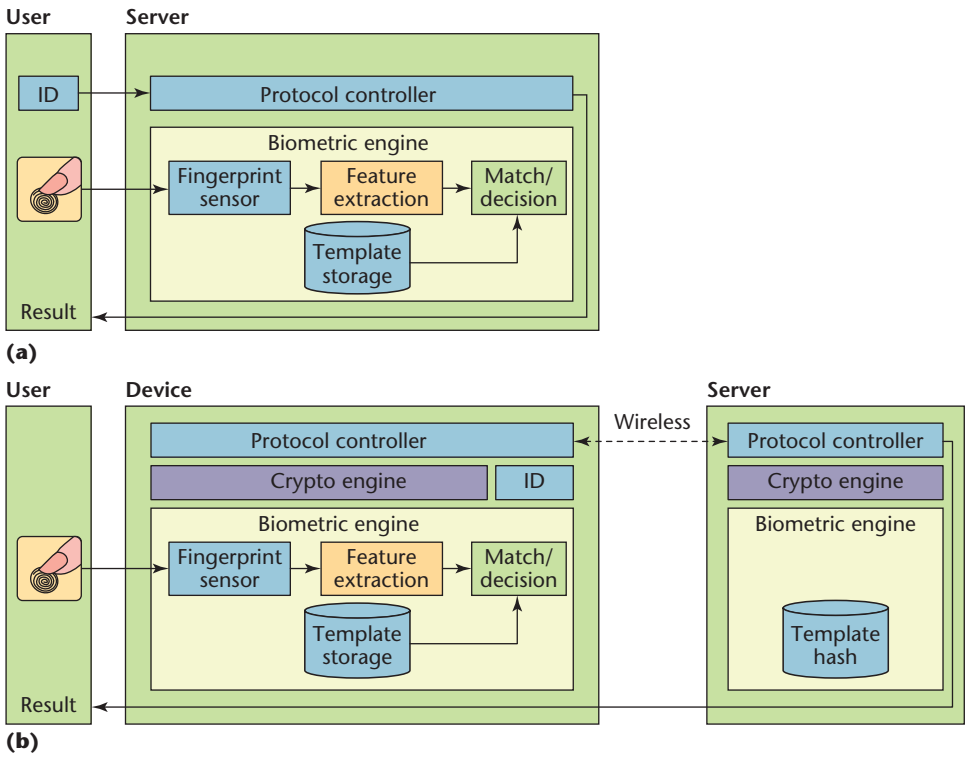


Figure 2. Biometric authentication. (a) In a server-based fingerprint authentication scheme, the server matches a user's fingerprint with a previously stored template. (b) In a device-based scheme, a user authenticates directly with the embedded device.

claimed identity (which can be stored on a magnetic card or smart card) into a server. After the server validates the claimed identity, the user impresses a fingerprint on the server's sensor. The server extracts the fingerprint's unique features and matches them with a previously stored template. The server decides to corroborate or deny the user's claimed identity based on a matching threshold.

Although this verification scheme is most common, it possesses potential security flaws. For example, a residual fingerprint image stays on the public sensor, where it's easily accessible for biometric theft and illegal reproduction. In addition, storing the actual biometric on a server can have sociological ramifications because every bank, employer, and computer a user interacts with must also store the template, which can be considered a privacy violation.

In a device-based scheme, shown in Figure 2b, a user authenticates directly with an embedded device (using biometric techniques), which wirelessly authenticates both itself and the user with a fixed server (using cryptographic techniques). The fingerprint sensor and the template are located on the device, avoiding the server-based security issues. In addition, all biometric-processing algorithms are performed on the device, localizing sensitive biometric data. The server stores only a keyed hash of the biometric template which, because of its noninvertible nature, doesn't violate the user's privacy. Other security issues exist, however, as we'll discuss later.

Protocol level: Wireless biometric verification

We designed our security protocols to operate on an embedded implementation. For example, the verification protocol in Figure 3 combines elements of challenge–response symmetric-key authentication and biometric verification. The protocol assumes that a shared key K exists between the device and the server. The device stores the template, while the server stores a hash of the template H(K, TEMPLATE).

The device initiates a verification transaction by transmitting its ID to the server. After corroborating the device's identity, the server sends two random numbers, RAND and RANDT, to the device (in the figure, | denotes concatenation). The device receives these values and begins the biometric-verification protocol. After obtaining the user's fingerprint locally, the device extracts the candidate minutiae and compares them with the stored template. If the match is positive, the device loads the key K and generates the hash of the template H(K, TEMPLATE). If the match is negative, the device loads a dummy key,

setting K = 0, and a dummy template, setting H(K, TEMPLATE) = 0.

Next, the device encrypts RAND to create a session key SK = E(K, RAND). It uses the session key to encrypt RANDT concatenated with H(K, TEMPLATE), producing an authentication token TOKEN = E(SK, RANDT | H(K, TEMPLATE)), which it forwards to the server. The server decrypts the token and the transmitted template hash H(K, TEMPLATE), and compares the template hash with the stored hash to check the device's authenticity. The server then sends a final transaction result to the device and lets it access the system if all tests check.

A single-level security flaw at the protocol level could allow a server masquerade attack. Because the device never authenticates the RAND | RANDT values, a false server could easily send these values undetected. We can fix this at the protocol level by requiring the server to send an additional hash, say H(SK′, RAND | RANDT | ID), to the device, where SK′ is a session hash/message authentication code (MAC) key. The device can then authenticate the server at each transaction. Other solutions at this level include using a MAC to protect token integrity, sequence numbers to prevent replay attacks, or a different key for token hashing.

Figure 3 shows a software bypass attack, an example of a translevel security flaw. In this attack, the adversary inserts malicious software into the device to bypass the biometric functions. Directly after receiving the random numbers, the hacked program loads the key K, falsely telling the device that a match has been made. This attack effectively breaks the biometric tie between the user and device, letting anyone use the hacked device without a correct fingerprint. The server assumes that the device is operating properly, so it allows unauthorized access to system resources. We can't fix the software bypass attack at the protocol level because it modifies any protocol it encounters. Instead, we must take measures at the architecture and microarchitecture levels.

Our protocol performs all biometrics on-device for maximum security. We can design a suite of protocols that variously partitions the biometric functions between the device and server, based on the limitations of embedded performance, latency, energy, and memory. Protection mechanisms similar to those used for the secret key should safeguard the template; in fact, greater protection mechanisms might be necessary because keys are replaceable, whereas biometrics are not.

Future directions for template storage involve storing a transformation (hash) of the template and performing a match directly in the transformed space. Further considerations of biometric security are available elsewhere.[3] Alternatives to embedded biometrics for authentication include smart cards, RFID tags, and secure authenticators such as RSA SecurID.

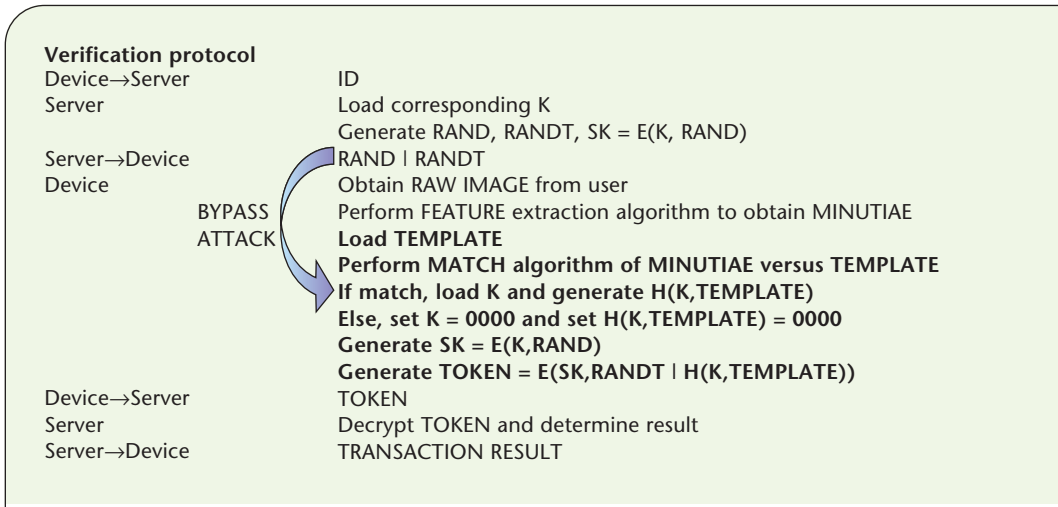


Figure 3. Device-based verification protocol. The device interacts with the user and the server to perform a biometric authentication transaction where all biometrics are performed on the device. (Bold text signifies secure functions.)

Algorithm level: Embedded biometric signal processing

At the algorithm level, a designer must select or design both cryptographic and application-specific algorithms for implementation on the embedded device. We implemented an Advanced Encryption Standard-128 cipher (128-bit key, 128-bit data) for the encryption operator—encryption on plaintext P with a key K produces ciphertext C = E(K, P). We similarly implemented the keyed-hash function via the standard cipher-block chaining (CBC) MAC mode, hashing a variable-length data stream D to a fixed 128-byte hash = H(K, D).

We also developed two embedded fingerprint signal-processing algorithms at this level:

- a *feature-extraction* algorithm, which extracts the minutiae from the raw image; and
- a *matching* algorithm, which performs a matching operation between these minutiae and a stored template.

We based our feature-extraction algorithm on a US National Institute of Standards and Technology floating-point algorithm. However, because this algorithm was intended for workstation-based computing, we converted it into fixed-point representation and optimized it for memory and energy for inclusion in the embedded device. We designed the matching algorithm as a local-neighborhood matching scheme for implementation on

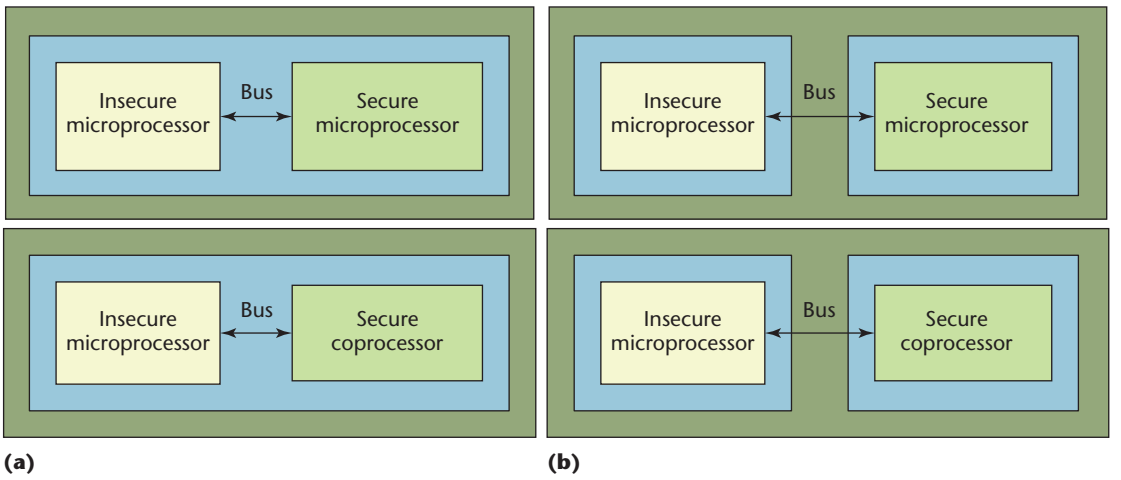


Figure 4. Partitioning topology strategies. (a) A single-chip solution partitions a chip into secure and insecure modules connected by a secure-to-insecure bus structure. (b) A dual-chip solution implements the same functionality as a single-chip solution but uses separate chips for greater design flexibility.

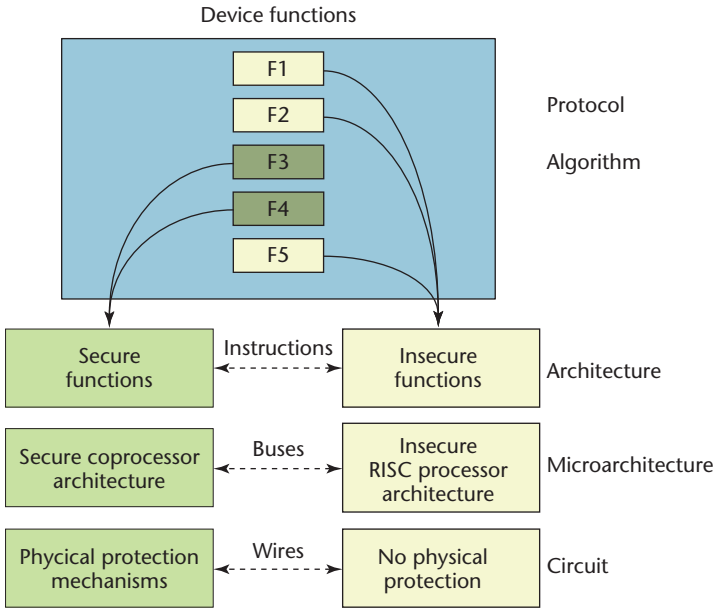


Figure 5. Partitioning of secure functions (mapping) phase. At this phase, we map functions in the protocol onto the secure or insecure modules in the architecture.

embedded devices. The experimental error rates for the combined algorithm are a 1-percent false reject rate (FRR) and a 0.01-percent false accept rate (FAR).

Architecture level: Security partitioning

At the architecture level, the protocol and algorithms must be mapped onto an embedded architecture platform. At this stage, we partition the device into secure and insecure modules. Secure modules are hardware modules that run secure functions, house secure memory, and use various hardware and software techniques to protect themselves. Insecure modules run insecure functions, house insecure memory, and aren't protected from attack.

Because providing security has a cost overhead in terms of area, power, and computation, we must clearly distinguish between the protocol's secure and insecure parts. Then, we can provide security to the secure parts only, reducing the overhead. Basically, we try to confine the secure parts to the smallest possible portion of the system. At the protocol level, we move secure parts from server to device; at the architecture level, we move them to a limited area of the device (the secure module). We call this *security-driven hardware–software partitioning*, or *security partitioning*. Security partitioning isolates the device's sensitive data and functions so both software (architecture-level) and physical (circuit-level) mechanisms can protect them.

Security partitioning is the application of a variant of Kerckhoffs' principle, which is to minimize the number of secrets in a system. If the device as a whole is physically compromised, it remains secure as long as the secure module is intact. The technique addresses the software bypass attack as well as resource limitation and physical accessibility issues. Later, we'll discuss a technique to secure a system at the circuit level, which requires approximately twice the normal area and power. Securing only a required subset of the system thus addresses area and energy limitations.

Partitioning the device into secure and insecure mod-

ules involves four primary steps at the architecture level: determine partitioning topology, determine coupling and secure-to-insecure bus structure, partition functions and their data, and construct a secure instruction set.

Partitioning topology. Figure 4 shows a partitioning topology that defines how a device is divided into secure and insecure modules. Consider the single-chip solution in Figure 4a. It partitions a single chip into two modules connected by a secure-to-insecure bus structure. The two modules can consist of either an insecure microprocessor and a secure microprocessor, or an insecure microprocessor and a secure coprocessor. Using a general-purpose microprocessor for the secure module provides for easy programmability and short design time, but wastes area and power. A custom-designed secure coprocessor is efficient in area and performance, but dramatically increases design time.

Figure 4b shows a dual-chip solution, which lets us design and fabricate each module independently, but the secure-to-insecure bus is physically exposed and accessible on a board, rather than contained in the chip package. The solution can also require additional power and interface circuitry and bus-handshaking protocols.

Our embedded system architecture uses the single-chip insecure microprocessor and secure coprocessor topology. The microprocessor is a 32-bit RISC processor called Leon (www.gaisler.com), which is an embedded Sparc V8 open source core. It possesses an arithmetic logic unit and data and instruction caches, as well as an advanced microprocessor bus architecture (AMBA) containing interfaces with memory, universal asynchronous receiver–transmitter, and other peripherals. We custom designed the secure coprocessor, which we describe later.

Coupling and bus structure. Assuming a fixed topology, we next determine the coupling between modules. The secure and insecure modules can be loosely coupled (memory-mapped) or tightly coupled (register-mapped). In general, loosely coupled coprocessors have performance advantages over tightly coupled coprocessors but design time disadvantages. Our coupling-mechanism selection directly determines the secure-to-insecure bus structure, which is the only means of communication between the secure and insecure modules. We selected a memory-mapped coprocessor with three buses shared between the secure and insecure modules: a 16-bit instruction bus INS, a 32-bit data-in bus D_IN (from insecure to secure), and a 32-byte data-out bus D_OUT (from secure to insecure).

Partitioning of secure functions, or mapping phase. Next, we map functions in the protocol onto the architecture's secure or insecure modules, as Figure 5 illustrates. A secure function contains secret or sensitive information as an input, output, or intermediate internal value. If such a function operates in an insecure location, puts sensitive data on a publicly exposed bus, or is executed out of sequence, it could potentially leak information. A clear example of a secure function is Figure 3's "If match, load K" function.

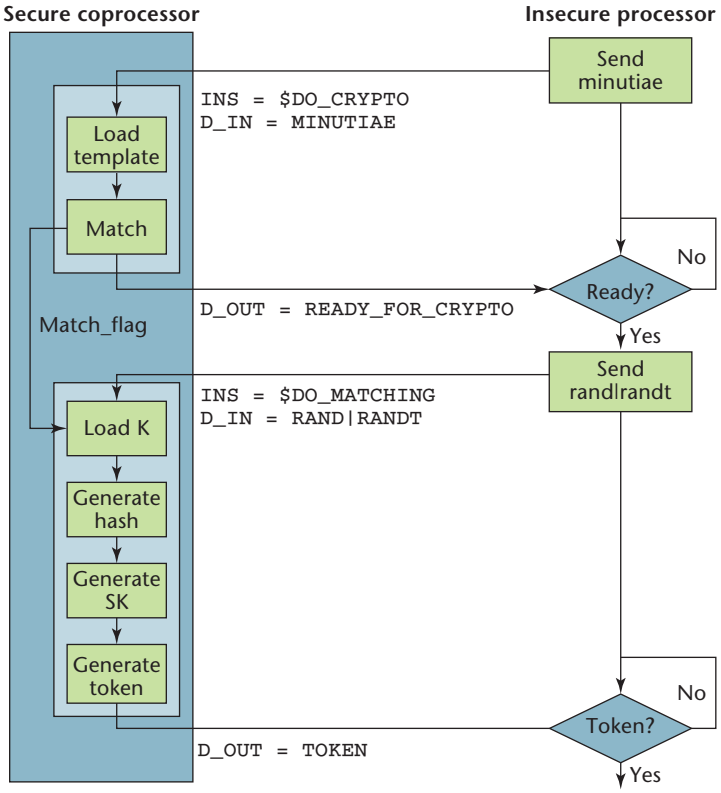


Figure 6. Secure instruction set control flow. The insecure processor accesses the coprocessor through the $DO_MATCHING and $DO_CRYPTO instructions. The first loads the stored template and matches it against the candidate template; the latter instructs the secure coprocessor to perform cryptographic steps to produce the token.

We partitioned all of the secure functions to operate on the secure coprocessor, which is accessible to the insecure processor only through an instruction set. We also partitioned long-term storage of secure data at this point; our device requires secure storage of the TEMPLATE and key K. The other secure functions are in bold text in Figure 3.

Secure instruction set. To protect the secure functions from software bypass attacks and their variants, we construct a secure instruction set. Recall that a software bypass attack skips over the biometric functions and immediately loads the key K to begin encryption.

Consider the control flow diagram in Figure 6 describing an (artificially simplified) secure instruction

Device(I)→Server	ID
Server	Load corresponding K
	Generate RAND, RANDT, SK = E(K, RAND)
Server→Device(I)	RAND | RANDT
Device(I)	Obtain RAW IMAGE from user
	Perform FEATURE extraction algorithm to obtain minutiae
Device(I)→Device(S)	INS = $DO_MATCHING, D_IN = minutiae
Device(S)	**Load TEMPLATE**
	Perform MATCH algorithm of minutiae versus TEMPLATE
	If MATCH, set MATCH_FLAG = 1; Else, set MATCH_FLAG = 0
Device(S)→Device(I)	D_OUT = READY_FOR_CRYPTO
Device(I)→Device(S)	INS = $DO_CRYPTO, D_IN = RAND | RANDT
Device(S)	**If MATCH_FLAG == 1, load K and generate H(K, TEMPLATE)**
	Else, set K = 0000 and set H(K, TEMPLATE) = 0000
	Generate SK = E(K, RAND)
	Generate TOKEN = E(SK, RANDT | H(K, TEMPLATE))
	Reset MATCH_FLAG = 0
Device(S)→Device(I)	D_OUT = TOKEN
Device(I)→Server	TOKEN
Server	Decrypt TOKEN and determine result
Server→Device(I)	TRANSACTION RESULT

Figure 7. Modified verification protocol. Intermodule communication between the secure and insecure modules (Device(I) and Device(S)) occurs over the secure-to-insecure bus structure. (Bold type indicates the secure functions.)

set. The insecure processor accesses the coprocessor via two instructions: `$DO_MATCHING` and `$DO_CRYPTO`. The `$DO_MATCHING` instruction loads the stored template, matches it against the candidate minutiae, and sets a match flag internal to the coprocessor. The secure coprocessor doesn't send the match flag result back to the insecure processor, but rather a READY_FOR_CRYPTO signal indicating that the matching function has completed. (To prevent a timing analysis attack, we ensure that the coprocessor produces the READY_FOR_CRYPTO signal after a constant number of cycles, regardless of whether the match is positive or negative.) At the `$DO_CRYPTO` instruction, the coprocessor loads the internal match flag and performs cryptographic steps to produce the token, which is returned to the insecure processor and forwarded to the server. The secure coprocessor resets the match flag as it produces the token.

This secure instruction set thwarts the software bypass attack. The insecure processor has no way to set the match flag high other than through biometric processing. In addition, neither module leaks sensitive information onto the secure-to-insecure bus, ensuring secure intermodule communication at the software level. The only information sent on the bus is the READY_FOR_CRYPTO response and the token, which the insecure processor can't decrypt because it doesn't have the key K. Hence, the construction of a secure instruction set demonstrates how we can fix a security hole created at the protocol level by addressing the issue at the architecture level.

Figure 7 shows the new protocol with architectural enhancements. Device(I) and Device(S) indicate the insecure and secure modules. Intermodule communication occurs over the secure-to-insecure bus structure. We mapped the indented functions in bold type onto the secure module and executed the other functions on the insecure device or the server.

Microarchitecture level: Hardware design

We designed and simulated the architecture's hardware implementation at the microarchitecture level. This involved designing the hardware for the insecure and secure modules and their interfaces. We used the Leon processor with a configurable Amba bus structure for the insecure module and designed a memory-mapped interface on the Amba peripheral bus to communicate with the secure module.

The secure module is a custom-designed secure coprocessor (see Figure 8). The coprocessor consists of a top-level controller, a cryptographic engine, and a matching engine (with template storage). The coprocessor includes two categories of buses: public and private. The public buses are the coprocessor's interface to the outside world (that is, the secure-to-insecure bus structure) and are thus insecure. The private buses are secure buses internal to the coprocessor, such as those between the matching and cryptographic submodules. These private buses ensure that sensitive data remain local to the coprocessor and aren't directly accessible to the insecure module. The top-level controller monitors for illegal and out-of-sequence instructions. Hence, the insecure processor can't access the coprocessor's internal data; it can access the coprocessor only using instructions on the public buses, which the top-level controller monitors for foul play.

One challenge at the microarchitecture level is to accurately cosimulate the secure and insecure modules. In our design, this implies a hardware (coprocessor)/software (Leon) cosimulation, including a thorough sequence of valid and invalid instructions to protect against false-instruction attacks. For such a cosimulation, we need a design and simulation environment that can simulate the secure and nonsecure modules together. This is because the boundary between hardware and software is often a weak part of secure systems, simply because they're designed and developed separately (and often by different teams).

In our Gezel environment (http://rijndael.ece.vt.edu/gezel2), we simulate the secure hardware module together with software running on the Leon processor's instruction set simulator (ISS). The protocol has a total of 487 million cycles and has been verified against invalid and out-of-sequence instruction attacks.

Another issue at the microarchitecture level is the formal constraints on the top-level controller—that is, the allowed combinations of instructions. The issue here is one of security versus flexibility. The more possible instructions the coprocessor has, the more flexible and programmable it is; however, the greater number of possible combinations of out-of-sequence attacks weakens security. A restricted instruction set limits the possibilities for instruction attacks, but also reduces flexibility and programmability.

Using a hardware coprocessor instead of a software implementation of cryptographic algorithms greatly reduces the energy required for encryption. For instance, an AES implementation, similar to the one used in our secure coprocessor, can generate more than 10 Gbits per Joule whereas AES in Java on an embedded core generates fewer than 10 Kbits per Joule.[4]

The microarchitecture level implements the design and security features described at the architecture level, but suppose an attacker chooses an out-of-band attack at the physical level. In this case, the architecture and microarchitecture defenses are meaningless. We therefore need new defenses at the circuit level.

Circuit level: Combating electrical side-channel attacks

At the circuit level, the system can be implemented as an integrated circuit with measures such as tamper proofing to defend against physical attacks.

In standard complementary CMOS logic (scCMOS), the building blocks of most modern integrated circuits, the only transition that causes dynamic power dissipation from the power supply is a $0 \rightarrow 1$ output transition. A $1 \rightarrow 0$ transition causes a stored output capacitance to discharge to ground. During a $0 \rightarrow 0$ or $1 \rightarrow 1$ transition, the circuit uses no dynamic power. Paul Kocher has shown that this asymmetry in power demand causes information leakage.[5] We can therefore successfully deduce an encryption circuit's secret key by analyzing its power traces' statistical properties. This differential power analysis (DPA) is unique to embedded systems, which are by definition easily accessible.

To combat a DPA attack at the circuit level, consider a circuit style that has the same dynamic power dissipation regardless of the transition ($0 \rightarrow 1$, $1 \rightarrow 0$, $0 \rightarrow 0$, or $1 \rightarrow 1$). This circuit style, which we call *sense amplifier based logic*,[6] uses elements of both differential and dynamic circuit styles to form a secure circuit. SABL makes the four output events equal by charging the same capacitance at every event. Figure 9a shows a sample SABL logic gate that can be used as a `NAND` or `AND`.

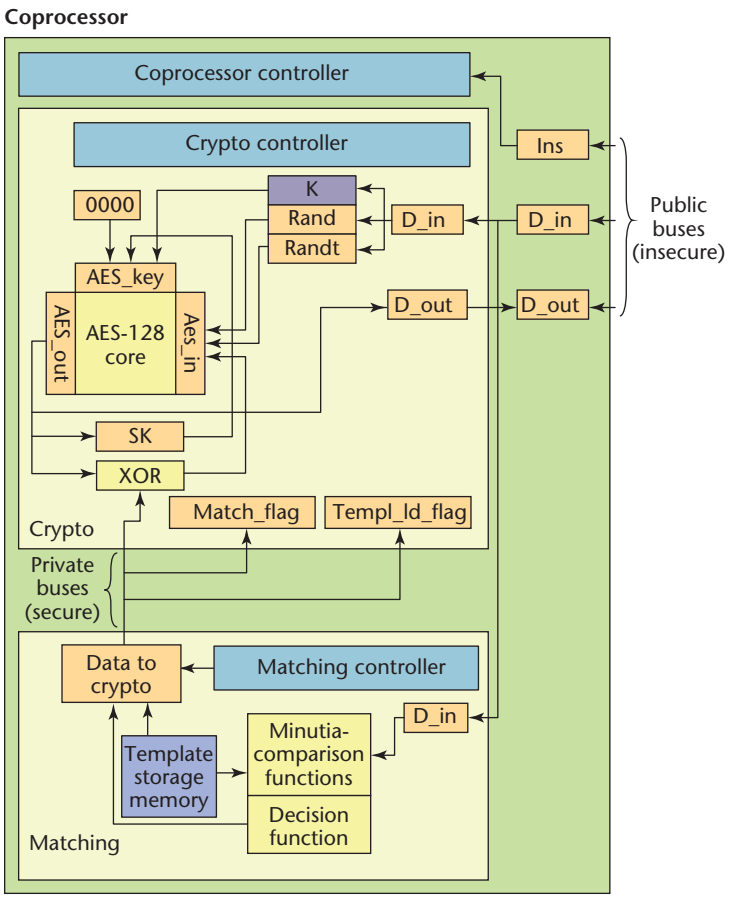


Figure 8. Secure coprocessor microarchitecture. A top-level controller, a cryptographic engine, and a matching engine make up the coprocessor. Public buses serve as the coprocessor's interface to the physical world and, as such, are insecure. Private buses are internal to the coprocessor and prevent the insecure module from accessing sensitive data.

Figure 9b shows the resulting energy consumption per cycle for a typical encryption operation (here, a sample Kasumi S9 box). As the figure shows, a standard CMOS module varies widely in energy dissipated—from 0 petaJoules (pJ) per cycle to 10.42 pJ per cycle—making it relatively easy to perform a DPA attack. SABL dissipates a narrow range of energy—between 11.14 through 11.51 pJ per cycle. This reduces dynamic power variation by 116 times as the cell essentially dissipates the same energy each cycle, thus foiling DPA attacks.

This secure circuit technology entails power and area penalties, however. Although it reduces dynamic power variation, it almost doubles average power consumption (11.32 pJ per cycle versus 5.92 pJ per cycle) and increases cell area 1.8 times. If the entire embedded device used this

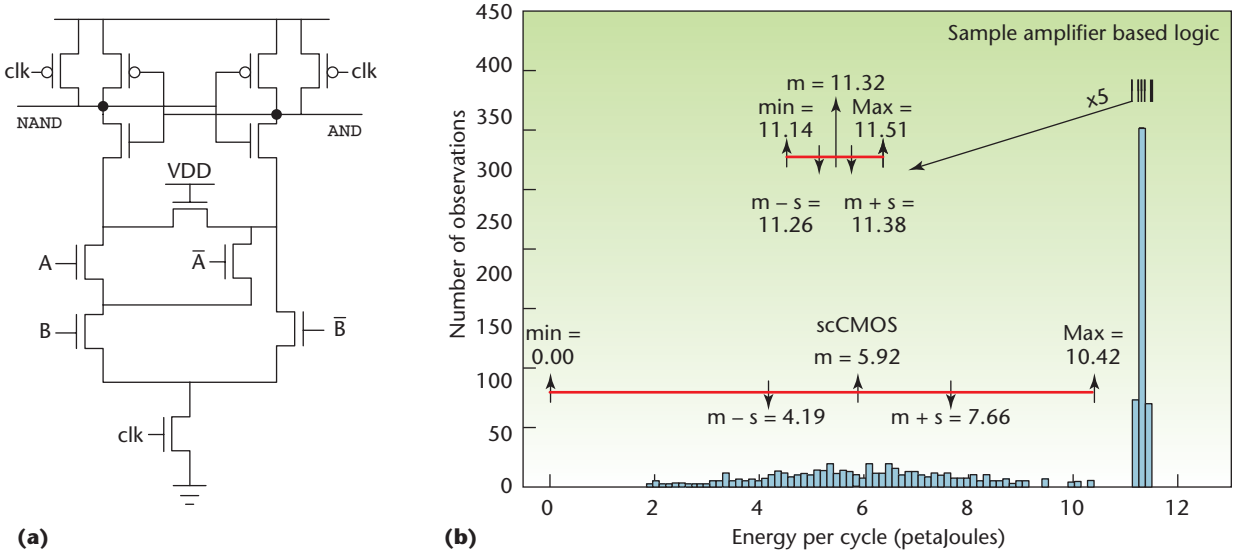


Figure 9. Sense amplifier based logic. (a) Sample SABL gate for use as a NAND or AND gate. (b) SABL energy consumption per cycle in comparison to scCMOS. Because it varies widely in the amount of energy dissipated, a standard CMOS module is vulnerable to differential power analysis attacks. SABL dissipates a far narrower range of energy, so is more robust against DPA attacks.

Table 1. Area comparison of scCMOS and SABL.

LEON PROCESSOR	SECURE COPROCESSOR	TOTAL AREA	AREA INCREASE
1.86 mm^2 (scCMOS)	1.06 mm^2 (scCMOS)	2.92 mm^2	N/A
1.86 mm^2 (scCMOS)	1.91 mm^2 (SABL)	3.77 mm^2	29%
3.35 mm^2 (SABL)	1.91 mm^2 (SABL)	5.26 mm^2	80%

technology, huge power and area penalties would result. However, recall that all the secure functions and sensitive data (which could leak in a DPA attack) reside in the secure module. Hence, security partitioning helps us because SABL needs to protect only the chip's secure portion.

Table 1 gives the estimated area of the device in a 0.18-micrometer (μm) Taiwan Semiconductor Manufacturing Company (TSMC) CMOS technology as 2.92 mm^2 without memory area. If we implemented the entire device in SABL, the total area would become 5.26 mm^2—an 80-percent increase. However, by judiciously partitioning at the architecture and microarchitecture levels, we need to secure only the coprocessor module, resulting in a total area of 3.77 mm^2—a 51-percent area savings over the full-SABL solution. We can make similar remarks for total power consumption. Hence, by combining the partitioning techniques in the architecture and microarchitecture levels with security techniques at the circuit level, we can make the entire device robust without wasting area and power. This again shows that embedded design must account for the interrelationships between security levels.

To address embedded systems' physical accessibility, we can incorporate tamper resistance (not addressed in this article) and resistance to side-channel attacks in the secure module. By using SABL, we achieve side-channel resistance against power-analysis attacks. We can mitigate timing attacks using a matching algorithm whose processing time is the same for a match or reject. Using a secure coprocessor increases performance (reducing cycle counts and overall energy expenditure as explained in the previous section) while minimizing the amount of area and power overhead required to maintain adequate security.

As embedded systems evolve from isolated devices to always-on networked devices, security will become increasingly important, as a hijacked device could wreak havoc on an entire network. Strengthening the security at all levels of the security pyramid will thus be simultaneously more challenging and more critical to society. ❑

Acknowledgments

We acknowledge the support of the US National Science Foundation (CCR-0098361), the Fannie and John Hertz Foundation, and the Design Automation Conference Graduate Scholarship.

Related work and design alternatives

A great deal of recent work has focused on security design alternatives and secure platforms for embedded systems. One approach to PC and embedded security is *trusted computing*, which is spearheaded by the Trusted Computing Group (www.trustedcomputinggroup.org). The TCG specifications localize a system's security components into a *trusted platform module*. The TPM is a hardware module used for secure key storage and generation, digital signature authentication, and attestation, depending on what hardware and software is present on the system. TCG members are preparing protection mechanisms for next-generation PCs (Microsoft's Next-Generation Secure Computing Base and Intel's LaGrande, for example) and are moving toward embedded platforms.

The embedded processor community has also presented some design alternatives to achieve security. ARM (www.arm.com) has recently developed the TrustZone security architecture (www.arm.com/products/esd/trustzone_home.html), which lets users designate system modules and data as secure or nonsecure via a security bit. A processor operating mode controls the security operating state and the transitions between nonsecure and secure domains. MIPS (www.mips.com) has also developed a secure processor core that includes secure memory management, protection against side-channel attacks, and uses an architecture that provides fast software cryptography using the SmartMIPS extensions to the MIPS32 architecture.

At the architecture and microarchitecture levels, IBM has produced a tamper-proof coprocessor that incorporates security at three abstraction levels,[1] which they denote as firmware, software, and hardware. Secure boot is included in the coprocessor, whose hardware components include a 486 processor, data encryption standard engine, modular math engine, and secure memories. One alternative to coprocessors for security is instruction set extensions to speed up cryptography. However, partitioning into secure and insecure instructions/modules might be difficult.

At the circuit or physical level, you can find research in tamper proofing and resistance techniques.[2,3] Another active research area in preventing side-channel attacks is the development of countermeasures such as power-analysis, timing, fault-injection, and electromagnetic-radiation attacks. Secure memory techniques under development include encryption and secure access-control techniques, such as those in the Dallas Semiconductor/Maxim-IC DS2432 (www.dalsemi.com).

References

1. J.G. Dyer et al., "Building the IBM 4758 Secure Coprocessor," *Computer*, vol. 34, no. 10, Oct. 2001, pp. 57–66.
2. R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, John Wiley & Sons, 2001.
3. O. Kommerling and M.G. Kuhn, "Design Principles for Tamper-Resistant Smartcard Processors," *Proc. Usenix Workshop on Smartcard Technology* (Smartcard), Usenix Assoc., 1999, pp. 9–20.

References

1. A. Perrig et al., "SPINS: Security Protocols for Sensor Networks," *Proc. 7th ACM Mobile Computing and Networks* (MobiCom), ACM Press, 2001, pp. 189–199.
2. P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," *Proc. Advances in Cryptology* (Crypto), LNCS 1666, Springer-Verlag, 1999, pp. 388–397.
3. S. Parbhakar, S. Pankanti, and A.K. Jain, "Biometric Recognition: Security and Privacy Concerns," *IEEE Security & Privacy*, vol. 1, no. 2, Mar./Apr. 2003, pp. 33–42.
4. P. Schaumont and I. Verbauwhede, "Domain-Specific Co-design for Embedded Security," *Computer*, vol. 36, no. 4, Apr. 2003, pp. 68–74.
5. P. Kocher et al., "Security as a New Dimension in Embedded System Design," *Proc. Design Automation Conference* (DAC), ACM Press, 2004, pp. 753–760.
6. K. Tiri, M. Akmal, and I. Verbauwhede, "A Dynamic and Differential CMOS Logic with Signal-Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards," *Proc. 28th European Solid-State Circuits Conf.* (ESSCIRC), IEEE CS Press, 2002, pp. 403–406.

David Hwang *is a design engineer with KeyEye Communications. His research interests include system architectures and VLSI implementations of secure embedded systems, as well as VLSI DSP architectures for digital communications. Hwang has a PhD in electrical engineering from the University of California, Los Angeles. He is a Hertz Foundation Fellow. Contact him at dhwang@ee.ucla.edu.*

Patrick Schaumont *is an assistant professor in the electrical and computer engineering department at Virginia Tech. His research interests are design methods and tools for embedded and domain-specific computing systems. Schaumont has an MS in computer science from Rijksuniversiteit Ghent, Belgium, and a PhD in electrical engineering from the University of California, Los Angeles. Contact him at schaum@vt.edu.*

Kris Tiri *is a research scientist with Intel. His research concerns design methodologies and circuit styles to reduce leakage of confidential information in cryptographic designs, thus resisting side-channel attacks. Tiri has an MS from Katholieke Universiteit Leuven, Belgium, and a PhD in electrical engineering from the University of California, Los Angeles. Contact him at tiri@ee.ucla.edu.*

Ingrid Verbauwhede *is an adjunct associate professor in the electrical engineering department at the University of California, Los Angeles and an associate professor in the electrical engineering department at Katholieke Universiteit Leuven, Belgium. Her research interests are in architecture design, VLSI implementation, and design methods for low-power embedded systems. Verbauwhede has a PhD from K.U. Leuven, Belgium. She is a senior member of the IEEE. Contact her at ingrid@ee.ucla.edu.*