Tamper Resistance Mechanisms for Secure Embedded Systems

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

Abstract

Security is a concern in the design of a wide range of embedded systems. Extensive research has been devoted to the development of cryptographic algorithms that provide the theoretical underpinnings of information security. Functional security mechanisms, such as security protocols, suitably employ these mathematical primitives in order to achieve the desired security objectives. However, functional security mechanisms alone cannot ensure security, since most embedded systems present attackers with an abundance of opportunities to observe or interfere with their implementation, and hence to compromise their theoretical strength.

This paper surveys various tamper or attack techniques, and explains how they can be used to undermine or weaken security functions in embedded systems. Tamper-resistant design refers to the process of designing a system architecture and implementation that is resistant to such attacks. We outline approaches that have been proposed to design tamper-resistant embedded systems, with examples drawn from recent commercial products.

1 Introduction

Digital computing and communications increasingly pervade our lives, our economy, and our nations' critical infrastructure. Almost everything today is electronic, digital and on-line. Security and protection of digital assets is emerging as a discipline of utmost importance. This is especially true for embedded systems, which, due to various constraints, present several unique security challenges [1, 2].

Embedded system security can be broken into a collection of more specific concerns, such as confidentiality, integrity, and availability. Confidentiality is about stopping unauthorized users from accessing sensitive information stored in, or communicated by, the system. The bulk of computer security research has centered around confidentiality, whose roots date as far back as ancient civilizations [3]. Data integrity ensures that data in the embedded system has not been deleted or altered by someone without permission. Software integrity ensures that the programs in the system have not been altered, whether by an error, a malicious user, or a virus. To a large extent, confidentiality is about unauthorized reading of data and programs, while integrity is concerned with unauthorized writing. Availability refers to the embedded system being accessible when needed, and without undue delay, upon demand by an authorized entity. For example, availability is about ensuring that denial of service attacks do not succeed.

Security has long been a concern in computing and communications systems, and substantial research effort has been devoted to addressing it. Cryptographic algorithms, including symmetric ciphers, public-key ciphers, and hash functions, form a set of primitives that can be used as building blocks to construct security mechanisms that target specific objectives [4]. For example, network security protocols, such as IPSec and SSL, combine these primitives in order to achieve authentication between communicating entities, and ensure the confidentiality and integrity of communicated data [5]. We refer to these mechanisms as *functional security mechanisms,* since they only specify what functions are to be performed, irrespective of how these functions are implemented. For example, the specification of a security protocol is usually independent of whether the encryption algorithms are implemented in software running on an embedded processor, or using custom hardware units, and whether the memory used to store intermediate data during these computations is on the same chip as the computing unit or on a separate chip.

The "separation of concerns" between functional security mechanisms and their implementation has enabled (and is, arguably, necessary for) rigorous theoretical analysis and design of cryptosystems and security protocols. However, in the process, various assumptions are made about the implementation of functional security mechanisms. For example, it is typically assumed that the implementations of cryptographic computations are ideal "black-boxes" whose internals can neither be observed nor interfered with by any malicious entity. Aided by these assumptions, the level of security is widely quantified in terms of the mathematical properties of the cryptographic algorithms and their key lengths.

In practice, however, functional security mechanisms alone are *far from being complete security solutions* [6, 7, 8, 9]. It is unrealistic to assume that attackers will attempt to directly take on the computational complexity of breaking the cryptographic primitives employed in security mechanisms. An interesting analogy can be drawn in this regard between strong cryptographic algorithms and a highly secure lock on the front door of a house [7]. Burglars attempting to break into a house will rarely try all combinations necessary to pick such a lock; they may break in through windows, break a doors at its hinges, or rob owners of a key as they are trying to enter the house.

Similarly, almost all known security attacks on embedded systems target weaknesses in the implementation and deployment of functional security mechanisms and their cryptographic algorithms. These weaknesses can allow attackers to completely bypass, or significantly weaken, the theoretical strength of security solutions. Such implementation vulnerabilities abound in embedded systems, due to the following reasons:

- **Operation in untrusted environment:** Many embedded systems have to guarantee secure operation even under the physical possession of untrusted owners. It is easier to design a secure embedded system if we can rely on innate physical security of the device, or assume that parts of the system cannot be physically accessed by malicious entities. However, embedded systems are sometimes required to work under complex trust relationships, where one party wants to put a secure device in the hands of another, with the assurance that the second party cannot modify the innards of the secure device. For example, a bank may want to keep some information on a smart card that is in the hands of its customers, while ensuring that the customers cannot tamper with the device or modify the information it contains. Another common scenario with embedded systems that are portable and have small form factors is loss or theft, which could place the system in the hands of untrusted entities for a significant period of time.
- **Network induced vulnerability:** An increasing number of embedded systems have networking capabilities, which exposes them to many sources of attack. It is no longer necessary to have physical possession of the device in order to break its security mechanisms. Devices with wireless connectivity, or those that connect to the Internet, are the most vulnerable.

- **Downloaded software execution:** The drive to provide richer functionality and increased customizability to the end-users of embedded systems often requires the ability to execute untrusted software (*e.g.*, freeware or third-party software downloaded from the Internet) on them. Software programs (including viruses, worms, and trojan horses) are by far the instruments of choice in launching security attacks. The magnitude of this problem will only worsen with the rapid increase in the software content of embedded systems.
- **Complex design process:** In order to meet stringent design turn-around time and cost constraints, complex embedded systems are being assembled using components from multiple sources spread across corporate boundaries. The responsibility for ensuring system security typically falls upon the manufacturer of the end product that is sold, or upon the entity that provides services based on the end product. However, it may not be possible to pre-validate each system component to ensure security. Furthermore, even if each part of a system is secure in itself, it is known that the composition of parts may expose new vulnerabilities [10]. Due to the lack of suitable design methodologies, modeling and optimization of security during embedded system design is already a poorly understood art [11]; the above factors only serve to exacerbate this problem.

Designing systems that are absolutely tamper-proof is often not possible, primarily due to two reasons: (1) prohibitive costs incurred in putting together a device that can withstand innumerable, often unknown, attacks, and (2) relentless and rapid improvements in technology constantly, which increase the reach and capability of attackers. In response to this reality, the practical approach is to implement *tamper-resistant* embedded systems, which translates to tamper-proof for almost all practical purposes.

In summary, achieving high levels of security requires strong functional security mechanisms that are embodied in tamper-resistant implementations. The design of tamper-resistant implementations requires a strong awareness of the potential implementation weaknesses that can become security flaws, and careful consideration of security during all aspects of the architecture, hardware, and software design processes. In this paper, we first outline the major attack techniques that can threaten the security of an embedded system. Then, we present various countermeasures for the prevention of, detection of, and recovery from, attacks, and discuss their effectiveness in enhancing embedded system security.

2 Attacks on Secure Embedded Systems

Figure 1 shows a broad classification of attacks on embedded systems. At the top level, attacks are classified into three main categories based on their functional objectives.

- *Privacy attacks*: The objective of these attacks is to gain knowledge of sensitive information stored, communicated, or manipulated within an embedded system.
- *Integrity attacks*: These attacks attempt to change data or code associated with an embedded system.
- *Availability attacks*: These attacks disrupt the normal functioning of the system by mis-appropriating system resources so that they are unavailable for normal operation.

A second level of classification of attacks on embedded systems is based on the agents or means used to launch the attacks. These agents are typically grouped into three main categories as shown in Figure 1:

- *Software attacks*, which refer to attacks launched through software agents such as viruses, trojan horses, worms, *etc.*

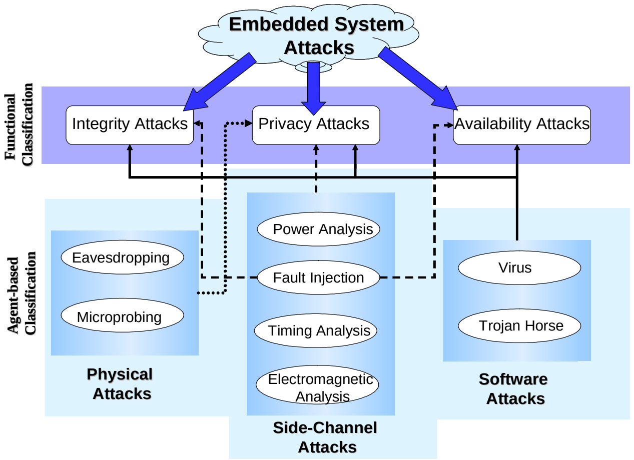

Figure 1: Taxonomy of attacks on embedded systems

- *Physical or Invasive attacks*, which refer to attacks that require physical intrusion into the system at some level (chip, board, or system level).
- *Side-channel attacks*, which refer to attacks that are based on observing properties of the system while it performs cryptographic operations, *e.g.*, execution time, power consumption, or behavior in the presence of faults.

The agents used to launch attacks may either be passive in the sense that they do not interfere in any manner with system execution (*e.g.*, merely probe or observe certain properties), or may actively interfere with the target system's operation. Integrity and availability attacks require interference with the system in some manner, and hence can be launched only through active agents.

It bears mentioning that, although we have classified attacks into various categories for the sake of understanding. In practice, attackers often use a combination of various techniques to achieve their objectives. For example, physical attacks may be used as a pre-cursor to side-channel attacks (removing a chip's packaging before observing the values on global wires within the chip). Our classification is also by no means exhaustive, nor is it intended to be — the ingenuity of attackers who invariably come up with new schemes to break security is arguably the greatest challenge to tamper-resistant design.

2.1 Software Attacks

Software attacks represent a major threat to embedded systems that are capable of downloading and executing application code. Compared to physical and side-channel attacks, software attacks typically require infrastructure that is substantially cheaper and easily available to most hackers, making them a serious immediate challenge to secure embedded system design. These attacks are implemented through malicious agents such as viruses, worms, trojan horses, *etc.*, and can compromise the security of a system from all standpoints – integrity, privacy, and availability.

Malicious software agents mount software attacks by exploiting weaknesses in the end-system architecture [12, 13, 14, 15, 16]. They typically arise due to shortcomings in the software, which can be termed as either *vulnerabilities* or *exposures* [12]. A *vulnerability* allows the attacker to gain direct access to the end-system, while an *exposure* is an entry point that an attacker may indirectly exploit to gain access.

The buffer overflow problem is a common loophole in operating systems and application software, which can be exploited during software attacks [17]. The problem can arise whenever buffers are present with poor bound checks. Buffer bounds may be violated due to incorrect loop bounds, format string attacks, *etc.* Buffer overflows effects can include overwriting stack memory, heaps, and function pointers. The attacker can use buffer overflows to overwrite program addresses stored nearby. This may allow the attacker to transfer control to malicious code, which when executed can have undesirable effects.

A good high-level introduction to the challenges involved in writing secure code can be found in [18].

2.2 Physical and Side-channel attacks

Various physical and side-channel attacks can be launched against an embedded system. Historically, many of these attacks have been orchestrated in the context of low-end embedded systems such as smartcards [19, 20, 21, 22, 23]. However, with these attacks increasingly becoming sophisticated and shown to be deployable against many electronic systems, they are considered a significant challenge to the process of designing secure embedded systems.

Physical Attacks

For an embedded system on a circuit board, physical attacks can be launched by using probes to eavesdrop on inter-component communications. However, for a system-on-chip, sophisticated microprobing techniques become necessary [19, 20]. The first step in such attacks is de-packaging. De-packaging involves removal of the chip package by dissolving the resin covering the silicon using fuming acid. The next step involves layout reconstruction using a systematic combination of microscopy and invasive removal of covering layers. During layout reconstruction, the internals of the chip can be inferred at various granularities. While higher-level architectural structures within the chip such as data and address buses, memory and processor boundaries, *etc.*, can be extracted with little effort, detailed views of lower-level structures such as the instruction decoder and ALU in a processor, ROM cells, *etc.*, can also be obtained. Finally, techniques such as manual microprobing or e-beam microscopy are typically used to observe the values on the buses and interfaces of the components in a de-packaged chip.

Physical attacks at the chip level are relatively har to use because of their expensive infrastructure requirements (relative to other attacks). However, they can be performed once and then used as precursors to the design of successful non-invasive attacks. For example, layout reconstruction is needed before performing electromagnetic radiation monitoring around selected chip areas. Likewise, the knowledge of ROM contents, such as cryptographic routines and control data, can provide an attacker with information that can assist in the design of a suitable non-invasive attack.

Power Analysis Attacks

The power consumption of any hardware circuit (cryptographic ASICs or processors running cryptographic software) is a function of the switching activity at the wires inside it. Since the switching activity (and hence, power consumption) is data dependent, it is not surprising that the key used in a cryptographic algorithm can be inferred from the power consumption statistics gathered over a wide range of input data. These attacks are called power analysis attacks and have been shown to be very effective in breaking embedded systems such as smartcards. Power analysis attacks are categorized into two main classes: Simple Power Analysis (SPA) attacks and Differential Power Analysis (DPA) attacks.

SPA attacks rely on the observation that in some systems, the power profile of cryptographic computations can be directly used to reveal cryptographic information [24]. For example, Figure 2 shows the power consumption profile for an ASIC implementing the DES algorithm. From the profile, one can easily identify the 16 rounds of the DES algorithm. While SPA attacks have been useful in determining higher granularity information such as the cryptographic algorithm used, the cryptographic operations being performed, *etc.*, they require reasonably high resolution to reveal the cryptographic key directly. In practice, SPA attacks have been found be useful in augmenting or simplifying brute-force attacks. For example, it has been shown in [25] that the brute-force search space for a SW DES implementation on an 8-bit processor with 7 Bytes of key data can be reduced to 2^{40} keys from 2^{56} keys with the help of SPA.

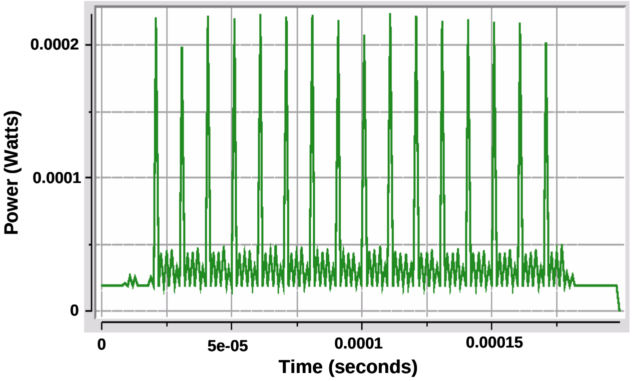

Figure 2: The power consumption profile of a custom hardware implementation of the DES algorithm

DPA attacks [26] employ statistical analysis to infer the cryptographic key from power consumption data. These attacks use the notion of differential traces (difference between traces) to overcome the disadvantages of measurement error and noise associated with SPA techniques. DPA has been shown to be highly robust and effective in extracting keys from several embedded systems, not limited to smartcards [26]. Recent approaches such as [25] enhance the effectiveness of DPA attacks by providing techniques that improve the signal-to-noise ratio. While the initial DPA attacks [25, 26, 27] targeted DES implementations, DPA has also been used to break public-key cryptosystems [28].

Timing Attacks

Timing attacks [29, 30, 31] exploit the observation that the execution times of cryptographic computations are data-dependent, and, hence, can be used to infer the cryptographic key. The variations in execution time can arise from implementation- or architecture-specific properties, such as:

- *Instruction Execution Time Variations*: Software implementations of cryptographic computations (such as the modular exponentiation operation) often invoke the processor's multiply and divide instructions. Since these instructions take a variable number of cycles based on the data inputs in many processors, execution time statistics of the cryptographic algorithm can be collected and analyzed for a wide range of data in order to break the key.
- *Performance optimizations*: The use of performance optimizations in a cryptosystem may introduce execution paths in its implementation that are more sensitive to data statistics than otherwise. For example, timing attacks against implementations of the RSA algorithm that use the Chinese Remainder Theorem (CRT) can expose the factors of the modulus, which, in turn, can be used to easily compute the decryption key.

Fault Injection Attacks

Fault injection attacks rely on varying the external parameters and environmental conditions of a system such as the supply voltage, clock, temperature, radiation, *etc.*, to induce faults in its components. The injected faults can be transient or permanent, and can compromise the security of a system in several ways:

- *Availability Attacks*: Faults can be injected to disrupt the normal functioning of the system. For example, the bus in an embedded system on chip can be made unavailable for performing inter-component communications through permanent faults that set the bus lines to a constant value.
- *Integrity attacks*: These attacks can be used to corrupt the secure or non-secure code or data stored in components such as memories.
- *Privacy attacks*: An interesting example of the use of fault injection attacks to reveal cryptographic keys involves RSA implementations that use the Chinese Remainder Theorem (CRT) optimization [32]. The optimization, intended to enhance the performance of the modular exponentiation operation in RSA, in fact, increases its vulnerability against fault injection attacks. It has been shown in [32] that the RSA modulus can be factored very easily if faults can be introduced to affect the outputs of one of the sub-exponentiations being performed.
- *Pre-cursor attacks*: Fault injection techniques are also useful as a pre-cursor to software attacks. For example, it has been shown in [33] that simple memory faults induced by heat can be exploited by an untrusted program running on a processor to assume complete control of its execution environment.

Electromagnetic Analysis Attacks

Electromagnetic analysis attacks (EMA) have been well documented since the eighties, when it was shown in [34] that electromagnetic radiation from a video display unit can be used to reconstruct its screen contents. Since then, these attacks have only grown in sophistication [35]. The basic premise of many of these attacks is that they attempt to measure the electromagnetic radiation emitted by a device to reveal sensitive information. Successful deployment of these attacks against a single chip would require intimate knowledge of its layout, so as to isolate the region around which electromagnetic radiation measurements must be performed. Like power analysis attacks, two classes of EMA attacks, namely, simple EMA (SEMA) and differential EMA (DEMA) attacks have been proposed [36, 37].

3 Tamper Resistant Design: Countering Security Attacks

In this section, we survey tamper-resistant design techniques that have been proposed to strengthen embedded systems against the various attacks described in the previous section. In order to better understand and compare approaches to tamper-resistant design, we decompose the objective of tamper resistance into more specific, narrower objectives, as shown in Figure 3.

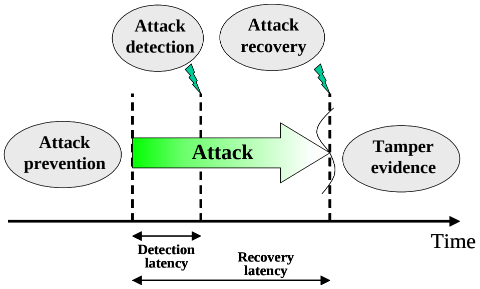

Figure 3: Specific objectives of tamper-resistant design approaches

- *Attack prevention* techniques make it more difficult to initiate an attack on the embedded system. These techniques can include physical protection mechanisms (*e.g.*, packaging), hardware design (*e.g.*, circuit implementations whose timing and power characteristics are data independent), and software design (*e.g.*, software authentication before execution).
- In the event that an attack is launched despite any employed prevention techniques, *attack detection* techniques attempt to detect the attack as soon as possible. The elapsed time interval between the launch of an attack and its detection (the detection latency) represents a period of vulnerability, and needs to be kept as low as possible. An example of attack detection is the run-time detection of illegal memory accesses to secure data from an untrusted software application.
- Once an attack is detected, the embedded system needs to take appropriate action. *Attack recovery* refers to techniques used to ensure that the attack is countered, and that the system returns to secure operation. Attack recovery techniques could include locking up the system and rendering it useless for further operation, zeroing out sensitive data in memory, or displaying a security warning and rebooting the system. The design of attack recovery schemes involves tradeoffs between the level of security and the inconvenience caused to users in the usage of the system after an attack.
- In some cases, it may be desirable to preserve an irrefutable, persistent record of the attack in the embedded system, for inspection at a later time. *Tamper evident* design techniques target this objective. Analogies of physical tamper evident design mechanisms abound: seals that have to be broken, wires that have to be cut, or coatings that have to be removed. In all cases, tamper evidence requires a mechanism that cannot be reversed by malicious entities.

In the rest of this section, we describe design techniques to counter each of the specific categories of attacks described earlier. Classification of these design techniques into attack prevention, detection, recovery, and tamper evidence is left as an exercise to the reader.

3.1 Countermeasures for Software Attacks

Countermeasures for software attacks are typically designed with one or more of the following considerations:

- Ensure privacy and integrity of sensitive code and data during every stage of software execution in an embedded system.
- Determine with certainty that it is safe from a security standpoint to execute a given program.
- Remove security loopholes in software that make the system vulnerable to such attacks.

Most system-level countermeasures attempt to, at least, address the first considerations listed above. A common feature of these countermeasures involves regulating the accesses of various software components (operating system, downloaded code, *etc.*) to different portions of the system (registers, memory regions, security co-processors, *etc.*) during different stages of execution (boot process, normal execution, interrupt mode, *etc.*), through a combination of hardware and software changes. Since an effective countermeasure must allow the system to provide guarantees about the security of the system starting from the powered-on state, most measures define notions of trust or *trust boundaries* (also referred to as *security perimeters*) across the various hardware and software resources. This allows the system to detect infringements of trust boundaries (such as illegal accesses to memory regions) and enforce recovery mechanisms (such as zeroing processor registers and memory regions). Thus, a trust boundary provides a natural and convenient foundation for the system to make judicious decisions about its security (or compromise, thereof).

In the rest of this section, we will focus on understanding individual countermeasures that typically make up a software tamper-resistance strategy – hardware additions, secure bootstrapping, secure OS features, software integrity and safety checks, and methods for finding and fixing security loopholes in software.

Hardware Support

A common approach to implementing tamper-resistance involves the use of a separate secure co-processor module [38, 39, 40], which is dedicated to processing all sensitive information in the system. Any sensitive information that needs to be send out of the secure co-processor is encrypted.

Many embedded system architectures rely on designating and maintaining selected areas of its memory subsystem (volatile or non-volatile, off-chip or on-chip) as secure storage locations. Physical isolation is often used to restrict the access of secure memory areas to trusted system components. When this is not possible, a memory protection mechanism adopted in many embedded SOCs involves the use of bus monitoring hardware that can distinguish between legal and illegal accesses to these locations. For example, the CrypoCell security solution from Discretix [41] features BusWatcher, which performs this function. Ensuring privacy and integrity in the memory hierarchy of a processor is the focus of [42], which employs a hardware secure context manager, new instructions, and hash and encryption units within the processor. The work in [43] describes a model of execute only memory (XOM), and architectural techniques to implement it, using hardware enhancements such as custom instructions and additional fields in cache lines, together with a software virtual machine monitor. Similar ideas were also described earlier in [44].

Recently announced commercial initiatives such as ARM's TrustZone [45], Microsoft's Palladium or NGSCB [46, 47], and Intel's LaGrande [48], *etc.* feature various hardware enhancements for security. Later in this paper, we will examine the hardware enhancements for security proposed in ARM's TrustZone technology (see Section 3.3).

Secure Bootstrapping

One of the early works that explores the notion of a trust boundary is the AEGIS architecture [49] that examines the problem of securing the boot process in the IBM PC architecture. AEGIS provides a hierarchical solution to the problem by exploiting the layered nature of the boot process. Starting from power on, the system can move to the next layer in the boot process if and only if a sequence of integrity checks have been successfully performed on the current layer (and all layers below it). The integrity checks involve computing the hash value of a boot process component and comparing it with a securely stored value. Thus, a trust boundary is progressively expanded prior to handing off the system controls to the operating system in a secure manner.

Operating System (OS) Enhancements

Most security schemes rely on OS modifications in order to provide protection to sensitive code and/or data. For example, Microsoft's NGSCB initiative advocates a secure Nexus mode for Windows that (a) provides strong process isolation, and (b) performs process-level attestation. Process isolation ensures that private resources of one process can be protected from another process, while attestation ensures that code can be authenticated before establishing communication channels between processes and devices. Other OS enhancements for security could include modifications to context switching, exception handling, inter-process communication, and memory management. It is important to note that many of these operating system changes would require (or are in response to) architecture-level modifications (such as memory management system changes) for security [50]. Emerging OS architectures such as [51] claim to offer good flexibility and better isolation than existing solutions under various application scenarios.

Other features offered by a secure OS include the usage of cryptographic file systems (CFSs) to provide secure storage [52, 53]. A CFS operates on the principle that trusted components of the system should encrypt information immediately before sending data to untrusted components. Therefore, a CFS moves the encryption and decryption services from the user level into the operating system itself, thereby protecting sensitive information from application-level vulnerabilities (assuming that the OS is secure).

Software Authentication and Validation

Secure execution of known software in a system requires that it is validated before execution. One of the most common techniques used to validate the integrity of a known piece of software involves computing a hash or checksum of the code (or its critical sections) and verifying it against a pre-computed golden value. More recent techniques such as oblivious hashing [54] also hash the execution trace of a piece of code, thereby verifying its run-time behavior.

In order to run untrusted application code, it is prudent to use techniques that can provide *sandboxes* (restricted environments for code) for execution. This is a feature of many virtual machines including the Java Virtual Machine (JVM). Software mechanisms such as proof-carrying code [55] require an untrusted code supplier to bundle safety proofs with the program executables, so that the system can ascertain that the code will not violate its security policies. This technique is useful, for example, in determining whether a piece of code can be allowed to execute in the kernel's address space. In such a case, the system would require proof that the program will maintain the consistency of the kernel's data structures. This approach requires the compiler to be enhanced for generating such proofs.

Program sheperding [56] is another approach that prevents execution of malicious code by monitoring all control transfers in a program and checking that a given security policy is not violated. Code origin checks, restricted control transfers, and guaranteed sandboxing are used to prevent program vulnerabilities, overwrites of stored program addresses, and execution of malicious code. This technique can be implemented to operate on generic executables by altering run-time execution environments.

Since many of the known reasons for software attacks stem from vulnerabilities in trusted software, software verification engines are becoming increasingly important for detecting errors that make a system prone to attacks. For example, extended static checking, which is useful for finding errors in source code during compile time, has been used to identify security flaws in many programs [57, 58]. Formal verification techniques such as model checking have also been successfully applied to verify implementations of security protocols [59, 60].

3.2 Countermeasures for Physical and Side-channel Attacks

Packaging technologies, physical security for sensitive information through the use of cryptoprocessors, environmental attack protection measures, careful design of the HW/SW implementation to make properties such as timing and power insensitive to the input data, *etc.*, are common ways of countering physical and side-channel attacks. These countermeasures are discussed below.

Physical Attack Protection

Several advanced packaging and attack response techniques have been recommended by the Federal Information Processing Standard (FIPS 140-2) [61]. For example, the standard specifies four increasing levels of physical (as well as other) security requirements that can be satisfied by a secure system. Security Level 1 requires minimum physical protection, Level 2 requires the addition of tamper-evident mechanisms such as a seal or enclosure, while Level 3 specifies stronger detection and response mechanisms. Finally, Level 4 mandates environmental failure protection and testing (EFP and EFT). Thus, increasingly high levels of security can be provided albeit at higher chip costs.

An example of a cryptographic module that provides very high levels of physical security is IBM's 4758 PCI cryptographic adapter [39, 40] (FIPS 140-1 Level 4). The device includes internal tamper circuitry to detect physical penetrations as well as sensor circuitry to detect and respond to temperature and voltage attacks.

Bus encryption

A good countermeasure against bus probing attacks involves the use of processors that encrypt all information sent on global buses [62, 63]. Such processors ensure that only encrypted code/data values remain in the open (memory, address and data buses, *etc.*), which are then decrypted within the processor on-the-fly. The processor is also required to encrypt any value before it is released outside its I/O boundary. While such processors tend to achieve high levels of security, they also entail significant performance overheads. In addition, practical implementations have been found to be vulnerable to specialized forms of side-channel attacks, necessitating additional countermeasures [64].

Side-channel Attack Protection Measures

Various countermeasures against side-channel attacks have been proposed to remove the symptoms that make an embedded system vulnerable to monitoring and analysis of side-channel information such as power, timing, and electromagnetic radiation. Randomization is frequently used as an effective measure against any side-channel attack that requires the attacker to know exactly when a certain operation is performed. For example, the use of a randomized clock signal is suggested as an effective means to introduce non-determinism in smartcard processors [19]. This countermeasure also requires the introduction of random switching activity during the idle cycles associated with a random clock to prevent reconstruction of the clock signal.

Several mechanisms have been proposed to counter individual side-channel attacks. Techniques to counter power analysis attacks [25, 26] include data masking to hide sensitive information, use of reduced signal amplitudes, and introduction of noise into power measurement data. These mechanisms provide tamper resistance by increasing the number of samples needed for a successful power analysis attack to an infeasibly large number. Aggressive shielding techniques as well as methods that break the locality of chip layout (that is, allow for components in a chip to be spread across the entire chip surface) are effective in defeating electromagnetic analysis attacks [36]. Transient fault attacks on cryptographic hardware can be prevented by using concurrent error detection methods [65], while sensors that monitor environmental changes can be effective in detecting various fault injection attacks and launching appropriate recovery mechanisms [66].

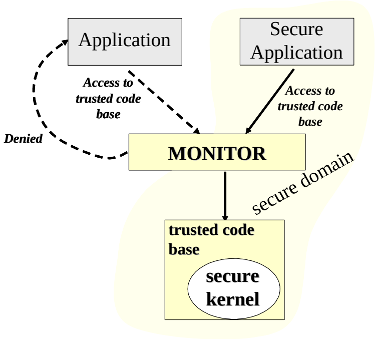

Figure 4: Providing security against malicious software attacks in ARM TrustZone [45]

3.3 Case Study: ARM TrustZone

The TrustZone security technology [45] from ARM provides a commercial example of how countermeasures against software attacks (and limited physical attack protection) are implemented for an embedded system-on-chip. The primary objective of TrustZone is to establish a clear separation of access to sensitive information and other HW/SW portions of an ARM-based system-on-chip architecture. This is achieved by evolving a secure domain using a "trusted code base" that resides in a secure area of the processor. The trusted code base is responsible for regulating the security of the entire system, starting from the system boot sequence. In addition, the trusted code is responsible for all security tasks that involve manipulation of keys.

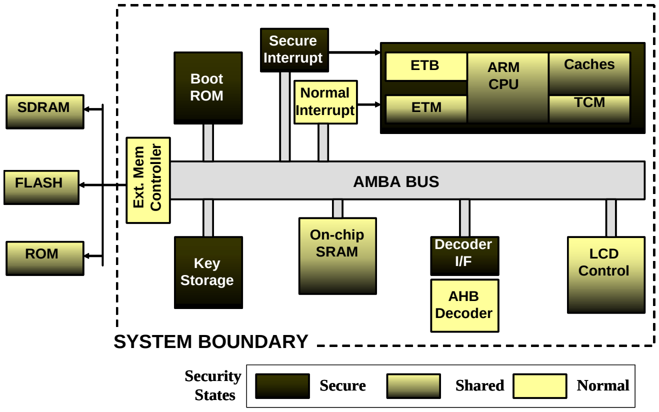

Figure 5: Components of an embedded system-on-chip architecture demarcated into secure and non-secure areas [45]

The trusted code base is protected by implementing a separate secure domain as shown in Figure 4. Non-secure applications are denied access to the trusted code base, while trusted applications are identified before they are provided access. This demarcation is enforced by the addition of a security tag called "S-bit" throughout the architecture. The S-bit defines the security operation state of the system and is used to denote parts of the system (ARM core, memory system, selected peripherals, *etc.*), which are secure. Access to the S-bit is through a separate processor operating mode called *monitor mode*, which itself can be accessed through a limited and pre-defined set of entry points. The monitor mode is responsible for controlling the S-bit, verifying that data and instruction accesses made by an application are permitted as well as ensuring a secure transition between secure and non-secure states.

The use of TrustZone to secure a typical embedded SOC is shown in Figure 5, wherein the security perimeter of the system extends beyond the processor core to the memory hierarchy and peripherals. The overall SOC architecture is divided into secure and non-secure regions. For example, the boot code is stored securely in the on-chip boot ROM since modifications to the boot process would render any security scheme ineffective. The memory is segmented into secure and non-secure areas. The S-bit and the monitor mode are used to ensure that secure data is not leaked to the non-secure area. Exception handling is also partitioned into normal and secure areas. Since interrupts can be used to freeze the processor when it is processing sensitive information, the monitor mode is used to process critical interrupts.

In summary, the TrustZone technology provides an architecture-level security solution to enforce a trusted code base, enable certification of trusted software independent of the operating system (OS), and provide protection against malicious software attacks.

4 Conclusions

In this paper, we examined the various ways in which embedded systems can be attacked by malicious agents. For these scenarios, we also saw how a wide array of countermeasures have been developed by researchers to provide tamper resistance in embedded systems. We believe that a clear understanding of attacks as well as the trade-offs associated with deploying tamper resistance mechanisms will enable a system architect to develop a truly secure embedded system.

Acknowledgements: The authors thank Divya Arora (Princeton University) and Vijay Raghunathan (University of California, Los Angeles) for their useful inputs and suggestions.

References

[1] S. Ravi, A. Raghunathan, and S. Chakradhar, "Embedding Security in Wireless Embedded Systems," in *Proc. Int. Conf. VLSI Design*, pp. 269–270, Jan. 2003.

[2] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in Embedded Systems: Design Challenges," 2004.

[3] *The code book: The science of secrecy from ancient Egypt to quantum cryptography*. Anchor Books, 2000.

[4] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, 1996.

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

[6] R. Anderson, "Why cryptosystems fail," *Communications of the ACM*, vol. 37, pp. 32–40, Nov. 1994.

[7] B. Schneier, *Security pitfalls in cryptography*. (`http://www.schneier.com/essay-pitfalls.html`).

[8] B. Schneier, "Cryptographic Design Vulnerabilities," *IEEE Computer*, vol. 31, pp. 29–33, Sept. 1998.

[9] *Security engineering: A guide to building dependable distributed systems*. John Wiley & Sons, 2001.

[10] J. Kelsey, B. Schneier, and D. Wagner, "Protocol interactions and the chosen protocol attack," in *Proc. Int. Wkshp. on Security Protocols*, pp. 91–104, Apr. 1997.

[11] C. Salter, O. S. Saydjari, B. Schneier, and J. Wallner, "Towards a Secure System Engineering Methodology," in *Proc. New Security Paradigms Wkshp.*, pp. 2–10, Sept. 1998.

[12] *Common Vulnerabilities and Exposures*. (`cve.mitre.org/`).

[13] *Latest Virus Threats*. Symantec Corporation (`http://www.symantec.com/avcenter/vinfodb.html`).

[14] *Virus Information*. Computer Security Resource Center, National Institute of Standards and Technology (`http://csrc.nist.gov/virus/`).

[15] *Vulnerability notes database*. CERT coordination center (`http://www.kb.cert.org/vuls/`).

[16] A. K. Ghosh and T. M. Swaminatha, "Software security and privacy risks in mobile e-commerce," *Communications of the ACM*, vol. 44, pp. 51–57, february 2001.

[17] E. Chien and P. Szor, *Blended attack exploits, vulnerabilities, and buffer-overflow techniques in computer viruses*. Symantec White Paper (`http://securityresponse.symantec.com/avcenter/whitepapers.html`).

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

[19] O. Kommerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," in *Proc. USENIX Wkshp. on Smartcard Technology (Smartcard '99)*, pp. 9–20, May 1999.

[20] *Smart Card Handbook*. John Wiley and Sons.

[21] E. Hess, N. Janssen, B. Meyer, and T. Schutze, "Information Leakage Attacks Against Smart Card Implementations of Cryptographic Algorithms and Countermeasures," in *Proc. EUROSMART Security Conference*, pp. 55–64, June 2000.

[22] J. J. Quisquater and D. Samyde, "Side channel cryptanalysis," in *Proc. of the SECI*, pp. 179–184, 2002.

[23] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," in *Proc. ESORICS'98*, pp. 97–110, Sept. 1998.

[24] P. Kocher, J. Jaffe, and B. Jun, *Introduction to differential power analysis and related attacks*. (`http://www.cryptography.com/resources/whitepapers/`).

[25] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Examining Smart-Card Security under the Threat of Power Analysis Attacks," *IEEE Trans. Comput.*, vol. 51, pp. 541–552, May 2002.

[26] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," *Advances in Cryptology – CRYPTO'99, Springer-Verlag Lecture Notes in Computer Science*, vol. 1666, pp. 388–397, 1999.

[27] L. Goubin and J. Patarin, "DES and differential power analysis," *in Proc. Cryptographic Hardware and Embedded Systems*, pp. 158–172, 1999.

[28] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Power analysis attacks of modular exponentiation in smartcards," *in Proc. Cryptographic Hardware and Embedded Systems (CHES)*, pp. 144–157, 1999.

[29] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," *Advances in Cryptology – CRYPTO'96, Springer-Verlag Lecture Notes in Computer Science*, vol. 1109, pp. 104–113, 1996.

[30] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack," in *Proc. Third Working Conf. Smart Card Research and Advanced Applications*, pp. 167–182, Sept. 1998.

[31] D. Brumley and D. Boneh, "Remote Timing Attacks Are Practical ," in *Proc. 12th USENIX Security Symp.*, pp. 1–14, Aug. 2003.

[32] D. Boneh, R. DeMillo, and R. Lipton, "On the importance of checking cryptographic protocols for faults," *in Proc. of Eurocrypt'97*, pp. 37–51, 1997.

[33] S. Govindavajhala and A. W. Appel, "Using Memory Errors to Attack a Virtual Machine," in *Proc. IEEE Symposium on Security and Privacy*, pp. 154–165, May 2003.

[34] W. van Eck, "Electromagnetic radiation from video display units: An eavesdropping risk?," *Computers and Security*, vol. 4, no. 4, pp. 269–286, 1985.

[35] M. G. Kuhn and R. Anderson, "Soft Tempest: Hidden Data Transmission Using Electromagnetic Emanations," in *Proc. Int. Wkshp. on Information Hiding (IH '98)*, pp. 124–142, Apr. 1998.

[36] J. J. Quisquater and D. Samyde, "ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards," *Lecture Notes in Computer Science (Smartcard Programming and Security)*, vol. 2140, pp. 200–210, 2001.

[37] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," *in Proc. Cryptographic Hardware and Embedded Systems*, pp. 251–261, 2001.

[38] B. Yee, *Using Secure Co-processors*. PhD thesis, Carnegie Mellon University, 1994.

[39] *Secure Coprocessing*. IBM Inc. (`http://www.research.ibm.com/scop/`).

[40] *The IBM PCI Cryptographic Coprocessor*. IBM Inc. (`http://www-3.ibm.com/security/cryptocards/`).

[41] *Cryptocell^{TM}*. Discretix Technologies Ltd. (`http://www.discretix.com`).

[42] 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.

[43] 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.

[44] T. Gilmont, J. D. Legat, and J. J. Quisquater, "An Architecture of Security Management Unit for Safe Hosting of Multiple Agents," in *Proc. Int. Wkshp. on Intelligent Communications and Multimedia Terminals*, pp. 79–82, Nov. 1998.

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

[46] *Next-Generation Secure Computing Base (NGSCB)*. Microsoft Inc. (`http://www.microsoft.com/resources/ngscb/productinfo.mspx`).

[47] P. N. Glaskowsky, *Microsoft Details Secure PC Plans*. Microprocessor Report, In-stat/MDR, June 2003.

[48] *LaGrande Technology for Safer Computing*. Intel Inc. (`http://www.intel.com/technology/security`).

[49] A. Arbaugh, D. J. Farber, and J. M. Smith, "A Secure and Reliable BootStrap Architecture," in *Proc. of IEEE Symposium on Security and Privacy*, pp. 65–71, May 1997.

[50] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *Proc. ACM Symposium on Operating Systems Principles*, pp. 178–192, Oct. 2003.

[51] T. Garfinkel, M. Rosenblum, and D. Boneh, "Flexible OS Support and Applications for Trusted Computing," in *Proc. 9th Wkshp Hot Topics in Operating Systems*, May 2003.

[52] M. Blaze, "A Cryptographic File System for UNIX," in *Proc. ACM Conf. on Computer and Communications Security*, pp. 9–16, Nov. 1993.

[53] E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing Remote Untrusted Storage," in *Proc. ISOC Network and Distributed Systems Security (NDSS) Symp.*, pp. 131–145, 2003.

[54] Y. Chen, R. Venkatesan, M. Cary, S. Sinha, and M. H. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *Proc. Int. Wkshp. Information Hiding*, pp. 400–414, Oct. 2002.

[55] G. C. Necula and P. Lee, "Proof-Carrying Code," Tech. Rep. CMU-CS-96-165, Carnegie Mellon University, Nov. 1996.

[56] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution via Program Sheperding," in *Proc. 11th USENIX Security Symp.*, Aug. 2002.

[57] D. L. Detlefs, K. Leino, G. Nelson, and J. Saxe, "Extended static checking," tech. rep., Systems Research Center, Compaq Inc., 1998.

[58] B. Chess, "Improving computer security using extended static checking," in *Proc. IEEE Symposium on Security and Privacy*, pp. 148–161, May 2002.

[59] E. M. Clarke, S. Jha, and W. Marrero, "Using state space exploration and a natural deduction style message derivation engine to verify security protocols," in *Proc. IFIP Working Conf. on Programming Concepts and Methods*, 1998.

[60] G. Lowe, "Towards a completeness result for model checking of security protocols," in *Proc. 11th Computer Security Foundations Wkshp.*, 1998.

[61] *Security Requirements for Cryptographic Modules (FIPS PUB 140-2)*. (`http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`).

[62] R. M. Best, *Crypto Microprocessor for Executing Enciphered Programs*. U.S. patent 4,278,837, July 1981.

[63] M. Kuhn, *The TrustNo 1 Cryptoprocessor Concept*. CS555 Report, Purdue University (`http://www.cl.cam.ac.uk/~mgk25/`), Apr. 1997.

[64] M. Kuhn, "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP," *IEEE Trans. Comput.*, vol. 47, pp. 1153–1157, Oct. 1998.

[65] R. Karri, K. Wu, P. Mishra, and Y. Kim, "Concurrent Error Detection Schemes for Fault-Based Side-Channel Cryptanalysis of Symmetric Block Ciphers," *IEEE Trans. Computer-Aided Design*, vol. 21, pp. 1509–1517, Dec. 2002.

[66] D. Samyde, S. Skorobogatov, R. Anderson, and J.-J. Quisquater, "On a new way to read data from memory," in *Proc. First Intl. IEEE Security in Storage Wkshp*, pp. 65–69, Dec. 2002.