SECA: Security-Enhanced Communication Architecture

Joel Coburn, Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar
NEC Laboratories America, Princeton, NJ 08540
jcoburn,sravi,anand,chak@nec-labs.com

ABSTRACT

In this work, we propose and investigate the idea of enhancing a System-on-Chip (SoC) communication architecture (the fabric that integrates system components and carries the communication traffic between them) to facilitate higher security. We observe that a wide range of common security attacks are manifested as abnormalities in the system-level communication traffic. Therefore, the communication architecture, with its global system-level visibility, can be used to detect them. The communication architecture can also effectively react to security attacks by disallowing the offending communication transactions, or by notifying appropriate components of a security violation. We describe the general principles involved in a *security-enhanced communication architecture* (SECA) and show how several security objectives can be encoded in terms of policies that govern the inter-component communication traffic. We detail the implementation of SECA in the context of a popular commercial on-chip bus architecture (the AMBA architecture from ARM) through a combination of a centralized security enforcement module, and enhancements to the bus interfaces of system components. We illustrate how SECA can be used to enhance embedded system security in several application scenarios. A simple instance of SECA has been implemented in a commercial application processor SoC for mobile phones. We provide results of experiments performed to validate the proposed concepts through system-level simulation, and evaluate their overheads through hardware implementation using a commercial design flow.

Categories and Subject Descriptors

B.4.3 [**Hardware**]: Input/Output and Data Communications - *Interconnections (Subsystems) - Interfaces, Topology (e.g. bus, point-to-point)*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems - *Real-time and embedded systems*; D.4.6 [**Software**]: Operating Systems - *Security and Protection - Security Kernels, Access Controls*

General Terms

Design, Security

Keywords

Embedded Systems, Security, Communication Architecture, Bus, System-on-Chip (SoC), Security-Aware Design, Access Control, Intrusion Detection, Attacks, Digital Rights Management (DRM), AMBA Bus

CASES'05, September 24–27, 2005, San Francisco, California, USA.

1. INTRODUCTION

Embedded electronic systems such as cell phones, PDAs, sensors, *etc.*, are routinely used to capture, store, manipulate, and access sensitive personal data, and perform various critical functions. Security is naturally considered by users of such systems as an important requirement for several applications and services. As embedded systems become more complex, extensible through software, and networked, they are susceptible to a wide range of security attacks that have hitherto been the bane of general-purpose computing systems.

While major advances have been achieved in developing theoretical underpinnings (such as cryptographic algorithms) and functional security measures (such as secure communication protocols), they are hardly sufficient to ensure security in practice. Most real security attacks do not directly take on the theoretical strength of cryptographic algorithms, choosing instead to target weaknesses in a system's implementation. This implies that security cannot be added as an afterthought, but must be built-in through careful consideration during various stages of the design process. Attacks that target implementation weaknesses can be classified into physical, side-channel, and software attacks. Of these, software attacks, which are launched by executing malicious software on the target system, or by exploiting vulnerabilities in software that is already installed on the system, are by far the most common, since they are relatively easy to design and deploy. Software attacks are typically addressed by reactive measures such as anti-virus tools, software patches, *etc.* These techniques are useful, but their scope is limited to known viruses, worms, and vulnerabilities. Hence, they are clearly not sufficient, as evidenced by the unabated growth in the instances of successful software attacks – for example, a study by IBM estimates that the number of new viruses increased from 4,551 in 2002 to 28,327 in 2004 [1]. The study also projects that newer security threats are likely to target embedded devices such as mobile phones, personal media players, satellite communication systems in cars, *etc.*

The trends described above make it apparent that the design of embedded systems cannot remain security-agnostic. Since it is often too late or too expensive to address security in the postmortem of an attack, it becomes imperative that security is factored in throughout the system design process. Security features are beginning to appear in general-purpose processors. For example, Intel and AMD's x86 architectures [2], and Transmeta's Efficieon [3] feature a non-executable bit that prevents the execution of code in selected areas of memory, thereby preventing some buffer overflow attacks. The Trusted Computing Group develops open specifications to strengthen the security of computing platforms against software attacks [4]. In the embedded domain, similar trends are reflected in recent developments such as ARM's TrustZone technology [5], and in security-aware SoCs such as TI's OMAP 2420 [6]

and NEC's MP211 [7] mobile application processors, wherein security measures have been incorporated in the design to achieve specific objectives such as the privacy or integrity of sensitive code and data. These technologies and other research efforts are discussed further in Section 6.

Embedded systems are typically designed by assembling various components (one or more processors, memories, application-specific hardware, peripherals, I/O controllers, *etc.*) that are integrated using standard communication architectures. In addition to serving as a fabric for integrating diverse system components, the communication architecture is also responsible for facilitating communications between them. In this work, we propose that the communication architecture, with its system-wide visibility and critical role in enabling system operation, can be exploited to detect and prevent a wide range of software-based security attacks. The communication architecture can answer questions such as (i) which components or programs are accessing a given memory region? (ii) are system-level access control rules (for example, read or write to a peripheral device, or read-once or write-once policies for memory locations) obeyed by a component? (iii) is the present configuration or setting for a peripheral device valid for the accessing component or component context? (iv) is the characteristic behavior of an application (as defined by communication traffic properties) violated due to an intrusion into the system?

We demonstrate that existing communication architectures, with minimal or no changes, allow us to define meaningful security policies that can thwart a wide range of software attacks such as information leakage and corruption, access control violations, and denial-of-service attacks. Security-enhanced communication architectures can be used to monitor and detect violations, block attacks, and provide diagnostic information for triggering suitable response and recovery mechanisms.

The modifications involved in SECA can easily be retrofitted onto an existing communication architecture. They include the addition of a security enforcement module (SEM) that can be programmed to enforce desired security policies, and security enhancements to the interfaces of selected system components. SECA can be used to implement features such as (a) *address-based protection*, which defines and regulates the access control privileges available to selected memory locations and peripherals for a given component or program, (b) *data-based protection*, which provides finer-grained access control by restricting the data values that are assumed by some memory locations and peripheral registers, and (c) *sequence-based protection* that can perform complex checks based on a sequence of transactions executed by the communication architecture.

We describe our implementation of SECA in the context of a popular commercial communication architecture – the AMBA on-chip bus from ARM [8]. Our experiments with NEC's in-house SoC platform demonstrate that SECA can be used to enforce various security policies with minimum overheads. We have also implemented a simple instance of SECA – an address-based protection scheme – in NEC's MP211 mobile phone application SoC.

The rest of this paper is organized as follows. Section 2 provides an introduction to communication architectures by examining the AMBA architecture that is used in this work. Section 3 uses various application scenarios to illustrate software attacks that undermine embedded system security. It also shows how these attacks are manifested in the traffic seen by the communication architecture, making a case for its use in detecting and preventing such attacks. Section 4 then presents SECA, details its internal architecture, and describes how it can be used to achieve various security objectives. Section 5 presents our experimental framework and provides various results. Section 6 positions our work with respect to related work in security-aware embedded system design.

2. PRELIMINARIES

Communication architectures such as ARM's AMBA [8], IBM's CoreConnect [9], SONICS's SMART Interconnect [10], *etc.*, are increasingly becoming a backbone of many embedded systems and System-on-Chips (SoCs). In this section, we use the AMBA bus [8] to introduce basic concepts, and to identify the information available in a communication architecture that can be used to enhance security.

Figure 1 shows an example system built using the AMBA bus. AMBA provides for a bus hierarchy consisting of (i) the Advanced High Performance Bus (AHB) for components that require high communication bandwidth (processors, memory, DMA controllers, *etc.*), and (ii) the Advanced Peripheral Bus (APB) for lower bandwidth peripheral devices.

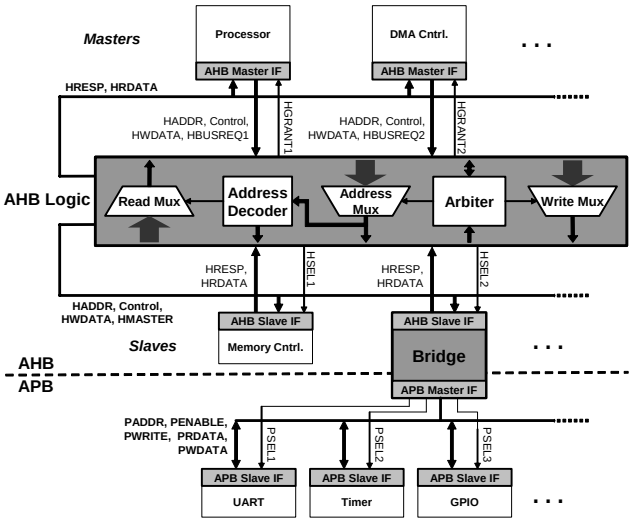


Figure 1: An example system that uses ARM's AMBA on-chip bus architecture

The AHB consists of (i) global interconnect wires for transferring address, control, and data values, and (ii) logic components, including bus interfaces, arbiter, address decoder, address multiplexer, and data multiplexers, which together implement the AHB protocol. The AHB facilitates communication between masters (components that initiate bus transfers) and slaves (components that can respond to transfer requests). All slaves are memory-mapped, meaning that communication transactions are encoded as reads and writes to specific addresses. A centralized arbiter is responsible for regulating bus traffic according to a configurable arbitration scheme. A transaction may be initiated when a master has requested access to the bus and has been granted access by the arbiter. During a transaction, multiplexers route address, control, and write data from the appropriate master to the slaves. The address decoder notifies the desired slave through a slave select signal. Another multiplexer routes the slave response and read data to the masters. The AHB and APB communicate via a bridge, which acts as a slave on the AHB. The bridge is the only master component on the APB.

AMBA SIGNAL	DESCRIPTION
HADDR[31:0]	32-bit system address bus
HRDATA[31:0]	Read data bus
HWDATA[31:0]	Write data bus
HWRITE	Indicates a read or write transfer
HTRANS[1:0]	Indicates transfer type
HSIZE[2:0]	Indicates transfer size
HBURST[2:0]	Indicates if the transfer is part of a burst
HPROT[3:0]	Control information (instr./data, privileged/user, etc.)
HRESP[1:0]	Indicates status of the transfer
HBUSREQx	Master bus request
HGRANTx	Master grant from arbiter
HMASTER[3:0]	Identifies current master
HSELx	Slave select
PADDR[31:0]	APB address bus
PRDATA[31:0]	APB read data bus
PWDATA[31:0]	APB write data bus
PENABLE	APB strobe to time the transfer
PWRITE	APB transfer direction (read/write)
PSELx	APB slave select

Control — AHB — APB

Figure 2: Description of various AMBA signals

The AMBA architecture allows us to easily observe the system-level communication traffic. Figure 2 describes the signals used in the AHB and APB. These signals contain bus transaction information. For example, in the system of Figure 1, suppose the processor experiences a data cache miss and needs to refill the cache line from memory. The processor raises the `HBUSREQ1` signal to inform the arbiter that it needs to use the bus. When the arbiter responds by asserting the `HGRANT1` signal, the `HADDR[31:0]` and other control signals are routed by the address multiplexer to the slaves, based on the `HMASTER[3:0]` signal that identifies the current master (the processor). The control signals indicate transfer properties such as the direction of data transfer (`HWRITE`), the size of the transfer (`HSIZE[2:0]`), the burst mode properties of the transfer (`HBURST[2:0]`), and protection control information (`HPROT[3:0]`) including whether the access is due to an instruction or data fetch, whether the processor is in privileged or user mode, *etc.* The address decoder uses `HADDR[31:0]` to raise the select signal for the appropriate slave, in this case `HSEL1` for the memory controller. After reading from memory, the memory controller returns the data through `HRDATA[31:0]` and indicates the transfer status with `HRESP[1:0]`. This information is returned to the processor through the read multiplexer. Thus, a sequence of address, control, and data values is visible on the bus, which is reflective of the communication transaction currently being performed in the system. In the following sections, we will see how this communication information is effectively used to implement various security policies.

3. MOTIVATION

In this section, we discuss security vulnerabilities and attacks in the context of a popular application – playback of protected multimedia content. We then demonstrate that the communication architecture provides useful information for detecting and preventing such attacks. Based on the examples, we make several key inferences that can be used to design a security-enhanced communication architecture. Before proceeding to the attacks, we first introduce the application under consideration and highlight its security requirements.

3.1 Example Application: Playback of Protected Content

Playback of multimedia content (audio/video) has emerged as a popular revenue-generating application in various consumer electronics appliances. In order to protect the content from unauthorized use, content providers depend on technologies such as digital rights management (DRM) protocols. Figure 3 shows a typical system architecture for portable systems that performs playback of protected audio/video content. The hardware architecture consists of an ARM920T embedded processor [11] (CPU_A), with 16kB instruction and data caches, a crypto-processor (CPU_B) to which cryptographic computations are offloaded by CPU_A, a memory controller, several standard peripherals (timer, UART, GPIO), all of which are connected by the AMBA bus [8]. The user interface consists of an LCD controller peripheral that drives an LCD panel and an audio codec interface that connects to the audio codec, which in turn drives the speaker sub-system.

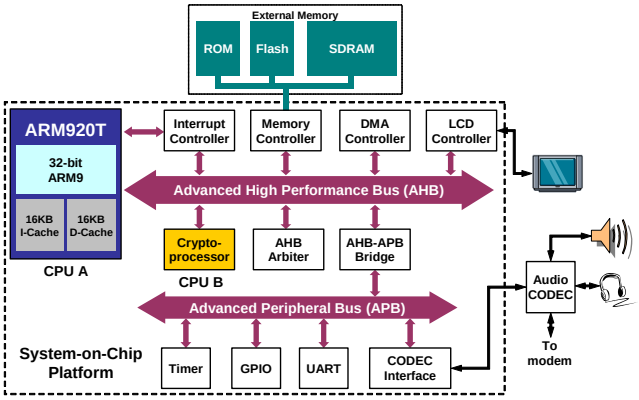


Figure 3: Architecture of a portable audio/video player

We consider the playback of audio and video content protected by the OMA DRM 2.0 protocol [12], using the system shown in Figure 3. For the purpose of the following illustration, we discuss only audio content (the same arguments apply to video playback as well). The audio content is received in encrypted form, along with an encrypted rights object. The rights object contains cryptographic keys for unlocking the content, message authentication codes to ensure that the content has not been tampered with, and permissions and constraints for use of the content on the device. The stored copy of the rights object is encrypted with a key that is device-specific (tied to the specific appliance that requested the content).

The process of playing protected content entails four phases as shown in Figure 4: registration, acquisition, installation, and consumption. For the lifetime of a particular piece of protected content on the system, the registration, acquisition, and installation phases occur only once. Upon their completion, the audio player has registered with a rights issuer, requested and received a protected rights object, verified the integrity and authenticity of the rights object, and unwrapped the security keys contained in it. We focus on the consumption phase of the application, but the reader can find further details of all phases in [12, 13].

The tasks of the content playback application are partitioned between the main processor CPU_A and the crypto-processor CPU_B. In Figure 4, the shaded blocks indicate the decryption and hash operations in the consumption phase that are performed by the crypto-processor. CPU_A interprets the rights object to determine if the content is valid for use. If so, a hash check of the encrypted content is

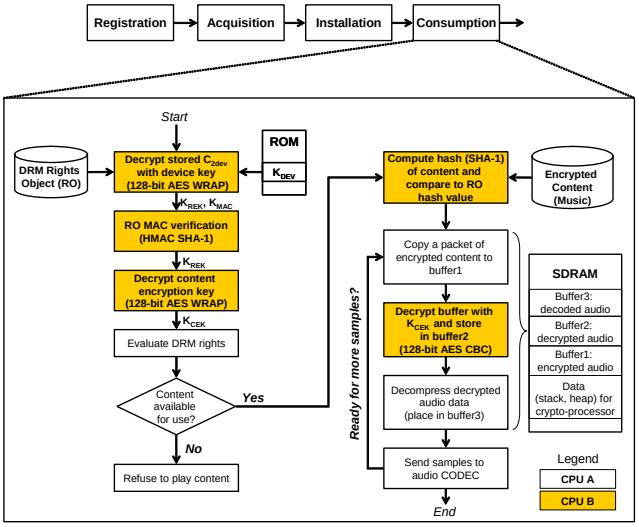


Figure 4: Flowchart for playback of protected content

performed by CPU_B. The computed hash value is compared against a reference value included with the rights object. Each encrypted block of data is decrypted by CPU_B using the content encryption key K_{CEK}. Prior to being encrypted, the content is typically compressed, *e.g.*, using the AAC (Advanced Audio Coding) standard, to minimize data transmission time and storage space. Therefore, each decrypted block is decompressed and then sent to the audio codec, which outputs the audio to the speaker.

3.2 Attack 1: Stealing the Cryptographic Key

We will now examine an example attack on the system presented in Section 3.1. We demonstrate how a stack overflow attack can be used to steal the device key K_{DEV} that is burnt into the system ROM. Knowing K_{DEV}, a user can circumvent the DRM rights object to gain unlimited use of the protected content, including the ability to distribute it in plain form. The stack overflow attack, described in Figure 5, is launched by exploiting a vulnerability in the audio player software – specifically by overflowing a buffer declared on a function's stack frame and consequently overwriting its return address to point to malicious code [14]. The function targeted for the attack is executed after the DRM rights object has been evaluated, when the application prints out user-supplied song information to the screen. The function `printTitle()` shown in Figure 5 uses the library function `strcpy()` to extract the song title from the input string *songInfo*. The function `strcpy()` does not perform bounds checking, so if the title exceeds the size of the array *buffer[]*, then `strcpy()` overwrites the local variable *temp*, the previous function frame pointer (FP), and the function return address. The input string is maliciously crafted to contain attack code and a corrupted return address that points to the initial instruction of the attack code, as shown in Figure 5. The code exploits the application's access rights to obtain a copy of the device key K_{DEV}.

Fortunately, the communication architecture reveals valuable information about system behavior and can expose security violations such as the stack overflow described above. We simulated the example system of Figure 4, including the execution of the audio player software and the stack overflow attack, using NEC's in-house system-level simulation platform. Figure 6 provides a timing diagram from the simulation showing the bus transactions that

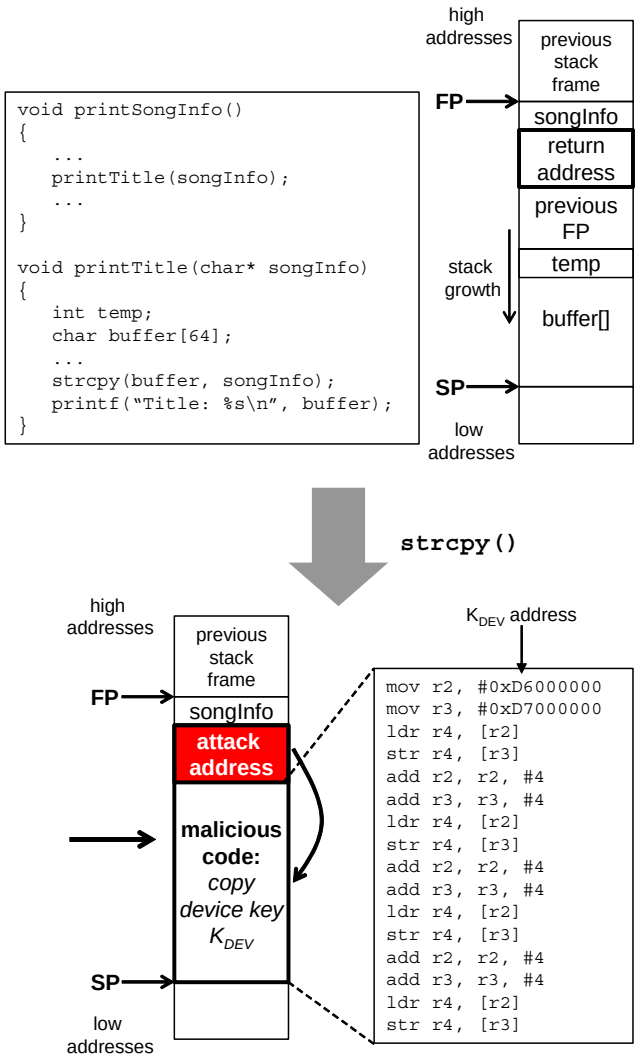


Figure 5: Stack overflow attack that targets the device key K_{DEV}

occur when CPU_A reads the device key from memory. From the state of the communication architecture during the attack, we can clearly detect the stealing of the device key. Access to K_{DEV} is distinguishable because it has a unique address that appears on bus signals `HADDR[31:0]`. Since 128-bit AES encryption was used in the DRM application, the key is 16 bytes long and we can also observe the corresponding four words on the read data signal (`HRDATA[31:0]`). Since the `HWRITE` signal goes low, these transactions are read operations. Finally, the `HMASTER[3:0]` signals show that CPU_A is initiating the read. Functionally, we know from the flowchart in Figure 4 that the crypto-processor CPU_B is the only bus master component that needs to read K_{DEV}. However, the observed bus transactions were initiated by CPU_A, so we can infer that a security violation has occurred.

Thus, we can conclude that the communication architecture can be enhanced to monitor communication traffic to detect such information stealing attacks. It should be noted that by just observing the communication architecture, we required no prior knowledge of the means used to launch the attack. For illustration, we used a simple stack overflow attack but any other attack including heap overflow, format string attacks, *etc.* could have been used.

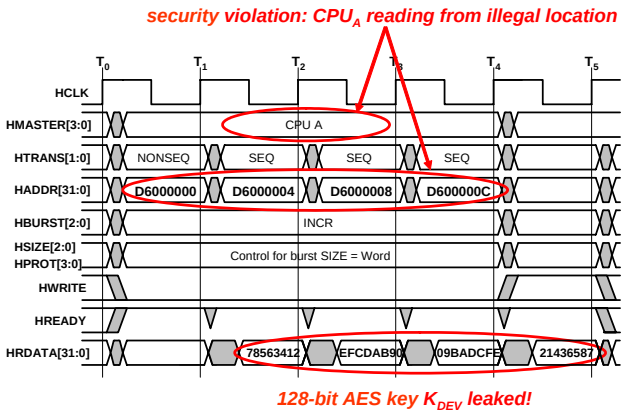


Figure 6: Bus transactions that read the device key K_{DEV} in the presence of a security violation

3.3 Attack 2: Exploiting Peripheral Vulnerabilities

Peripheral devices are beginning to present several security concerns in the end-to-end security chain. For example, the IEEE 1394 interface used in Apple's FireWire and Sony's iLink ports permits client devices to access system memory directly, which can be used to launch various attacks including kernel memory tamper, peripheral data corruption, *etc.* [15].

We now examine the security vulnerabilities in the ARM PrimeCell Advanced Audio CODEC Interface [16], in the context of the SoC in Section 3.1 that is used to play protected content. The CODEC interface is a slave on the APB that communicates with off-chip CODECs through the AC-link protocol. There are four separate channels to support modem, audio, headset, and microphone devices. For our discussion, we assume that channel 1 contains audio data for the speaker, channel 2 carries modem data, channel 3 contains headset data, *etc.* The configuration of the CODEC interface depends on the requirements of the current application. For example, if the DRM rights object prohibits the distribution of content, the data cannot be transmitted to any device other than an audio output device (headset or speaker). Therefore, the audio player is limited to using only channels 1 or 3 to play audio. Any attempt to use other channels can lead to forwarding of content to other media/users, bypassing the protection of the DRM protocol.

Figure 7 shows the memory map for the CODEC interface's control and data registers, along with a DRM-compliant configuration. Based on the restricted usage model of the DRM application, the transmit control registers of channels 2, 3, and 4 (`AACITXCR2-4`) are set to zero, while `AACITXCR1` is set to a value of `0x0000C019` (this setting enables parameters $TX3 = 1$ and $TX4 = 1$ in `AACITXCR1` that allows for the usage of the audio CODEC for PCM left and PCM right audio data output only). Our DRM application sets this configuration prior to playing protected content. However, any application vulnerability, such as a buffer overflow, can be exploited to re-configure the CODEC interface and circumvent the protection mechanism.

The attack code shown in Figure 8 configures the CODEC interface to transmit modem data through channel 2. We can again observe the AMBA bus signals to detect this violation. From the `HMASTER[3:0]` signals, we see that the main processor CPU_A has initiated the data transfer. The address of `AACITXCR2`, the

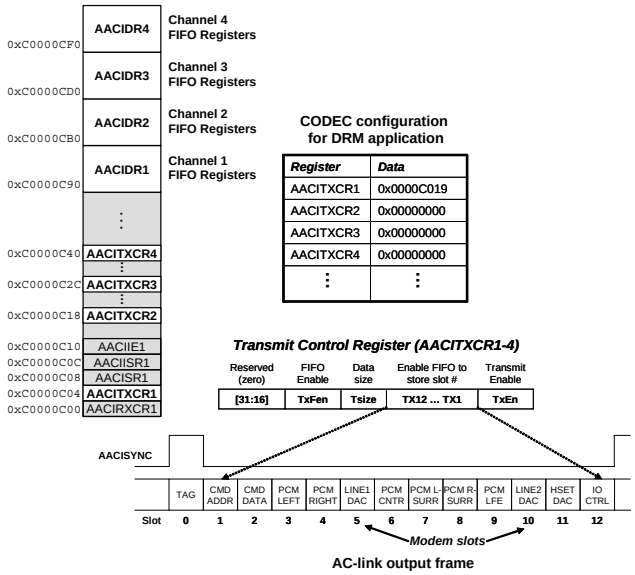


Figure 7: CODEC interface memory map with a configuration for the DRM application

transmit control register for channel 2, appears on `HADDR[31:0]`. The `HWRITE` signal goes high indicating that the transaction is a write. One cycle later, the configuration data is visible on `HWDATA[31:0]`, where it is apparent that a "non-zero" value (`0x00008021`) is written to `AACITXCR2` (a setting of $TxFen = 1$, $TxEn = 1$, and $TX5 = 1$, which results in forwarding of unencrypted, audio samples from the device to the AC-link modem).

```
mov r2, #0xC0000C18 ; address of AACITXCR2
mov r3, #0x8021     ; TxFen=1, TX5=1, TxEn=1
str r3, [r2]        ; set AACITXCR2
```

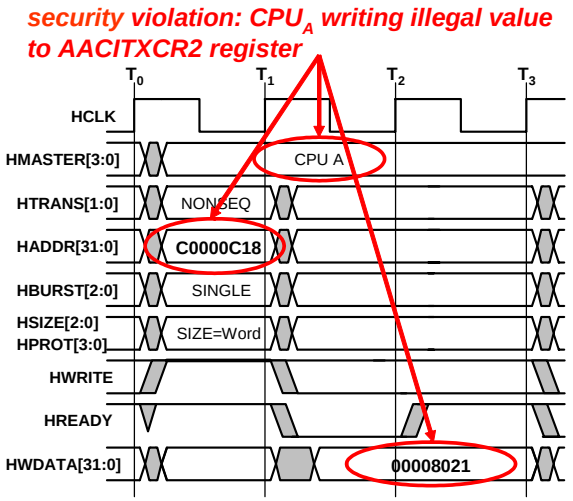


Figure 8: A software attack to enable an AC-link modem to distribute protected content

The above example shows that peripheral vulnerabilities can be used to launch security attacks. However, such attacks can be detected by monitoring a combination of bus signals – address, control, and data, and enforcing appropriate policies that regulate peripheral configuration and usage.

3.4 Summary

The examples presented above show that the communication architecture provides a system-level view of the interactions between various components in an embedded system. Illegal accesses to memory locations, invalid configuration settings to peripherals, *etc.*, are manifested as specific communication transactions, and can hence be detected by observing the communication architecture's address, control, and data lines. As described later in the paper, more complex security attacks can be detected by observing a sequence of transactions. The following section describes how the communication architecture can be enhanced to implement various policies for the detection and prevention of security attacks.

4. PROPOSED ARCHITECTURE

In this section, we describe the design of the proposed security-enhanced communication architecture (SECA). First, we provide an overview of SECA at the system level. Next, we describe in detail the Security Enforcement Module (SEM), the main block for observing communication traffic and enforcing security policies.

4.1 System-level Overview

Figure 9 shows SECA implemented over a typical communication architecture such as AMBA. The architectural enhancements of SECA can be realized as a single centralized module or as a distribution of modules across the topology of the communication architecture. The complexity and heterogeneity of system components dictate the partitioning of SECA logic.

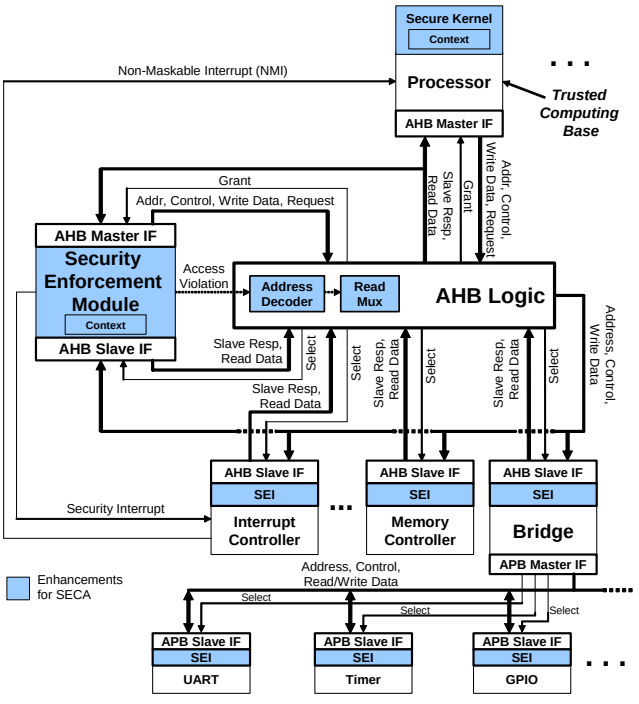


Figure 9: SECA design using the AMBA bus

We present a SECA configuration consisting of a single Security Enforcement Module (SEM) and a Security Enforcement Interface (SEI) for each slave device. The SEM is a plug-in hardware block responsible for monitoring system communication and enforcing programmed security policies including inter-component access control and basic intrusion detection (to be discussed in Section 4.2). The SEM appears as both a master and slave on the AHB. Through its AHB master interface, the SEM can configure the slave SEIs and generate security status messages when violations are detected. Through the AHB slave interface, the SEM is programmed with security configurations for multiple contexts. The context can range from coarse-grained demarcations of execution such as trusted or untrusted, or be as fine-grained as the application (process) ID. At any point in time, the *Context* register of the SEM determines which security policy is enforced. When a security violation occurs, the SEM generates an interrupt that appears on the non-maskable interrupt line of the processor.

The SEI helps the SEM filter the values that can reach the data and control registers of a peripheral, in order to keep the peripheral in a state that corresponds to the current execution context. Note that in addition to the APB devices themselves, the AHB-APB bridge is enhanced with a SEI. Depending on the complexity of the APB slaves, some security policies may be incorporated into the bridge.

To make SECA a complete security solution, we need an established trusted computing base (TCB) [17], consisting of a secure kernel running on the main processor. The TCB allows us to safely program the SEM, update the *Context* register, and respond to security violations. When a context switch occurs, the secure kernel must send the new *Context* to the SEM. It is crucial that this transaction cannot be falsified, because it guarantees application authenticity. SECA operates in three modes throughout the lifetime of an application: *program*, *monitor*, and *response*. Program mode involves transferring security configuration data from the TCB to the SEM, and the SEM in turn configures the SEIs. In monitor mode, the SEM samples each bus transaction and checks for security violations according to the programmed security policies and the current *Context* value. When a security violation occurs, the SEM notifies the processor with a non-maskable interrupt, which is vectored to a response interrupt service routine (ISR) within the secure kernel. The security status data is written to a buffer in memory to be read by the response ISR. In addition to responding to security violations with a protected ISR, the AHB logic can be enhanced to block bus transfers when an illegal access is attempted. The hardware components in the AHB logic that are modified to prevent access violations are indicated by the shaded parts of the AHB logic in Figure 9.

4.2 Security Enforcement Module (SEM)

The Security Enforcement Module (SEM), shown in Figure 10, is the central architectural component responsible for monitoring communications. To enforce various security policies, the SEM includes three security modules: *Address-based Protection Unit* (APU), *Data-based Protection Unit* (DPU), and *Sequence-based Protection Unit* (SPU). The SEM also contains transaction assembly logic to buffer each transaction and a central controller to manage the security modules and interface with other AHB devices.

4.2.1 Address-based Protection Unit (APU)

The APU enforces access control rules that specify how a component can access a device (read-only, write-only, read-write, and not accessible) while in a particular context. The APU uses a look-up table where each entry contains permissions for a region in the address space. We have chosen a simple 2-bit encoding (a read bit and a write bit) for the permissions: 00 is not accessible, 01 is read-only, 10 is write-only, and 11 is read-write. Each entry in the table is indexed by the input signal *APU_Key* from Figure 10, which is

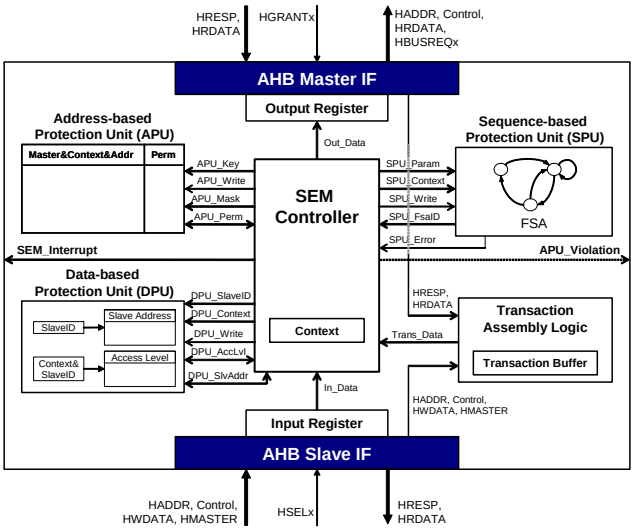


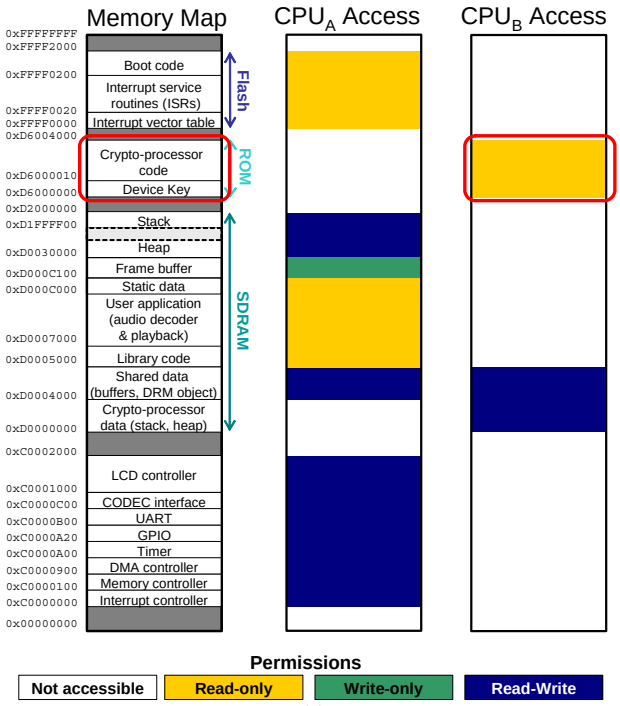
Figure 10: Block diagram of the Security Enforcement Module (SEM)

the concatenation of the AHB signal `HMASTER[3:0]`, the *Context*, and the `HADDR[31:0]` signal. An entry can be programmed through *APU_Key*, *APU_Mask*, and *APU_Perm* inputs when *APU_Write* is high. For efficiency, the look-up table does not contain entries for the entire address space, but instead contains entries only for regions that are accessible (readable, writeable, or both). Therefore, any *APU_Key* that cannot be found in the table indicates that the address is not accessible (00 permission value) by the requesting bus master in the current context. The APU signal *APU_Perm* returns the permissions for the attempted access to the SEM controller when *APU_Write* is low.

Figure 11(a) illustrates the memory protection regions for the DRM application presented in Section 3. The protection regions for CPU_A and CPU_B isolate the data and code sections of the processors from one another. Recall the stack smashing attack of Section 3.2 that was used to access the device key. Now the device key is protected because CPU_A does not have permission to access the key data stored at address `0xD6000000` and CPU_B has read-only access to this location.

Figure 11(b) shows the look-up table entries for safe execution of the DRM application. Each entry defines a region of memory, which is determined by a stored key (first column), a mask value (second column), and the corresponding permissions (third column). The mask specifies the bits of the search key that are don't cares. The search key includes four bits for the master component, four bits for the *Context*, and 32 bits for the memory address. In this example application, CPU_A is master 0 and CPU_B is master 1, and the TCB has assigned $Context = 0$. Consider the last entry in the table (highlighted in the figure) where the stored key is `0x10D6000000`, the mask is `0x0000003FFF`, and the permission is `01`. Based on the stored key, the start address for the memory region is `0xD6000000`. A bitwise OR of the start address and the mask gives an end address of `0xD6003FFF`. Since the permission is `01`, CPU_B has read-only access to this address range. CPU_A is not allowed access to this memory region because there is no corresponding entry in the table.

In hardware, the look-up table is implemented as a conventional ternary content addressable memory (TCAM), which is frequently used for address lookup in Internet routers [18]. A TCAM cell stores a 0, 1, or X (don't care), hence a single TCAM entry with



(a)

Master & Context & Addr	*Mask*	*Perm*
0x00C0000000	0x0000001FFF	11
0x00D0004000	0x0000000FFF	11
0x00D0005000	0x0000000FFF	01
0x00D0006000	0x0000001FFF	01
0x00D0008000	0x0000003FFF	01
0x00D000C000	0x00000000FF	01
0x00D000C100	0x00000000FF	10
0x00D000C200	0x00000001FF	10
0x00D000C400	0x00000003FF	10
0x00D000C800	0x00000007FF	10
0x00D000E000	0x0000001FFF	10
0x00D0010000	0x000000FFFF	10
0x00D0020000	0x000000FFFF	10
0x00D0030000	0x000000FFFF	11
0x00D0040000	0x000003FFFF	11
0x00D0080000	0x000007FFFF	11
0x00D0100000	0x00000FFFFF	11
0x00D0200000	0x00001FFFFF	11
0x00D0400000	0x00003FFFFF	11
0x00D0800000	0x00007FFFFF	11
0x00D1000000	0x0000FFFFFF	11
0x00FFFF0000	0x0000001FFF	01
0x10D0000000	0x0000003FFF	11
0x10D0004000	0x0000000FFF	11
0x10D6000000	0x0000003FFF	01

(b)

Figure 11: (a) Memory protection regions for CPU_A and CPU_B running the DRM application, and (b) APU look-up table entries

don't cares may represent multiple memory addresses. In order to simplify the look-up table design, we require that an entry contains a varying number of significant address bits followed by don't cares for the consecutive low-order bits [19]. Consequently, the address range of an entry is limited to a block of $2^{\#\,don't\,care\,bits}$ locations[1]. Assuming the programmer adheres to this addressing scheme, we eliminate the need for complex decision logic to handle multiple matches. Each TCAM entry indexes a location in a RAM which holds the permissions for that memory region.

The described look-up table is essentially a fully-associative cache of memory protection regions. The number of entries per application and bus master component is not fixed. Each entry contains a valid bit (not shown in the figure) indicating whether or not the entry is currently being used by an application. When an application terminates or is killed, the SEM controller invalidates all of the application's protection region entries. During the programming phase, each new memory region is written to a vacant (invalid) TCAM entry and the corresponding permission value is written to RAM.

4.2.2 Data-based Protection Unit (DPU)

The DPU is responsible for configuring the SEI at each peripheral for data-based protection (refer to Figure 10). In the DPU, there is a memory to store access level values – an access level represents a set of valid operations for the device in a particular execution context. The number of access level bits is scalable, but we choose to limit it to 4 bits, giving a peripheral 16 potential operating modes. There is another memory to store the address of each peripheral's configuration register. The SEI of each peripheral contains a configuration register that stores the access level of the currently executing context. The *DPU_SlaveID* input is used to look up the configuration register address, which appears on the *DPU_SlvAddr* lines. The inputs *DPU_SlaveID* and *DPU_Context* are concatenated to index the access level which is sent to the SEM controller through the *DPU_AccLvl* signal. The SEM controller initiates a bus transaction to write *DPU_AccLvl* to the configuration register at *DPU_SlvAddr*. The DPU can be programmed by setting *DPU_Write* high and providing values on the *DPU_SlvAddr* and *DPU_AccLvl* lines.

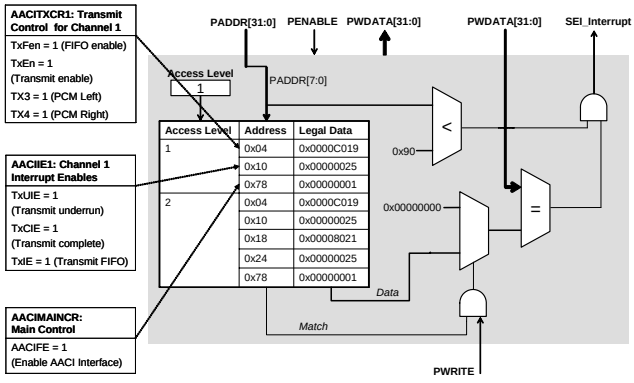


Figure 12: SEI for CODEC interface

The SEI that accompanies each slave device is responsible for enforcing data-based protection. Figure 12 illustrates the SEI for the CODEC interface. A look-up table holds the valid peripheral configuration data that is indexed by the access level and register address. There are three access levels depicted in this security model:

- *Level 0*: Access level 0 is implicit and does not need a look-up table entry. If an application operates at this level, it may only write a disabling value to the control registers – the peripheral is essentially frozen and cannot be put in an operational mode. For the CODEC interface in Figure 12, a value of zero disables the features specified by any control register. Any application which must explicitly turn off the CODEC is configured with level 0 access. However, the APU can keep the application from modifying the current peripheral configuration if required.

- *Level 1*: The CODEC interface is configured for the DRM application in which one channel is used for audio output. This access level can also be used for any other application that involves simple audio playback. In Figure 12, we show three registers that must be set correctly to permit use of the CODEC interface. The transmit control register `AACITXCR1` is configured to enable AC-link output frames, enable the data FIFO, and map the data to the PCM Left and PCM Right slots of the output frame. Transmit interrupts for channel 1 are enabled in the `AACIIE1` register. The interface enable bit of the main control register `AACIMAINCR` is raised high to turn the CODEC interface on.

- *Level 2*: This level is available for applications that need to be able to output both audio and modem data from the CODEC. Besides the control registers that configure channel one for audio output, the transmit control register `AACITXCR2` (address 0x18) and the interrupt enable register `AACIIE2` (address 0x24) for channel two have to be set correctly.

Any control register not defined in the look-up table is inoperable from the current access level. Notice that the control logic includes an address comparator to determine if the intended access is to a control register or to a data register. The channel data FIFOs occupy the addresses above 0x90, so the *SEI_Interrupt* is activated only when the address is below this threshold.

The implementation of the SEI depends on the complexity of the peripheral interface. Simple peripherals with few operating modes need only a handful of registers to hold the legal data values. If the device has many control registers and multiple access levels, the SEI requires a look-up table as shown in Figure 12, and the look-up table is implemented as a CAM.

4.2.3 Sequence-based Protection Unit (SPU)

Sequence-based protection relies on the fact that a sequence of bus transactions can be used to define a signature of expected behavior or an attack. Logically, such behaviors can be specified as finite-state automata (FSA), also termed *security automata* in some scenarios [20]. The SPU can be used to implement various application-specific security policies based on the execution context. At present, we consider the SPU to be built as a static configuration for simplicity and to minimize overhead. The security automata parameters are configurable at run-time, but the security automata themselves are fixed during the design phase. The input *SPU_Param* is used to initialize the FSAs based on the current *SPU_Context*. When an error is detected by an FSA, the SPU

[1] In Figure 11(a), there are eight unique memory protection regions, which, as a consequence of our addressing scheme, map to the 25 TCAM entries shown in Figure 11(b). We believe the increase in the number of entries is more area-efficient than alternative methods.

raises the *SPU_Error* flag and returns the identification of the FSA through the *SPU_FsaID* output. In the rest of this section, we illustrate the applicability of FSA as a part of the SEM hardware to enforce specific security policies.

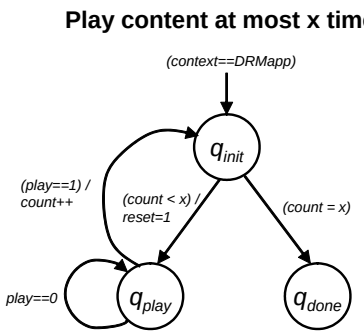


Figure 13: Security automaton to enforce "play at most x times" policy

We consider the problem of enforcing the DRM application security policy "play content at most *x* times," and we address it by providing two security automata. The first automaton (Figure 13) monitors and enforces that the content is played upto *x* times, while the second automaton recognizes when content has been played once and signals the first automaton (flag *play*). The maximum number of allowed plays, i.e. the value of *x*, is given by the DRM rights object. Through a non-volatile, memory-mapped register, the application reads back the number of plays used *count* to determine if a play request is valid. If the application attempts to playback content when *count* has reached *x*, then a policy violation is detected and notified to the processor.

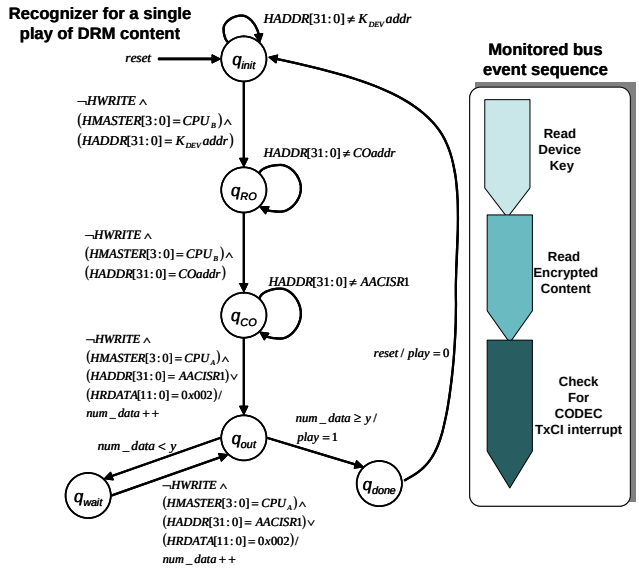


Figure 14: Security automaton to detect playing of DRM content

The automaton in Figure 14 generates the *play* input for the first automaton, if the correct sequence of bus events occur. The first step of the sequence is for CPU_B to read the device key, indicated to the automaton by the parameter $K_{DEV}addr$. Next, the automaton waits in the q_{RO} state to signify that the rights object is being processed. When CPU_B reads the first address of the encrypted content, we enter state q_{CO} to show that the content is being read. The automaton compares the address seen on the bus with the address associated with the encrypted content (parameter *COaddr*). The last step in the sequence is to count the number of audio samples (*num_data*) output to the CODEC and compare it with the parameter *y*, which equals a threshold specified in the DRM rights object. When CPU_A reads the interrupt status register `AACISR1` for CODEC channel one, we check the read data to see if a transmit complete interrupt (TxCI) has occurred. If so, the automaton transitions to state q_{out} and increments the *num_data* variable. Until the next interrupt occurs, the automaton remains in the q_{wait} state. Once $num\_data \geq y$, a "play" of the content is assumed to have occurred.

5. EXPERIMENTAL RESULTS

In this section, we evaluate SECA through simulation and actual hardware implementation using two case studies: (i) secure execution of a DRM application running on a simulated SoC, and (ii) code/data protection of cryptographic firmware in NEC's MP211 mobile phone application chip. We then examine various trade-offs associated with the hardware implementation of SECA.

Our experimental framework included NEC's in-house SoC simulation platform configured to model the system shown in Figure 3 that uses the AMBA 2.0 architecture. The ARM920T processor was modeled using an instruction set simulator, while SystemC [21] was used to implement cycle-accurate functional models of the other SoC components and transaction-level models for the AMBA bus. Benchmark applications written in C were cross-compiled with the GNU ARM toolchain [22], and system simulation was performed with the OSCI reference simulator [21]. Simulation was controlled by the GNU Insight debugger, which communicates with the target system through the remote GDB protocol [23].

5.1 Case Study: DRM Application

Our SoC platform was augmented with SECA to protect the DRM application discussed in Section 3. The address-based protection configuration used is as shown in Figure 11. We implemented data-based protection for the CODEC interface as illustrated in Figure 12 and for an LCD controller from ARM [24]. The security automata shown in Figure 13 and Figure 14 were incorporated into the SPU. We now discuss the hardware area overheads and the performance impact (on program execution time) due to the proposed enhancements.

The SECA area overheads can be categorized into those due to the APU, DPU, SPU, general control logic including the SEM controller and transaction assembly logic, and the SEI logic. For the APU and DPU, we estimated the area of the TCAM and CAMs by using the memory models described in [25]. The FSMs and control logic were synthesized into technology-mapped gate-level netlists using Synopsys Design Compiler [26] with NEC's 0.13μm CB-130M CMOS standard cell library. To report area, we use the ARM920T 32-bit RISC processor core with 16KB instruction and data caches running at 250MHz as a base case. The processor occupies an area of $4.7mm^2$ in 0.13μm technology [11], and all area numbers are given relative to this value.

Figure 15(a) displays the area overheads for three SECA configurations: APU only (SECA-1), APU and DPU (SECA-2), and APU, DPU, and SPU (SECA-3). The area overheads are quite low in comparison to the processor area. When SECA is viewed in the context of the entire SoC, the relative area overheads will be

even lower. In Figure 15(b), the area of the *SECA-3* configuration is decomposed into its constituent logic blocks. The area of the protection blocks varies according to the security demands of the application under consideration. The TCAM for the APU (56%) and the CAMs for the DPU (33%) dominate the SECA area because these modules require a large amount of data storage for the DRM application. On the other hand, the SPU occupies only 6% of the total area, while the control logic overhead is negligible.

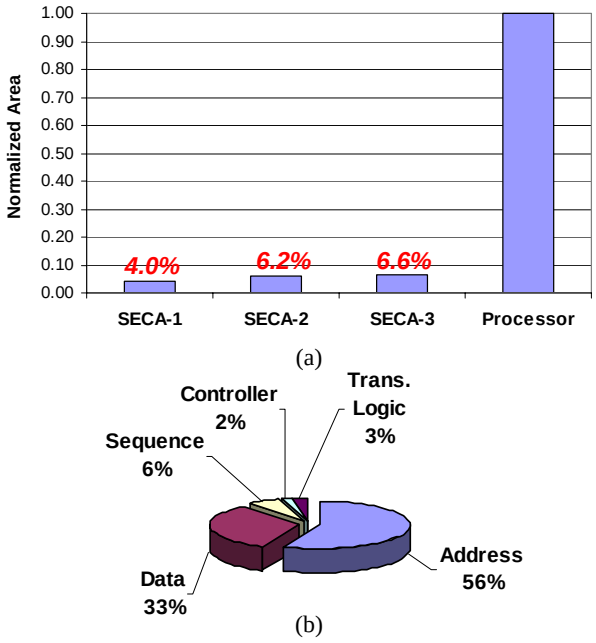


Figure 15: (a) Area overheads for three configurations of SECA, and (b) Breakdown of area by major logic blocks for the SECA-3 configuration

From the perspective of application performance, SECA adds overhead due to the programming needed to configure SECA, and the inherent latency of the SEM hardware. The impact of the programming overhead depends on the normal application execution time and the frequency of context switches. The DRM application consists of standard cryptographic algorithms and the *Mad* MPEG audio decoder from the MiBench benchmark
suite [27]. Both a small input mp3 file (42kB) and a large input mp3 file (382kB) were encrypted with the content encryption key and input to the DRM application running on the simulated SoC. The small file was decrypted and decompressed in 174 million cycles, and the large file was decrypted and decompressed in 1.54 billion cycles. The programming overhead is approximately 300 cycles, and therefore has a negligible effect on application performance. Also, when an application is run multiple times, the programming overhead is incurred just once. In the absolute worst case, context switching between the DRM application and other processes in the system occurs with a frequency of 48kHZ. This frequency represents the best sampling rate of digital audio, and the CODEC generates an interrupt every sample period when it is not capable of buffering samples. At this extreme, the programming overhead due to context switching is 0.03% of the application execution time.

In order to evaluate the latency of the SEM, we synthesized the FSMs for sequence-based protection and the control logic, and the results indicate that the SEM hardware can easily meet the 250MHz frequency target for this SoC.

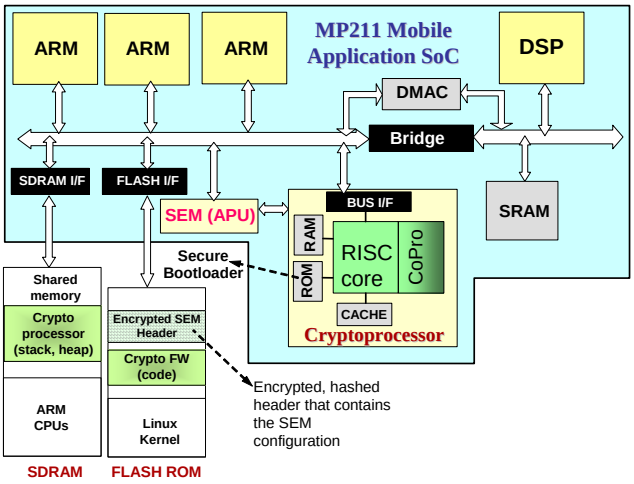


Figure 16: Block diagram of NEC's cell phone application SoC

5.2 Case Study: NEC's Mobile Phone Application SoC

In this section, we describe the implementation of a specific instance of SECA for NEC's MP21x cell phone application SoC [7].

A simplified version of the SoC block diagram is shown in Figure 16 and it includes 3 ARM 926 CPUs and a cryptoprocessor. The cryptoprocessor is a lightweight, RISC processor that has been enhanced with special instructions for accelerating cryptographic algorithms such as RSA, AES, 3DES, SHA-1, MD5, *etc*. The firmware that executes on the cryptoprocessor is stored in an off-chip Flash, and follows an execute-in-place (XIP) model [28]. The cryptoprocessor is associated with an off-chip data space that includes the shaded area in the SDRAM (where the firmware's stack and heap exist). In addition, a shared memory is provided for communication between the ARM CPUs and the cryptoprocessor. The cryptoprocessor is responsible for securely executing portions of security protocols such as SSL, and one of the security objectives is to ensure that the private data space in the SDRAM and the cryptoprocessor firmware in the Flash are protected from tamper by applications running on the ARM CPUs.

We used a simple instance of SECA to achieve this objective. We describe the hardware and software modifications employed to achieve this:

- The SEM includes an APU, which is physically realized by registers used for storing the addresses associated with start and end of the protected memory segments in the Flash and SDRAM, respectively. The SEM is physically configurable only by the cryptoprocessor. Once programmed, these memory segments are configured to allow for exclusive access by the cryptoprocessor. The SEM asserts the non-maskable interrupt line of the cryptoprocessor in the event of a protection violation.

- Programming of the APU registers is performed at boot time as shown in Figure 17. The code responsible for programming the SEM is part of the cryptoprocessor boot code that resides in on-chip boot ROM local to the cryptoprocessor. Since the hardware was fixed early on in the SoC design cycle, it was not possible to fix the start and end addresses of the firmware and data spaces apriori. As a consequence, these addresses were not a part of the boot code burnt into the chip,

but are included in an off-chip header stored in the Flash. The privacy of the addresses is ensured by 3DES encryption with the device key. For ensuring integrity, a golden hash of the addresses is computed and included in the header. During the boot process, the cryptoprocessor decrypts the header, computes a hash and compares against the golden value, and if successful, programs the APU.

During execution, if the SEM interrupts the cryptoprocessor in the event of a violation, the cryptoprocessor invokes the appropriate interrupt service routine (ISR) that is also a part of the boot ROM. The cryptoprocessor executes the ISR, zeroizes the protected data memory, and notifies the ARM CPU for initiating further recovery mechanisms.

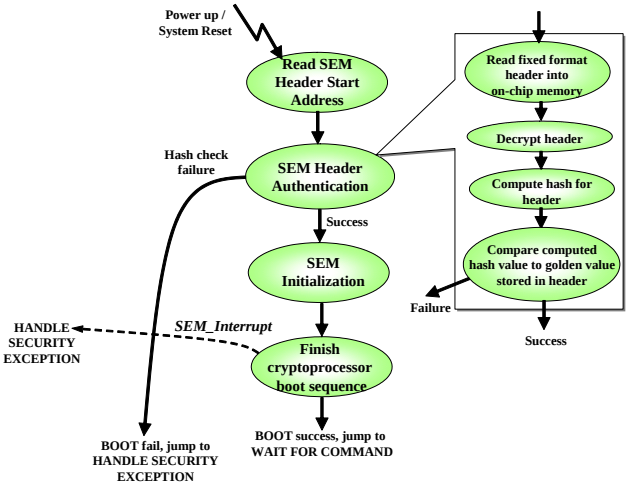


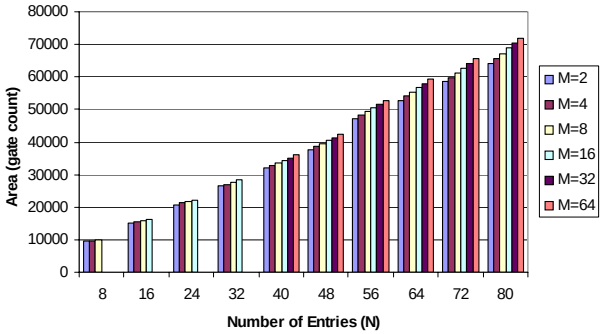
Figure 17: Cryptoprocessor boot sequence including programming the SEM registers

5.3 Design Trade-offs

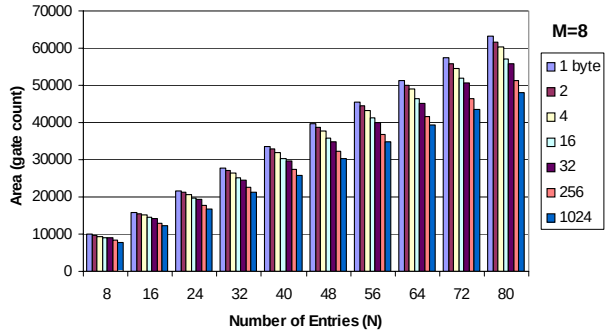
In this section, we discuss a few design trade-offs involved in a SECA implementation. Since our earlier experiments indicated that the performance impact of the proposed architectural enhancements is negligible, we focus on the area overheads due to SECA.

From our experiments, we found that the access control TCAM in the SEM incurs the largest area overhead. Its size is parameterized according to the number of entries (N), the number of contexts (M), and the protection granularity that it must support. In Figure 18(a), we give an equivalent gate count for the TCAM as the number of entries and contexts are varied. We limit the value of M to 64 contexts, and expect a typical SoC to operate with less than 16 contexts. For a small number of entries, we only consider values of M that allow at least one entry per context (*i.e.*, if $N = 16$, then $M \leq 16$). When $N = 80$, the gate count of the TCAM is greater than 60K gates or 10% of the ARM920T area.

The memory protection granularity of TCAM entries can be fixed in order to shorten the width of the search index and reduce area. Figure 18(b) shows the gate count for an increasing number of entries as we scale from protection of a single byte to protection of a 1kB block. For an increasing number of entries, the benefit of a coarse granularity for memory protection becomes more pronounced. Switching from byte resolution to word resolution returns a minimum of 4% area savings. At a 1kB protection resolution, we



(a)



(b)

Figure 18: (a) TCAM area as a function of the number of entries and contexts, and (b) TCAM area scaling with protection granularity varying from 1B to 1kB

see an area savings of well over 20% with respect to byte resolution. Clearly there is a notable area reduction if the TCAM is hand-tuned to match the constraints of the system and a known set of applications.

6. RELATED WORK

In this section, we first briefly outline research in other domains that guided the design of SECA. We then examine the domain of security-aware processors or SoCs and see how they are designed to achieve various security objectives.

The design of SECA was influenced by the concept of firewalls and intrusion detection systems in classical networked systems. Firewalls [29] monitor the information passing through a network and function as a trusted point responsible for auditing and controlling accesses into a system. In some sense, the address-based and data-based protection units of SEM allow it to function as memory and peripheral firewalls. SECA also employs run-time monitoring of inter-component communication to detect deviations from expected system behavior in SoCs. This is conceptually similar to classical network intrusion detection approaches [29, 30], which use a wide range of statistical, rule, and/or model based techniques to detect abnormalities based on usage patterns.

The domain of security-aware SoCs include solutions such as TI's OMAP 2420 [6], which start with limited security objectives such as securing the boot process. This is achieved by including the boot code on-chip, which is then used to verify the integrity of the operating system that is stored off-chip. Other efforts

have attempted to provide security solutions that are applicable during application runtime. For example, recent architectures such as AEGIS [31] and XOM [32] attempt to provide strong process-level isolation to protect a process's sensitive data or code, even when an untrusted operating system and physically insecure memory are used. Isolation is achieved by associating unique tags with each process and encrypting code or data with these tags. The tag-based scheme is implemented by using extra bits throughout the architecture (register files, caches, memory subsystem, communication architecture, *etc.*).

While the security assurance achieved by such designs is high, it comes with significant overheads. Commercially, ARM's TrustZone [5] attempts to provide assurance at a much coarser level by defining two bins for applications – "trusted" and "untrusted". A single tag or bit (called S-bit) is needed to implement this scheme, and again, this extra bit is used throughout the system to achieve its objective. On the software side, the scheme relies on a small, trusted code base to handle entry and exit into, and management of, the trusted domain – the idea is that this code base will be relatively easier/less expensive to secure from various attacks. ARM TrustZone also requires ARM's AMBA bus architecture and peripheral interfaces to be modified to support the S-bit.

Conceptually, ARM's TrustZone and AEGIS/XOM represent two possible extremes in a spectrum of possible security policies. Tags required in both these architectures have specific connotations or usage models: Either a single tag is used to bind an application to its security state (trusted/untrusted), or a unique tag is used to lock or unlock each application's code or data. As seen in this work, tag semantics in SECA are intended to be general-purpose and can be defined according to the desired security model. SECA can easily support the modifications to the communication architecture required in ARM's TrustZone or AEGIS/XOM by using single or multiple access control bits in the SEM and in SECA's bus interface layers.

It is also worth noting that security-aware design of communication architectures is becoming a necessity in the context of overall embedded SoC/device security, and emerging commercial products such as ARM's AMBA 3.0 system bus and SonicsMX SMART Interconnect [10] testify to this trend.

7. CONCLUSIONS

In this work, we presented a framework called SECA for security-aware design of on-chip communication architectures. SECA exploits the fact that a communication architecture can be used to monitor and regulate the interactions between various components in an SoC. SECA offers access control and intrusion detection capabilities by adding a central, regulatory block called SEM and minimal modifications at the bus interfaces. We have validated the ability of the proposed architecture to monitor and detect attacks using NEC's embedded SoC simulator, and evaluated the area and performance overheads of our modifications. Finally, we demonstrated the efficacy of our proposed scheme by implementing a specific instance of SECA in a commercial mobile phone application SoC.

8. REFERENCES

[1] IBM, *Global Business Security Index Report*. `http://www.ibm.com`, 2005.

[2] D. C. Sharma, *Transmeta to add antivirus feature to chips*. CNET Networks Inc. (`http://news.zdnet.com/2100-1009_22-5214194.html?tag=nl`).

[3] M. Kanellos, *AMD, Intel put antivirus tech into chips*. CNET Networks Inc. (`http://news.zdnet.com/2100-1009_22-5137832.html?tag=nl`).

[4] *Trusted Computing Group*. (`https://www.trustedcomputinggroup.org/home`).

[5] R. York, *A New Foundation for CPU Systems Security*. ARM Limited (`http://www.arm.com/armtech/TrustZone?OpenDocument`), 2003.

[6] *OMAP Platform - Overview*. Texas Instruments Inc. (`http://www.ti.com/sc/omap`).

[7] *MP21x mobile application processors*. NEC Electronics Corp. (`http://www.necel.com/en/techhighlights/application_processor/`).

[8] "AMBA 2.0 Specification. (http://www.arm.com/products/solutions).".

[9] *Coreconnect bus architecture*. IBM, `http://www-03.ibm.com/chips/products/coreconnect/`.

[10] *SonicsMX SMART Interconnect*. Sonics Inc. (`www.sonicsinc.com/sonics/products/smx/SonicsMX_Product_Brief.pdf`), 2004.

[11] "ARM920T High Performance and Low Power Platform OS. (http://www.arm.com/products/CPUs/ARM920T.html).".

[12] "DRM Specification V2.0. (http://www.openmobilealliance.org).".

[13] D. Thull and R. Sannino, "Performance Considerations For an Embedded Implemenation of OMA DRM 2," in *Proc. Design Automation & Test Europe (DATE) Conf.*, pp. 46–51, Mar. 2005.

[14] M. Howard and D. LeBlanc, *Writing Secure Code*. Microsoft Press, 2002.

[15] I. Arce, "Bad Peripherals," *IEEE Security & Privacy*, vol. 2, pp. 66–69, Jan./Feb. 2005.

[16] "ARM PrimeCell Advanced Audio CODEC Inteface, ARM Limited. (http://www.arm.com/pdfs/DDI0173B_AACI_trm.pdf).".

[17] DOD, "Trusted Computer System Evaluation Criteria," Tech. Rep. 5200.28-STD, Department of Defense, Dec. 1985.

[18] A. McAuley and P. Francis, "Fast Routing Table Lookup Using CAMs," in *Proc. Joint Conf. of IEEE Computer and Communications Societies*, pp. 1382–1391, Mar. 1993.

[19] E. Witchel, J. Cates, and K. Asanović, "Mondrian Memory Protection," in *Proc. Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 304–316, Oct. 2002.

[20] F. Schneider, "Enforceable Security Policies," *ACM Transactions on Information and System Security*, vol. 3, pp. 30–50, Feb. 2000.

[21] "The Open SystemC Initiative. (http://www.systemc.org).".

[22] "GNU ARM toolchain for Cygwin, Linux, and MacOS. (http://www.gnuarm.com).".

[23] "The Free Software Foundation. (http://www.gnu.org).".

[24] "ARM PrimeCell Color LCD Controller, ARM Limited. (http://www.arm.com/pdfs/DDI0161E_clcdc_pl110_r1p2_trm.pdf).".

[25] J. Mulder, N. Quach, and M. Flynn, "An Area Model for On-Chip Memories and its Application," *IEEE J. Solid-State Circuits*, vol. 26, pp. 98–106, Feb. 1991.

[26] "Synopsys Design Compiler. (http://www.synopsys.com).".

[27] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commerically representative embedded benchmark suite," in *Proc. Annual Wkshp. on Workload Characterization*, pp. 3–14, Dec. 2001.

[28] A. Silberschatz, P. Galvin, and G. Gagne, *Applied Operating System Concepts*. John Wiley and Sons, Inc., 2003.

[29] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.

[30] K. Ilgun, R. A. Kemmerer, and P. A. Porras, "State Transition Analysis: A Rule-Based Intrusion Detection Approach," *IEEE Trans. Software Engineering*, vol. 32, pp. 181–199, March 1995.

[31] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," in *Proc. Intl Conf. Supercomputing (ICS '03)*, pp. 160–171, June 2003.

[32] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 168–177, 2000.