Topics in Secure Embedded System Design

Nachiketh Potlapally

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Electrical Engineering

January, 2008

Abstract

Pervasive networks have led to widespread use of embedded systems, like cell phones, PDAs, RFIDs etc., in increasingly diverse applications. Many of these embedded system applications handle sensitive data (e.g., credit card information on a mobile phone/PDA) or perform critical functions (e.g., medical devices or automotive electronics), and the use of security protocols is imperative to maintain confidentiality, integrity and authentication of these applications. Typically embedded systems have low computing power and finite energy supply based on a battery, and these factors are at odds with the computationally intensive nature of the cryptographic algorithms underlying many security protocols. In addition, secure embedded systems are vulnerable to attacks, like physical tampering, malware and side-channel attacks. Thus, design of secure embedded systems is guided by the following factors: small form factor, good performance, low energy consumption (and, thus, longer battery life), and robustness to attacks.

This thesis presents our work on tackling three issues in the design of secure embedded systems: energy consumption, performance and robustness to side-channel attacks. First, we present our work on optimizing the energy consumption of the widely employed secure sockets layer (SSL) protocol running on an embedded system. We discuss results of energy analysis of various cryptographic algorithms, and the manner in which this information can be used to adapt the operation of SSL protocol to save energy. Next, we present results of our experiments on optimizing the performance of Internet protocol security (IPSec) protocol on an embedded processor. Depending on the mode of operation, the IPSec computation is dominated by cryptographic or non-cryptographic processing. We demonstrate how both these components of the IPSec protocol can be optimized by leveraging the extensible and configurable features of an embedded processor. Next, we introduce a satisfiability-based framework for enabling side-channel attacks on cryptographic software running on an embedded processor. This framework enables us to identify variables in the software implementations which result in the disclosure of the secret key used. Thus, security of software implementations can be improved by better protection of these identified variables. Finally, we conclude by introducing a novel memory integrity checking protocol that has much lower communication complexity than existing Merkle tree-based protocols while incurring a modest price in computation on the processor. This scheme is based on Toeplitz matrices, and can be very efficiently realized on embedded systems with hardware extensions for bit matrix operations.

Acknowledgments

First and foremost, I would like to thank my advisors, Prof. Niraj Jha and Prof. Ruby Lee, for generously supporting me during my stay at Princeton. This thesis would not have been possible in its present form without their guidance, and the freedom they gave me in pursuing various ideas. The thesis bears marks of their meticulous attention to consistency and clarity of content. Any shortcomings in the thesis are entirely due to my oversight.

I have benefitted immensely with my association with Dr. Anand Raghunathan, Senior Research Staff Member, NEC Labs America who generously gave his time and suggestions which significantly influenced this thesis. I offer my sincere gratitude for everything he has done for me from the time I went to NEC labs as a novice researcher in 1999. My other collaborator at NEC Labs, Dr. Srivaths Ravi, helped me at various stages of my research with valuable advice, and his diligent attention to making me present various aspects of my research lucidly. I wholeheartedly thank him.

Taking courses in the Math and Computer Science departments was an intellectually rewarding experience, and a great pleasure. The credit for this largely goes to Prof. Avi Wigderson, Prof. Nicholas Pippenger, and Prof. Peter Sarnak. My stay at Princeton would have been much poorer without the company of some good friends I made along the way. Loganathan Lingappan acted as a sounding board for many of my ideas, and I enjoyed the discussions I had with him. Patrick Murphy was a wonderful roommate, and introduced me to the music of Bob Dylan with his “Dylan for Dummies” compilations. Sathyakama Sandilya helped me with my generals, and I have seen few as generous and modest as him. My first year at Princeton was made agreeable by my association with Subbu

Rajagopalan. Alex Kattamis kept days interesting with either politics or food or beverages or some combination of the three. Vassos Soteriou's quest for salvation made sure there were not many dull moments in the office. I would like to acknowledge Yogesh Mahajan, Pallav Gupta, and Gregory Nieberg. I thank all these people for the rich experience they afforded me.

She gently propped me up when I was down. She planted my feet firmly on the ground when I had my head in the clouds. She enriched my vocabulary with words like *riparian*, *eutrophication*, and *methanogenesis*. I cannot thank you enough for all the things you have done for me, Junu Shrestha. Finally, I humbly and respectfully thank my parents and grandmother for their selfless love and constant encouragement. They set aside personal desires so that I could achieve this. I remain indebted to them.

To my grandmother, Ananthalakshmi.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Security is an important concern in networked applications due to increasingly sensitive data being exchanged on the computer networks. This concern necessitates the use of security protocols to protect the sensitive data from unauthorized snooping or manipulation by adversaries. A significant fraction of the devices making up present-day networks are embedded systems which are marked by low processing power and finite energy supply derived from a battery. However, existing security protocols are computationally intensive, and executing them on resource-constrained embedded systems leads to the problem of reduced performance and battery life. In addition, many embedded systems are especially susceptible to a type of non-invasive attacks called side-channel attacks. This thesis proposes techniques for improving the performance, battery life and robustness to side-channel attacks of secure embedded systems.

This chapter begins with an introduction to various security threats present in computer networks, and the security protocols which are formulated to counter these threats in Section 1.1. Section 1.2 motivates the need for secure embedded systems. Section 1.3 discusses the challenges involved in designing secure embedded systems, and provides the motivation for the thesis. Finally, the chapter concludes with Section 1.4 which gives the outline of chapters of this thesis.

1.1 Threats and security protocols

Data being exchanged on computer networks are open to a variety of threats from unauthorized parties, collectively known as adversaries. The presence of high-speed and large-throughput networks has enabled the migration of many important applications, like voting, filing taxes, banking, heath monitoring, *etc.*, online. The high value of the data being exchanged by these applications has made it an attractive target for attacks by adversaries motivated by commercial gain, fame for notoriety and other factors. According to a CERT report on electronic crime [2], 150 million dollars were lost due to electronic attacks on the organizations in spite of 62% of the surveyed organizations refusing to disclose their losses due these attacks. Thus, it is important to employ security mechanisms in order to protect electronic transactions. Security refers to techniques employed to prevent adversaries from carrying out any unpermitted action which affects the data being exchanged between legitimate parties in an unauthorized manner. Usually, in the security literature, any two legitimate entities interacting with each other are referred to as Alice and Bob, and an adversary is called Eve. We adopt the same convention in rest of this chapter. Data exchanged on the computer networks are susceptible to the following adversarial threats:

1. Eve observes data being exchanged between Alice and Bob with the intention of finding out the content.
2. Eve modifies communication between Alice and Bob.
3. Eve impersonates as Alice to Bob, or as Bob to Alice.
4. Eve disrupts the network so that Alice and Bob cannot communicate with each other.

We employ security protocols in order to counter the above-mentioned threats. A security protocol is a sequence of steps, followed by two or more parties, such that certain security objectives are satisfied [3]. A security objective is formulated to either counter one of the threats listed above, or to ensure that interactions between legitimate parties satisfy some requirements. Following are the common security objectives which need to be satisfied by security protocols:

1. Confidentiality (or secrecy): Information is not disclosed to unauthorized entities.

2. Integrity: Any unauthorized manipulation of data can be detected.

3. Authentication: An unauthorized entity should not be able to pose as a legitimate entity.

4. Non-repudiation: An authorized entity should not be able to disown its legitimate communication with other authorized entities.

5. Availability: A system should be usable on demand by an authorized entity.

In the list given above, non-repudiation (4) is the only property which does not involve an adversary, and is essential for upholding the validity of legitimate online transactions. Typically, security protocols are built using cryptographic algorithms to satisfy confidentiality (1), integrity (2), authentication (3) and non-repudiation (4), while availability (5) is made possible through the use of access control security mechanisms. In general, it is expensive to design security protocols which can counter all the possible threats. Moreover, higher the complexity of a security protocol, greater are the chances of introducing design flaws into the protocol implementation, and thereby creating inadvertent security vulnerabilities. Therefore, in practice we adopt a application-specific threat model approach. In this approach, we begin by defining a reasonable threat model comprising a set of threats which are relevant to the target application. Next, we identity the appropriate security objectives needed to counter the threats listed in the threat model, and then proceed to define a security protocol that achieves all the chosen objectives. Most security protocols are aimed at protecting data while being transmitted over communication channels. However, that alone is not sufficient, and we need to protect the end points too. By end points, we refer to computing devices that Alice uses to generate her data before she sends it over to Bob, and likewise for Bob. Attacks on these devices have the goal of retrieving sensitive material, like cryptographic keys, stored in the device, and these attacks are either invasive or non-invasive. Invasive attacks consist of physically probing the internal circuitry of a device in order to extract the sensitive material, while non-invasive techniques consist of software attacks (using viruses, worms, *etc*) and attacks based on the statistical analysis of operational characteristics of the device to extract secret information. Non-invasive attacks of the latter kind which are based on analyzing the device operational characteristics to extract the secret key are called side-channel attacks. Examples of operational characteristics which formed the basis of successful side-channel attacks include timing information, power dissipation, electromagnetic radiation, and device operation in the presence of faults. Side-channel attacks are quite dangerous because of two reasons. First, they do not cause any damage to the device, and thus the probability of escaping detection is very high, and second, they provide a tractable way of breaking algorithms with very high theoretical cryptanalytic strength. The nature of embedded systems makes them a target of effective side-channel attacks. The portable nature of embedded devices makes it easier for an adversary to take possession of the device temporarily in order to record operational characteristics for various inputs, and perform the statistical analysis off-line. Also, most embedded systems are designed for specific applications, and the resulting simpler circuitry facilitates effective side-channel attacks on these devices. Thus, securing electronic data against adversaries constitutes two parts: first, improve the robustness of end systems against attacks aimed at securing the sensitive information, and formulate security protocols to protect data while in transmission between end points.

The steps of a security protocol are implemented using cryptographic algorithms. A cryptographic algorithm is a mathematical function that implements encryption and decryption parameterized on a key. One of the basic assumptions of modern cryptography is that security of a cryptographic algorithm must be based only on the secrecy of the key used, and not in keeping the algorithm secret. This principle is called Kerchoff's principle [4]. For any algorithm, the total number of key values possible is referred to as the keyspace. Thus, Kerchoff's principle dictates that the key should be so chosen that given any input and the corresponding output of a cryptographic algorithm, it should be computationally infeasible for an adversary to search the entire keyspace in order to find the correct key. Let M be the plaintext message, C be the corresponding ciphertext, and K be the key used to map M to C using encryption function $E()$. Then, encryption is formally denoted as, $C = E_K(M)$ where $E_K()$ represents encryption function $E()$ parameterized on key K. Similarly, if $D_K()$ represents the decryption function $D()$ parameterized on key K, then, we have $M = D_K(C)$. For any key K and plaintext M, the encryption function $E()$ and decryption function $D()$ satisfy the property that $D_K(E_K(M)) = M$. If the constructions of functions $E()$ and $D()$ satisfy Kerchoff's principle, then, it will be computationally infeasible for an adversary to find the value of key K used to map M to C using $C = E_K(M)$ given that $E()$, $D()$, M, and C are known. Cryptographic algorithms can be divided into the following three types:

- Symmetric algorithms: In these algorithms, encryption and decryption processes use the same key which has to be kept secret. Therefore, these algorithms are also known as secret-key algorithms. They are primarily used for realizing confidentiality. If Alice and Bob wish to encrypt their communication using a symmetric algorithm, then they have to agree upon a key before they start communicating. The process of securely agreeing upon a secret key in the presence of adversaries can be quite challenging.

- Asymmetric algorithms: In these algorithms, encryption and decryption functions use different keys such that the key used for encryption is made public and the key used for decryption is kept secret. The encryption and decryption keys are called public and private keys, respectively. These algorithms are also known as public-key algorithms, and they are primarily used for implementing authentication and non-repudiation. Let K_A^{public} and $K_A^{private}$ be the public and private keys of Alice, respectively. For any plaintext M, the public and private keys satisfy the property that $M = D_{K_A^{private}}(E_{K_A^{public}}(M))$. Alice publishes key K_A^{public} and keeps key $K_A^{private}$ secret. If Bob wishes to send an encrypted form of message M to Alice, he computes ciphertext $C = E_{K_A^{public}}(M)$ and sends it to Alice. Since Alice alone knows her private key, she obtains message M by computing $M = D_{K_A^{private}}(C)$. When Alice wishes to authenticate herself, she derives a signature S by encrypting a short message M with her private key, $K_A^{private}$, *i.e.*, $S = E_{K_A^{private}}(M)$. Anyone can verify Alice's signature S by decrypting it with her public key, K_A^{public}, and comparing the result with M. Since only Alice knows her private key, successful decryption implies that Alice generated the signature S.

- Hash algorithms: Hash algorithms take an arbitrary-sized input and map it to a fixed-length hash value using a efficiently computable compression function. The hash functions are primarily used to check for data integrity, and sometimes employed for authentication when the hash function is parameterized on a key (known as hash message authentication code). It is absolutely necessary that a cryptographic hash function satisfy three properties, namely: *pre-image resistance* (meaning it is computationally infeasible to invert a hash function), *second pre-image resistance* (which implies that given message M and its corresponding hash value h, it is infeasible to find another message $M' \neq M$ that maps to the same hash value h), and *collision resistance* (which says that it is intractable to find two messages M and M' that map to the same hash value).

1.2 Secure embedded systems

An embedded system is defined as a computing platform which performs a limited set of functions using a combination of hardware and software under real-time constraints. Development of embedded systems has transformed the computing paradigm from a desktop and server-dominated scenario to a distributed computing model comprising lots of embedded systems distributed within the environment, taking information as input and performing some processing on the data before transmitting them to the centralized servers. Examples of these embedded systems include sensor nodes gathering sense data about their environment and transmitting it to a central data aggregation node in a distributed sensor network, Internet connectivity enabled handhelds used by users to access their online bank accounts, *etc.* The versatility of this computing model coupled with the widespread deployment of high-bandwidth wireless networks has led to many important applications executing on desktops to be ported to embedded systems. According to a recent survey, the amount of retail commercial transactions using smartcard-based embedded systems is projected to reach $2 billion by 2008 [5]. Embedded systems have a diverse array of system characteristics and are being increasingly deployed in applications which handle sensitive data. Examples of some of these applications are given below:

- Machine-readable travel document: An Epassport has a radio frequency identification (RFID) tag, containing important personal information of the holder of the passport, embedded into it. The RFID tag interacts with Epassport readers installed in the airports to automatically verify the holder of the passport. Typical system features of a RFID tag are:
  - 1.28-1.92MHz processor.
  - 128-512 bit ROM.
  - 32-128 bit RAM.
  - Around 60000 transistors.
  - Battery energy supply (for an active tag).
- Epayment: Europay, Visa and Mastercard (EMV) compliant smartcard is an embedded device that enables users to perform highly secure retail transactions. In order to authorize payment for a sale, the smartcard information is verified by a point of sale reader installed in the commercial establishment. Typical system features of a smartcard are:
  - 66-100MHz processor.
  - 240KB ROM.
  - 16KB RAM.
  - 500000-1000000 transistors.

  - Battery energy supply (for an active card).

- Mobile brokerage: Embedded devices, like PDAs and cell-phones, are used being used to manage portfolios with help of software, like Microsoft Money. In these applications, the embedded devices carry transactions with a central server through the Internet. Typical characteristics of the PDA-like devices are:

  - 133-300MHz processor.
  - 16MB ROM.
  - 64MB Flash RAM.
  - $>$5-10 million transistors.
  - Battery energy supply.

In all the above applications, embedded systems process sensitive information. Therefore, we need to deploy security protocols on these systems to protect the information from unauthorized disclosure and tampering. As mentioned in the previous section, we need security mechanisms to not only protect data which are being transmitted by the embedded device, but, also protect the sensitive information stored on the device. Also, from the list of typical system characteristics of embedded systems listed above, we can see that the embedded systems have the following constraints: low processing capability, limited memory, and finite battery-derived energy supply. However, security protocols are computationally intensive, and deploying them on resource-constrained embedded systems without appreciably degrading their performance is a challenging task. In the next section, we will elaborate more on the challenges involved in the design of secure embedded systems.

1.3 Challenges of secure embedded systems

In the last section, we saw that embedded devices have inherent operating constraints of low processing power, small memory storage, and limited energy supply. In addition, embedded systems are marked by portability and a small form factor. These unique traits of embedded systems make it especially challenging to realize security functionality in embedded systems, as explained below:

- Reduced performance: Cryptographic algorithms, which are one of the building blocks of security protocols, are computationally intensive. Executing security protocols on low-end processors present in embedded systems drastically degrades their performance. For example, the Intel SA-1100 embedded processor offers 150 million instructions per second (MIPS) of computational power at 133MHz. Cryptographic algorithms, when 3DES and SHA are used to provide confidentiality and data integrity, respectively, at a data rate of 2Mbps, require 130 MIPS, leaving only 20 MIPS for non-cryptographic processing. Thus, security protocols degrade performance of embedded systems [6].

- Short battery life: The computationally intensive nature of cryptographic algorithms results in an appreciable reduction in the battery life of embedded systems when security protocols are used. Consider a Sensoria wireless sensor node equipped with a Motorola DragonBall MC68328 processor used in an application where it senses data and transmits them to a centralized node. In sensor nodes, communication is the most expensive operation with respect to energy consumption and the Sensoria node requires 21.5 mJ to transmit a bit at a data rate of 10 kbps. If the RSA algorithm is used for authenticating the node, it requires an additional 42 mJ/bit. This results in the node battery with a capacity of 26kJ running out more than twice as fast as when there is no security [7].

- Vulnerability to attacks: The small form factor and portable nature of embedded systems makes them especially susceptible to side-channel attacks. In order to launch a side-channel attack, an adversary first needs to record information, like power, timing, electromagnetic emission, *etc.*, while the device is performing computations parameterized on the secret information, and later, analyze the recorded information using statistics to extract the secret information. The portable quality of the embedded device together with the non-invasive nature of side-channel attack make it easier to compromise the secret information and escape detection.

Thus, designing secure embedded systems has three objectives, namely, maintaining good performance, ensuring a sufficiently long battery life, and making them robust to known side-channel attacks.

1.4 Dissertation contributions

This dissertation takes steps in the direction of building secure embedded systems that have good performance, whose battery life is sufficiently long, and those that are robust to non-invasive attacks, like side-channel attacks. The contribution made by this dissertation are as follows:

- Provide a comprehensive analysis of energy consumption characteristics of cryptographic algorithms on a handheld. Based on this analysis, protocol-level energy optimizations are suggested for the secure sockets layer (SSL) protocol.
- Give an in-depth analysis of the performance of Internet security (IPSec) protocol execution on an embedded processor, in terms of cryptographic and protocol components. And, optimize both the cryptographic and protocol components of IPSec execution through the use of extensibility and configurability features of an embedded processor.
- Introduce a new side-channel attack framework on cryptographic software running on an embedded processor using a Satisfiability solver. Using this framework, intermediate variables in computation of popular cryptographic algorithms are identified which would result in the secret key being leaked.

The dissertation is organized as follows. Related work is described in Chapter 2. Chapter 3 describes the results on energy analysis of cryptographic algorithms, and ways of optimizing the energy consumption of the SSL protocol through protocol-level optimizations. Chapter 4 describes the performance optimization of IPSec protocol execution on an embedded processor by using its configurable and extensible features to target non-cryptographic and cryptographic computations, respectively. Chapter 5 presents details of the novel SAT-based side-channel attack framework, and demonstrates it in the context of popular cryptographic algorithms DES, 3DES and AES. Chapter 6 gives the details of a novel memory integrity checking protocol that makes much fewer additional queries to the external memory than existing methods based on Merkle trees. Finally, we conclude with a summary of the dissertation and ideas for future work in Chapter 7.

Chapter 2

Related Work

In this chapter, we survey existing literature related to issues involved in implementing security functionality in embedded systems. We restrict ourselves to three aspects of embedded system security, namely: impact of security on the battery life, the performance of secure embedded systems, and their susceptibility to side-channel attacks. We begin by discussing works which analyze the energy cost of embedded security processing in Section 2.1. Section 2.2 surveys the works dealing with improving the performance of security processing on embedded systems. Section 2.3 gives a discussion of the literature on side-channel attacks on embedded systems.

2.1 Energy consumption of security protocols in embedded systems

Most embedded systems derive their energy supply from a battery. Research has shown that the battery capacity is growing at a much slower rate than the energy requirement of the newer applications, resulting in a *battery gap* [8]. Security protocols and the underlying cryptographic algorithms impose significant computational requirements, and introducing security functionality into embedded systems further exacerbates the existing battery gap. The shortened battery life of secure embedded systems has undesirable consequences for the usability of embedded systems. This observation has motivated researchers to investigate energy-efficient ways of implementing security functionality in embedded systems.

The work by Carmen *et al.* [7] was one of the first to study the energy consumption of cryptographic algorithm execution on embedded systems. Based on analytical estimation of the number of operations involved in different cryptographic algorithms, they computed energy consumption values of some asymmetric, symmetric, and hash algorithms on various popular embedded processors. This information was used to evaluate the energy efficiency of different asymmetric-key and symmetric-key key management protocols for distributed sensor networks. A similar study on the energy cost of establishing keys in sensor nodes was done by Hodjat *et al.* [9]. They restricted their study to Elliptic-curve Diffie-Hellman Key exchange protocol, and AES algorithm, and obtained energy values by measuring energy consumption of software implementations of these algorithms running on a strongARM-based wireless integrated network sensor (WINS) node [10]. An important step toward energy-efficient security protocols for resource-constrained sensor nodes was proposed by Perrig*et al.* in SPINS [11]. They proposed a set of protocols based on block ciphers and message authentication codes for providing data confidentiality, two-party data authentication, data freshness, and authenticated broadcast with low overhead in sensor networks. A comprehensive study of energy costs associated with implementing security features at the data link layer and network layer in sensor nodes was done in [12]. Jakobsson *et al.* [13] and Wong *et al.* [14] proposed energy-efficient mutual authentication and key exchange protocols targeted at client-server interactions where a handheld embedded system and a much more powerful desktop act as client and server, respectively. Karri *et al.* studied the energy consumed in various operations of a wireless transport layer security (WTLS) protocol running on a pocket PC containing a StrongARM embedded processor [15]. They identified two main sources of energy consumption in the WTLS protocol execution, namely: cryptographic computations and messages exchanged during secure session establishment and during secure data transactions. Based on these observations, they suggest techniques to optimize the energy dissipation of the WTLS protocol based on compression of messages, modifying the public-key algorithm-based operations in the handshake stage, and making cryptographic computations efficient through the introduction of custom hardware. The changes to the handshake stage include the server obtaining the client certificate from a trusted source rather than requiring the client to transmit it, embedding the client master secret in its certificate and refreshing it by combining it with random numbers exchanged between the client and server, and having a pre-determined master secret which is safely stored in the client and server devices. However, these proposals introduce additional security vulnerabilities, and they have not been examined in the paper. In chapter 3, we present the first comprehensive analysis of the energy consumption of a wide array of common asymmetric, symmetric and hash algorithms on an embedded system [16, 17]. Based on this information, suggestions for protocol-level energy optimizations for the SSL protocol are given.

2.2 Performance of security protocols in embedded systems

Security protocols and cryptographic algorithms impose significant computational overhead, and studies have indicated that they affect the performance of embedded systems in a significant manner [6, 18–22]. For example, Ravi *et al.* showed that a software implementation of SSL, using 3DES for encryption and SHA for message authentication, running on an iPAQ, requires 650 MIPS to achieve a link speed of 10 Mbps while the StrongARM embedded processor (in iPAQ) can deliver only 235 MIPS. Thus, we see that there is a significant difference between requirements of security processing and the capability of an embedded processor. This difference is termed *security processing gap* [23].

Evolution of embedded processing architectures to handle security processing can be divided into three stages [24]. In the first stage, we have optimized software implementations of cryptographic algorithms running on embedded processors. Though this approach has fast design times and good application-level flexibility, it has the drawback of low performance. In order to satisfy the high performance requirements of cryptographic algorithms, the second stage in the design evolution of secure embedded processors proposed the use of application-specific integrated circuits (ASICs). In this approach, a custom-designed circuit, which can accelerate execution of a particular cryptographic algorithm (or a limited set of algorithms), is introduced into the embedded processor. This allows the processor to offload expensive cryptographic computation onto the ASIC which is usually orders of magnitude faster. However, ASICs have very limited flexibility, and are not desirable when the target system has to support multiple cryptographic algorithms and emerging security standards. Thus, in order to combine the flexibility of the software approach with the high performance of ASICs, hybrid hardware-software approaches were proposed. This is the third stage in the evolution of embedded processors targeted toward efficient handling of cryptographic processing. An example of this new approach is an application-specific instruction-set processor (ASIP) where custom hardware is tightly coupled into the datapath of the processor, and is invoked through custom instructions. Usually, common operations in cryptographic algorithms, like permute, rotate, 1024-bit addition and multiplication, *etc.*, are the target for introduction as custom hardware [23]. The preliminary work in this hybrid approach was done by Shi and Lee [25] and Burke *et al.* [26] in the general-purpose processor domain. Shi and Lee [25] proposed the introduction of instructions to efficiently perform bit-level manipulations, like permute and rotate. While current microprocessors only support rotate, the ability to realize any arbitrary permutation of n bits is a very powerful diffusion operation. Diffusion is a fundamental class of operations in a wide array of block ciphers, and an instruction which implements permute efficiently can speed up the execution of all these block ciphers. While Lee *et al.* [27] targeted both new and existing ciphers, Burke *et al.* introduced instructions which implemented a set of operations common to many existing symmetric ciphers. These include substitutions, rotates and modular addition. Ravi *et al.* [23] designed an embedded processor with custom instructions aimed at accelerating modular exponentiation-based asymmetric algorithms, in addition to some symmetric and hash algorithms. Hodjat *et al.* [28] designed an embedded processor enhanced with a co-processor which implements the entire round of the AES algorithm. Using this design, they were able to demonstrate Gbit throughput rates for encrypting datastreams. This hybrid approach has found widespread acceptability in the commercial embedded processors used in wireless handsets, handhelds, *etc.* For example, the SmartMIPS MIPS32 4KSd [29] processor extends the basic MIPS instruction-set architecture to speed up cryptographic computations. Similar extensions can be found in ARM's SecurCore SC200 [30], TI's OMAP 1610 [31] and NEC's MP211 [32]. Table 2.1 presents a comparision of the features present in the above mentioned secure embedded processors. In Table 2.1, *secure mode execution* refers to isolation of a secure process through hardware support, *secure memory* denotes memory whose contents are erased in the presence of tampering, and *secure booting* is accomplished using the trusted platform technology.

Table 2.1: Comparision of features of some popular secure embedded processors

Features	SC200 (ARM)	MIPS32 4KSd (MIPS)	OMAP 1610 (TI)	MP211 (NEC)
Custom functionality support	Y	Y	Y	Y
Secure execution mode	Y	Y	Y	Y
Secure memory	Y	Y	Y	Y
Secure booting	N	N	N	Y
Random number generation support	N	N	Y	N
Tamper resistance	Y	Y	N	Y
Side-channel resistance	Y	N	N	N

ASIPs enhanced with cryptographic hardware extensions are extremely effective in improving the performance of cryptographic computations while maintaining flexibility to accommodate different algorithms and standards. However, an analysis of execution of network security protocols, like SSL and IPSec, reveals that protocol processing imposes as much computational burden as cryptographic computations [24]. By protocol processing, we refer to activities like building and stripping security protocol headers on packets, looking up security databases based on values in the headers, *etc.* Thus, cryptographic hardware accelerators alone will not suffice in such applications, and we need security protocol engines which accelerate both the protocol and cryptographic functionality. For application-level protocols, such as SSL, tuning the operating system to the working of the application [33] is one possible way to address some of the non-cryptographic processing bottlenecks. In the case of IPSec running on a general-purpose computing system, it has been shown that processing overheads, such as network processing, data copying and transfer times, severely limit the utility of cryptographic hardware in certain scenarios (e.g., TCP bulk transfers) [34]. Commercial hardware solutions for speeding up both cryptographic and packet processing in high-performance processors can be found in the enterprise server and backbone router markets [35–37]. These products integrate multiple specialized cores on a single chip to provide high-performance secure packet processing. These specialized cores perform activities like, searching the packet stream to identify those which require IPSec processing, building special packet headers, mainting the database of security policies associated with packet streams and so on. All the above mentioned commercial solutions have hardware support for optimizing execution of multiple cryptographic algorithms. For example, Hifn processors [35] have support for DES, AES, SHA, MD5, RSA and DSA, while Nitrox processors [36] support the same set of processors, in addition, to having a fast hardware random number generator. In order to speedup non-cryptographic processing in protocols like IPSec, these processors [35–37] incorporate dedicated memories with fast interfaces for efficient lookup into security policy databases. These designs fall under the category of secure network processors which are ASIPs designed to perform ordinary networking functions and network security processing efficiently [38]. A comparison of various techniques adopted for speeding up IPSec processing on high-performance processors can be found in [39]. However, the hardware costs of these solutions preclude them from being directly applicable in many embedded systems.

In chapter 4, we investigated the design of security protocol processing engines for low-end embedded devices [40,41]. These works developed a high-performance processor for performing efficient IPSec processing on resource-constrained embedded devices. Towards this end, they first analyze the performance of a lightweight IPSec software implementation [42] on a state-of-the-art embedded processor (Xtensa from Tensilica, Inc.). They exploit the processor's configurability (microarchitectural parameters) and extensibility (custom instructions) to provide a speedup of the protocol and cryptographic processing components of the IPSec protocol, respectively.

2.3 Side-channel attacks on embedded systems

Cryptography provides techniques for securing systems and the basic services of confidentiality, authentication and integrity. However, there exist different ways of circumventing the security provided by cryptographic algorithms to compromise the security of the system [43]. Techniques for doing this include

- Logical attacks: These attacks exploit flaws in the design or implementation of security protocols. These include software attacks, like buffer overflow attacks, cryptographic key management flaws, random number generator defects, use of debug modes to bypass security, improper algorithm implementation, use of weak passwords, *etc.*

- Invasive physical attacks: These are physical attacks which consist of cutting open the chip packaging and examining its internals with micro-probing techniques to eavesdrop on communication between components to read secret information.

- Non-invasive physical attacks: These attacks monitor information leaked by the circuit during cryptographic computation in order to figure out the secret key. The information leaked includes timing information, power dissipation, and electromagnetic radiation. These attacks are commonly referred to as physical side-channel attacks. In addition, behavior of a cryptographic algorithm in the presence of faults can also act as side-channel information.

Among the physical attacks listed above, the most dangerous ones are the non-invasive attacks. In these attacks, the attacker only needs to collect information during the normal operation of the target embedded device, and analyze this information in order to extract the secret key information. In this section, we will restrict our attention to a survey of these non-invasive attacks.

Differential and linear cryptanalysis are two powerful mathematical cryptanalytic techniques developed to break symmetric ciphers [44, 45]. However, these attacks require huge amounts of input data, thereby making a practical attack infeasible. For example, to break the 16-round DES, differential and linear cryptanalysis require 2^{47} chosen and known plaintexts, respectively (encrypted with the key to be computed). Development of side-channel attacks enabled practically feasible attacks on a number of popular cryptographic algorithms, *e.g.*, DES, RSA, DSS, *etc.* [46–49]. Kocher demonstrated how an RSA secret key could be determined by recording the time taken by the modular exponentiation computation, and analyzing small variations in time (depending on the presence of 0 or 1 in the secret key). In this attack, predictions are made about individual key bits, and these guesses are verified by using statistical techniques to determine whether there is a correlation between target system's timing behavior and the behavior expected by the prediction. Another popular source of side-channel information is power dissipation [47, 48]. In these attacks, power dissipated by the physical implementation of a cryptographic algorithm while processing various plaintexts, is used to extract the bits of the secret key. The principle underlying a power analysis attack is the observation that power dissipated by a circuit is dependent on the values of the internal *state bits* being manipulated. The state bits, in turn, are parameterized on the value of the secret key. Thus, knowledge of the state bits would give us information about the value of the bits of the unknown secret key. In these attacks, guesses are made for the value of few appropriate state bits, and the guess which results in maximum statistical correlation with the power dissipated is considered to be the actual value of the chosen state bits. Based on the value of the state bits, the value of the key bits is inferred. There are two types of power-based side-channel attacks, simple power analysis (SPA) and differential power analysis (DPA). In SPA, values of the bits of cryptographic keys is inferred directly from the power profile of the cryptographic computation by exploiting any obvious correlations between the two. For example, in modular exponentiation computation, presence of one in the private key exponent leads to an additional modular multiplication operation compared to when there is a zero bit. This extra modular multiplication operation leads to additional power consumption, and this information can be used to infer the value of the private key bits. On the other hand, a DPA attack is much more sophisticated than a SPA attack, and can infer secret key information even when there is no direct correlation between the power profile and the secret key bits. It is based on computing the value of some internal state bits by exploiting their correlation with the power dissipation values which are known. To realize this attack in practice, we need to implement the following steps:

- Select a set of internal state bits which are parameterized on a small subset of secret key bits, and plaintext (or ciphertext).
- Formulate a function which takes as its inputs, the values of plaintext (or ciphertext) and a subset of secret key bits, and calculates the value of the state bits selected above. This function is known as the *objective function*.

It is necessary that the size of the subset of secret key bits, given as input to the objective function, is small. This enables us to exhaustively enumerate all values of the subset, and for each value we evaluate the objective function in order to compute the value of chosen state bits. This process is repeated for several plaintext (or ciphertext) values. The value of secret key bits subset for which output of the objective function has greatest statistical correlation with the power dissipated by the circuit, over all the plaintext (or ciphertext) values, is chosen as the actual value of the subset. A DPA attack is a divide-and-conquer attack where the secret key is partitioned into subsets, and the above described procedure is repeated for each subset until the value of all secret key bits are found. For example, the 56-bit DES key is partitioned into seven subsets of eight bits each. Since DPA is one of the most successful side-channel attacks used in practice to get secret keys used in cryptogrpahic algorithms, we will present more details of this attack. Figure 2.1 illustrates the steps of this attack. The basic methodology of a DPA attack is as follows:

1. In order to enable the statistical correlation analysis, we need to collect multiple traces of power dissipated by the circuit while it is performing the cryptographic computa-

tion. So simulate the circuit with t plaintexts $\{P_1, P_2, \ldots, P_t\}$, record the corresponding ciphertexts $\{C_1, C_2, \ldots, C_t\}$, and the power traces $\{T_1, T_2, \ldots, T_t\}$. Each trace T_i is a collection of M discrete signals sampled over the entire period of computation of the cryptographic algorithm, *i.e.*, $T_i = \{T_i^1, T_i^2, \ldots, T_i^M\}$, $1 \leq i \leq t$.

2. Since DPA is a divide-and-conquer approach, partition the N-bit secret key into m equal-sized subsets, $\{K_1, K_2, \ldots, K_m\}$, where $|K_j| = l = N/m$, $1 \leq j \leq m$. $|K_j|$ indicates the number of bits in subset K_j.

3. Choose a secret key subset K_j from the set of secret key partitions. Select a state bit, S_j, whose value can be computed from two input parameters: plaintext P_i (or ciphertext C_i) and K_j using the *objective function* $F()$, *i.e.*, $S_j = F(K_j, X)$, $1 \leq j \leq m$, where X is either P_i or C_i, $1 \leq i \leq t$. In Figure 2.1, we assume the objective function takes the plaintexts as inputs.

4. The l-bit value K_j, $1 \leq j \leq m$, takes values from zero to L where $L = 2^l - 1$. Let v be a value of K_j where $0 \leq v \leq L$. For each value v of K_j, do the following,

   - For each value v, compute value of the state bit S_j for all the t plaintexts (or ciphertexts). The t values of the state bit are represented by the set $\{S_j^1(v)$, $S_j^2(v)$, $\ldots, S_j^t(v)\}$ where $S_j^i(v) = F(v, X)$ and X is either P_i or C_i, $1 \leq i \leq t$ (columns of the table in Figure 2.1).

   - Based on the values of the state bit in the set $\{S_j^1(v), S_j^2(v), \ldots, S_j^t(v)\}$, we generate two partitions, $S_j^v(0)$ and $S_j^v(1)$ where $S_j^v(0) = \{T_i,\, 1 \leq i \leq t | S_j^i(v) = 0\}$ and $S_j^v(1) = \{T_i,\, 1 \leq i \leq t | S_j^i(v) = 1\}$ such that $|\{S_j^v(0)\}| + |\{S_j^v(1)\}| = t$.

   - The power traces in sets $S_j^v(0)$ and $S_j^v(1)$ are averaged, (at each of the M sample points), and the difference of the two average traces is taken to produce a *differential trace*, DT_j^v. Note that the differential trace has M samples.

5. By iterating through all possible values for the l-bit secret key subset K_j, we get L differential traces DT_j^v, $0 \leq v \leq L$. Only the differential trace corresponding to the

correct value of K_j exhibits an appreciable spike, while the remaining traces show small noisy spikes of roughly the same amplitude. This is illustrated in Figure 2.1 where the actual value of key subset K_j is equal to p.

6. We compute the N-bit secret key by repeating the steps 3, 4 and 5 for all the m secret key subsets. Thus, the complexity of key search is reduced from $2^N - 1$ to $m \cdot (2^l - 1)$.

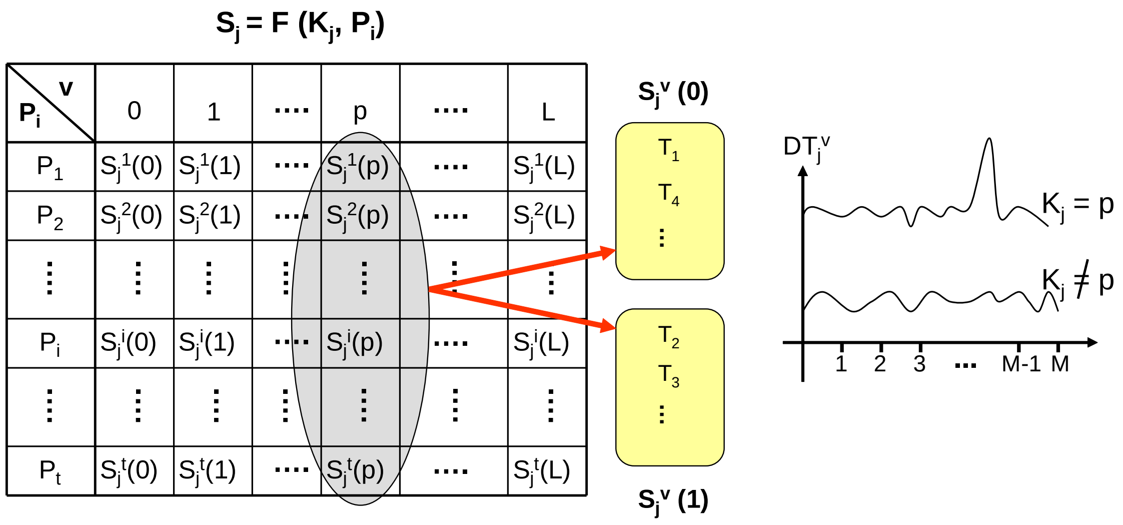


Figure 2.1: DPA attack

Now, we provide some rationale for the behavior of step 5. The set of power traces are partitioned into two sets based on the value of a state bit s (computed by the objective function). During the computation of the cryptographic algorithm, suppose the state bit s flips from 1 to 0 (or 0 to 1) at time (or sample) T in the power trace. Therefore, one set has power traces with a spike at time T due to s being set to 1, while traces in the other set have a smaller spike at time T as a result of s being set to 0 (power consumed by the 0-to-1 transition is more than for the 1-to-0 transition). Given a sufficiently large number of traces in each set, the differential trace will exhibit a spike at time T that represents the difference in power consumption when state bit s is set to 1 and 0. Also, this substantial peak at time T is observed only when the objective function correctly calculates the value of s, which in turn happens when the correct guess is made for the key bits. At any time $T' \neq T$, a state bit s' other than s toggles between 0 and 1. Since partitioning of the traces was conditioned on the value of s, both the sets contain some traces with s' set to 1 and others traces with s' set to 0. In the differential trace, peaks at any time $T' \neq T$ appear as noise. Similar reasoning applies for appearance of noisy peaks in differential traces computed for incorrect values of the key subset. Thus, the spike at time T will be much longer than the spikes at all other times.

Since one of the inputs to the objective function is the plaintext (or ciphertext), only the state bits in the first or last round can be used for partitioning the power traces. Consider the last round of DES shown in Figure 2.2. We assume that we know the ciphertext values *Ciphertext[0:31]* and *Ciphertext[32:63]*. The objective function calculates the value of a bit in register $L16$ based on the values of the known ciphertext and guesses for a 6-bit subset of the round key $K16$. The transition of the chosen state bit in register $L16$ contributes to the power dissipated by the circuit. So the attack on DES in based on the observation that only for the correct guess of the secret key subset, the differential trace has peak which represents power dissipated by the chosen state bit. In Figure 2.2, by using the known value *Ciphertext[32:63]*, and guess for a subset of key $K16$, we can compute a subset of the output of F function. By XORing the F function output with *Ciphertext[0:31]*, we obtain the value of the target state bit in $L16$. Similarly, an objective function can be formulated to calculate a state bit in the first round of DES using the known plaintext values. Most of the DPA research has focussed on improving the statistical analysis methods used to extract the key bits from the power traces.

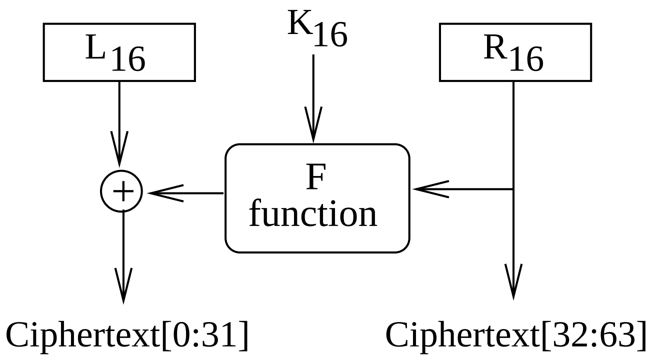


Figure 2.2: Last round of DES

In addition to timing and power, the presence of faults in circuits performing cryptographic computations have been used as a source of side-channel attacks [49, 50]. Fault attacks exploit the presence of faults in the implementation of cryptographic algorithms to find out the secret key. Transient faults resulting in temporary flipping of bits in a circuit are caused in a variety of ways, like variations in supply voltage, high temperature, clock jitters, ion beams, *etc.* Optical fault injection techniques exist which can be used to set or reset any individual bit of an SRAM in a microcontroller [51]. A plaintext encrypted with the same key produces two different ciphertexts in the presence and absence of a fault, respectively. Fault analysis attacks compare these two ciphertexts to extract information about the secret key. One of the popular fault based attacks is the differential fault analysis (DFA) attacks. DFA attacks are targeted toward secret key cryptosystems, like DES, AES, *etc.* [50]. We explain the working of a DFA attack using DES as an example. A single-bit transient fault is assumed to occur in *any* bit position of a register in DES computation in a *random* round. The attacks proceed in two stages. First, trial and error-based analysis is done based on the knowledge of the structure of the algorithm to find out the location of the bit flip. Next, the S-boxes lying along the path of propagation of the error from its fault site to the output (*i.e.*, ciphertext) are targeted for differential cryptanalysis-based attacks for deducing a subset of the secret key bits.

Assume that a single-bit fault is present in the last round of DES, as shown in Figure 2.3. E represents the expansion permutation which expands the 32-bit $R16$ to a 48-bit value, which is XORed with the 48-bit round key, $K16$. The 48-bit result is broken into eight 6-bit chunks which are input to the S-boxes, $(S1, S2, \ldots, S8)$. An S-box substitutes a 6-bit input with a 4-bit output. Thus, the eight S-boxes (collectively termed $SUBSTITUTE$) produce a 32-bit output. P permutes the 32-bit output of the $SUBSTITUTE$ operation. The ciphertext comprises two 32-bit components, L_{out} and R_{out}. From the figure, we can see that

$$Lout \;=\; P(SUBSTITUTE(K16 \oplus E(R16))) \oplus L16$$

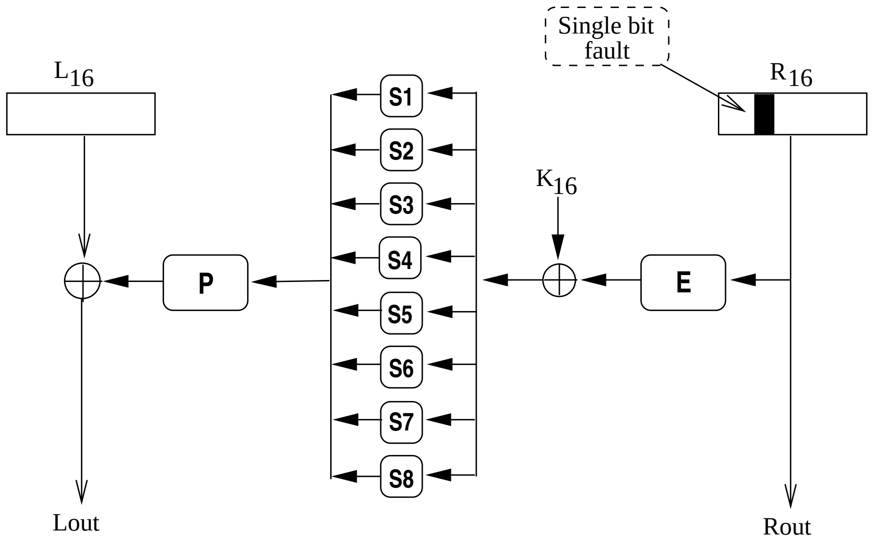


Figure 2.3: Last round of DES

$$= \quad P(SUBSTITUTE(K16 \oplus E(Rout))) \oplus L16 \text{ (since, Rout = R16)}$$

Assume there is a single-bit transient fault in $R16$ which produces erroneous, $\overline{Lout}$ and $\overline{Rout}$. We can write $\overline{Lout}$ as

$$\overline{Lout} = P(SUBSTITUTE(K16 \oplus E(\overline{Rout}))) \oplus L16$$

When we XOR $Lout$ and $\overline{Lout}$, we get,

$$Lout \oplus \overline{Lout} = P(SUBSTITUTE(K16 \oplus E(Rout))) \oplus P(SUBSTITUTE(K16 \oplus E(\overline{Rout})))$$

In the equation above, we know all values except the round key $K16$. When a single bit fault is present in $R16$, then only one (or two) S-boxes are affected, which in turn means that the above equation can be applied to 6-bit subsets of $Lout$, $Rout$, $\overline{Lout}$, $\overline{Rout}$, and $K16$. This allows us to enumerate all the 64 possible values of the 6-bit subset of $K16$, and narrow down the candidates that satisfy the above equation. The correct value of the 6-bit subset is identified by intersecting the candidate sets produced for multiple single-bit faults. By repeating the experiment for each of the remaining 6-bit subsets, the 48-bit round key $K16$ can be identified. After finding out the last round key, $K16$, the remaining eight bits of the 56-bit DES secret key can be found out by exhaustive enumeration.

Another well-known fault attack was directed against modular exponentiation based cryptosystems by Boneh *et al.* [49]. They demonstrated how the presence of a fault during Chinese remainder theorem (CRT) computation in RSA decryption can reveal the secret bits. Let p and q be the two prime factors of an RSA modulus n [3]. Assume that there was a fault which led to incorrect mod p computation. This leads to the output of the CRT being correct with respect to mod q, but incorrect with respect to mod p. Thus, the difference between the correct result and incorrect result will be a multiple of q. This allows the adversary to find out q by computing the GCD of this difference and modulus n.

In general, studies have been carried out to investigate the effectiveness of side-channel attacks against cryptosystems. Kelsey *et al.* [52] proved the power of side-channel attacks by demonstrating that a minimal amount of side-channel information is required for breaking some popular cryptographic algorithms. Coron *et al.* [53] formulated a set of statistical tests which can be used to detect the presence of side-channel leakage from any given cryptographic computation. In chapter 5, we formulated a new side-channel attack based on Boolean reasoning techniques from artificial intelligence [54, 55]. This work proposes a side-channel attack framework to exploit the leakage of values of intermediate variables in the software computation of cryptographic algorithms in order to get the secret key using a Satisfiability (SAT) solver. This work was the first instance of use of Boolean reasoning tools for the purpose of launching side-channel attacks, and is complementary to existing side-channel attacks based on statistical techniques. Prior to this work, there were some works which employed formal methods for mathematical cryptanalysis of cryptographic algorithms. However, this work differs from them in terms of recognizing and demonstrating the effectiveness of SAT as a tool for enabling side-channel attacks. Schaumuller-Bichl [56] introduced the method of formal coding in which XOR sum-of-product expressions are formulated for the DES output bits in terms of the plaintext and key bits. For a known plaintext and ciphertext pair, the equations are solved to get the key bits. However, the high complexity of the resulting equations limited the attack to being a theoretical one. Massacci and Marraro [57] proposed the the use of automated reasoning techniques for cryptanalysis.

As an example, they illustrated the modeling of DES algorithm as a propositional formula, and using a SAT solver to search for the secret key used. The focus of their work was two-fold. First, use cryptographic instances as means to generate solved instances of SAT problems which are difficult to solve. This would provide benchmarks for measuring the effectiveness of SAT algorithms. Second, they propose usage of SAT formulation to test for cryptographic properties of a cipher like, presence of trapdoors, check for existence of weak keys *etc.* Toward the latter goal, they performed few preliminary experiments on the effectiveness of SAT as a tool for cryptanalyzing Feistel ciphers, like DES. However, instances of DES algorithm with more than three rounds proved intractable for the SAT solver.

In this chapter, we presented a summary of work done in regard to improving performance of cryptographic algorithms on embedded systems, studying the effect of cryptographic processing on energy consumption of embedded systems, and on side-channel attacks directed at embedded devices. In context of embedded systems, performance of cryptographic algorithms is much better understood than energy cost of cryptographic processing, and side-channel attacks. However, with regards to performance, new developments in embedded processor architectures have enabled novel ways of optimizing cryptographic algorithm processing. Thus, in the subsequent chapters of this thesis, we present our work done on energy consumption analysis of cryptographic algorithms, leveraging an extensible and configurable processor to optimize embedded IPSec protocol execution, and side-channel attack framework based on a SAT solver.

Chapter 3

Energy Analysis of Security Protocols and Cryptographic Algorithms on an Embedded System

Security is becoming an everyday concern for a wide range of electronic systems that manipulate, communicate, and store sensitive data. An important and emerging category of such electronic systems is battery-powered mobile appliances, such as personal digital assistants (PDAs) and cell phones, which are severely constrained in the resources they possess, namely, processor, battery and memory. This work focuses on one important constraint of such devices – battery life – and examines how it is impacted by the use of various security mechanisms.

In this work, we first present a comprehensive analysis of the energy requirements of a wide range of cryptographic algorithms that form the building blocks of many security protocols. We then study the energy consumption requirements of the most popular transport-layer security protocol: Secure Sockets Layer (SSL). We investigate the impact of various parameters at the protocol level (such as cipher suites, authentication mechanisms, and transaction sizes, etc.) and the cryptographic algorithm level (cipher modes, strength) on the overall energy consumption for secure data transactions. To our knowledge, this is the first comprehensive analysis of the energy requirements of SSL.

For our studies, we have developed a measurement-based experimental testbed that consists of an iPAQ PDA connected to a wireless local area network (LAN) and running Linux, a PC-based data acquisition system for real-time current measurement, the OpenSSL implementation of the SSL protocol, and parametrizable SSL client and server test programs. Based on our results, we also discuss various opportunities for realizing energy-efficient implementations of security protocols. We believe such investigations to be an important first step towards addressing the challenges of energy-efficient security for battery-constrained systems. This chapter contains results presented in [16, 17].

3.1 Introduction

Today, an increasing number of battery-powered embedded systems – PDAs, cell phones, networked sensors, and smart cards, to name a few – are used to store, access, manipulate, or communicate sensitive data, making security an important issue. Security concerns in such systems range from user identification, to secure information storage, secure software execution, and secure communications. Most battery-powered systems contain wireless communication capabilities for untethered operation, introducing new security concerns due to the public nature of the physical communication medium or channel.

With the evolution of the Internet, network and communications security has gained significant attention [3,4,58,59]. Secure communication across wired and wireless networks is typically achieved by employing security protocols at various layers of the network protocol stack (*e.g.*, WEP [60] at the link layer, IPSec [61] at the network layer, TLS/SSL [62] and WTLS [63] at the transport layer, SET at the application layer, *etc.*). The building blocks of a security protocol are cryptographic algorithms, which are selected based on the security objectives that are to be achieved by the protocol. They include asymmetric and symmetric encryption algorithms, which are used to provide authentication and privacy, as well as hash or message digest algorithms that are used to provide message integrity.

While security protocols and the cryptographic algorithms they contain address security considerations from a functional perspective, many embedded systems are constrained by the environments they operate in and the resources they possess. For such systems, there are several challenges that need to be addressed in order to enable secure computing and communications. For battery-powered embedded systems, perhaps one of the foremost challenges is the mismatch between the energy and performance requirements of security processing,[1] and the available battery and processor capabilities. Rapid increases in communication data rates and security levels required, together with slow increases in battery capacities, threaten to widen this "battery gap" to a point where it will impede the adoption of applications and services that require security.

In this work, we demonstrate that security processing can have a significant impact on battery life. *Addressing the battery gap in secure communications requires that we first analyze and understand the energy consumption characteristics of security protocols and cryptographic algorithms.* This chapter presents comprehensive energy measurement and analysis of the most popular transport-layer security protocol used in the Internet, the SSL or Transport Layer Security (TLS) protocol. The energy analysis in this study is performed by executing secure data transactions on a battery-powered system (a Compaq iPAQ PDA [64]), measuring the current drawn from the power supply, and calculating the energy consumed during the time intervals in which the security protocol or its constituent cryptographic algorithms are executed. Our results can be used to explore the impact of various parameters, at the protocol and cryptographic algorithm levels, on overall energy consumption for secure data transactions. Based on our analysis, we discuss various opportunities for energy-efficient implementations of security protocols.

The rest of this chapter is organized as follows. Section 3.2 motivates the need for addressing energy consumption issues in security protocols. Section 3.3 introduces the reader to pertinent security terms and concepts. Section 3.4 describes the experimental

[1]We use the term *security processing* to refer to any computations performed for the sake of security, including the execution of security protocols and cryptographic algorithms.

testbed used in our work to execute and analyze secure wireless transactions, and provides details of the energy measurement setup. Section 3.5 presents the results of our energy measurements, applies this information to analyze the SSL protocol, and suggests ways of optimizing the energy requirements of SSL. Section 3.6 summarizes the insights gathered in this work.

3.2 Motivation

In this section, we provide an example to motivate the need for studying the energy consumption of security protocols. Then, we discuss the possible ways in which the energy consumption of security protocols can be optimized. Finally, we conclude by surveying related work on performance/energy analysis and optimization of security protocols.

3.2.1 The impact of security on battery life: An illustration

In order to illustrate the burden imposed by security processing, we consider an example involving the operation of a sensor node with and without the need for security. We show that computations resulting from the use of security algorithms significantly reduce the amount of energy available for normal operations of the node.

One of the most important examples of embedded systems are sensor nodes. Sensor nodes are normally used for aggregating specific information about their surroundings and transmitting it to a centralized location. Figure 3.1 shows the architecture of a sensor node. As shown in the figure, a sensor node has three components: a sensing component, a processing component, and a communication component. The sensing component has a sensor which gathers information about its surroundings, and a Analog-to-digital converter (ADC) which transforms the analog sensed data to its equivalent digital representation. Usually, the sensor node applies some transformations like, compression, encryption *etc.*, to the sensed data. These operations are applied by the processing unit which consists of processor and memory. In the communication component we have a transceiver to send and receive signals. All these three components of a sensor node are powered by a battery.

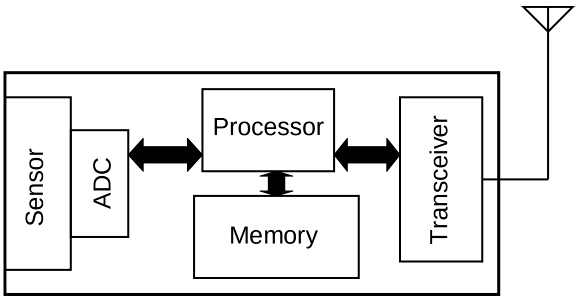


Figure 3.1: Architecture of a sensor node

The sensor node used in our motivational example uses a Motorola "DragonBall" MC68328 processor, operates at a data rate of 10Kbps, and has a battery capacity of $26KJ$ (typical). Studies [7] have shown that the node consumes $13.9\mu J$ and $21.0\mu J$, for receiving and transmitting a bit, respectively. When encryption is used, the node consumes $41.0\mu J$ per bit during asymmetric algorithm operation, and expends $7.9\mu J$ per bit for symmetric algorithm operation. In the absence of encryption, the node transmits the collected data as is. However, when encryption is used, the data transfer is broken up into *sessions.* Each session comprises two stages: authentication and key-establishment using an asymmetric algorithm, followed by transmission of data after encrypting them using a symmetric algorithm with the key established (in the first stage of the session). The amount of data transmitted in a session is referred to as *session length.* The maximum amount of data which can be transferred in each session, *i.e.,* upper bound on the session length, is specified by the security policy, and is determined by the sensitivity of the data; the greater the sensitivity of the data, the shorter the session length, and vice versa. Thus, in the case of sensitive data, session lengths are short, and the frequency of setting up new sessions is high. This is illustrated in Figure 3.2 where the x-axis gives the session length, and the y-axis gives the frequency of session set-up (until the battery runs out). For example, when the session length of 1KB is used, the node has to negotiate nearly five new sessions per second. For highly sensitive data, session lengths are made less than or equal to 4KB and, consequently, greater than or equal to two sessions have to be set up every second.

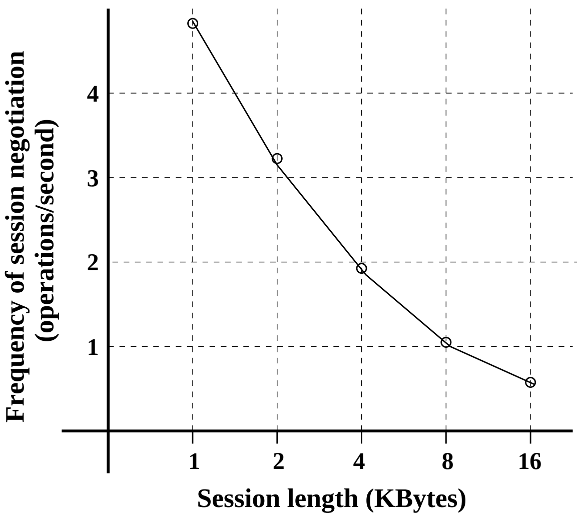


Figure 3.2: Frequency of session set-up as a function of session length

Figure 3.3 shows the effect of encryption on battery life as the session length is varied. "Battery life (encrypted)" refers to the number of sessions which can be transacted before the battery drains out. Similarly, "Battery life (unencrypted)" is the multiple of session lengths of data that can be sent without encryption until the battery drains out. We assume the session length to be the same in both the cases. The y-axis in Figure 3.3 is the ratio of "Battery life (encrypted)" to "Battery life (unencrypted)," and is an indicator of how fast the battery gets drained in the presence of encryption. The energy consumption values were obtained by multiplying the session lengths by energy consumed per byte by the Dragonball processor for performing modular exponentiation [7]. The figure shows that security processing causes an appreciable reduction in battery life. For example, when security requirements are high (and sessions are made less than or equal to 4KB), we see that the battery runs out more than twice as fast as when there is no encryption. Thus, there is a strong motivation to investigate techniques which lead to energy-efficient execution of security protocols.

3.2.2 Energy-efficient security protocols

The objective of energy-efficient security protocol execution can be achieved in multiple ways, which can be divided into two broad classes listed below.

- By making the execution of constituent cryptographic algorithms (also referred to as

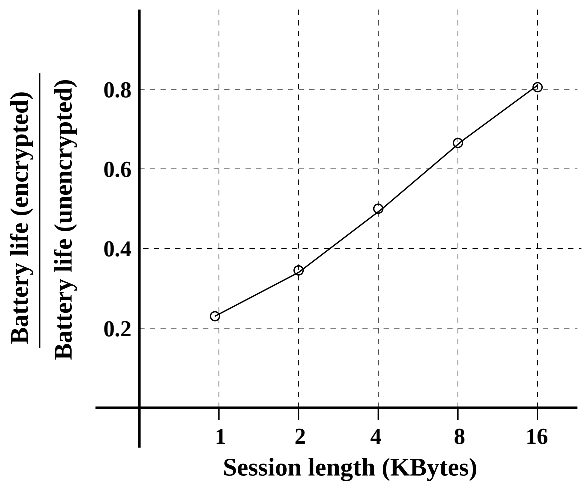


Figure 3.3: Effect of encryption on battery life

cryptographic primitives) efficient through a combination of hardware and software techniques [23,25,26,65,66], we can improve the performance and energy requirements of security protocols. Usually, in these techniques, there is an overhead in the form of increase in silicon area or more complex software.

- We can make the security protocols energy-cognizant, by allowing them to alter their operation depending on the operating environment. This adaptation of behavior is guided by rules, which determine the best possible alternative with respect to energy efficiency, under any given input conditions. These changes may involve a conscious and a conservative tradeoff between the level of security and energy.

In either scenario, the challenges of energy-efficient secure communications can be better addressed if energy requirements and bottlenecks of the underlying security protocols are better understood. Commonly used security protocols, like SSL/TLS, IPSec, *etc.*, have the freedom of realizing the desired security objectives by choosing specific cryptographic algorithms from a pre-defined set. In addition, the communicating parties can also decide upon parameters which influence the mode of operation of the chosen cryptographic algorithm. These provisions are made in security protocols primarily to lend flexibility to interactions between parties having diverse capabilities, in terms of the number of cryptographic algorithms supported by each of them (usually, resulting from the usage of different versions of security protocol software).

In this work, we perform a detailed analysis of the energy requirements of various cryptographic primitives, with the intention of using this data as a basis for devising energy-efficient security protocols. We performed several experiments where we varied several protocol and cryptographic algorithm-level parameters and observed the impact on energy. Based on these experiments, we gathered series of observations on ways to optimize the energy consumption of network security protocols by adapting their operational characteristics. We exemplified our energy optimization strategies using SSL protocol. However, our results are broadly applicable to any security protocols which involve a client and a server negotiating cryptographic parameters using handshake based on public-key algorithms, and protecting the data exchanged using symmetric and hash algorithms.

3.3 Preliminaries

In this section, we provide a brief overview of commonly employed security concepts and terminology [3,4]. We begin by defining the widely used terms in the fields of cryptography and network security, and follow it by describing different kinds of protection measures, referred to as *security objectives*, desired in practical applications with a need for security. The concern for security in practice is addressed by choosing a security protocol, which achieves all the required security objectives. Security protocols realize the security objectives through the use of appropriate cryptographic algorithms. In the latter part of the section, we define the three classes into which all the cryptographic algorithms can be categorized based on their characteristics, and conclude the section by illustrating the working of a widely used security protocol, SSL.

3.3.1 Basic security terminology

A message present in a clear form, which can be understood by any casual observer, is known as the *plaintext*. The *encryption* process converts the plaintext to a form that hides the meaning of the message from everyone except the valid communicating parties, and the result is known as the *ciphertext*. *Decryption* is the inverse of encryption, *i.e.*, the ciphertext is mapped back to its corresponding plaintext. The processes of encryption and decryption are parameterized on a quantity known as the *key*, which is ideally known only to the legitimate communicating parties. Since the strength of a security scheme depends on the secrecy of the key(s) used, it is highly imperative that the communicating parties take utmost precaution to safeguard the keys belonging to them. There are three kinds of cryptographic algorithms: symmetric, asymmetric and hash. A *security protocol* formally specifies a set of steps to be followed by two or more communicating parties, so that the mutually desired security objectives are satisfied. It is assumed that the parties involved have the means to execute the various steps of the security protocol. A protocol uses cryptographic algorithms as its building blocks to realize a combination of four security objectives, namely: confidentiality, authentication, integrity, and non-repudiation.

3.3.2 An example security protocol: SSL

SSL is one of the most widely used security protocols on the Internet. It is implemented at the transport layer of the protocol stack. SSL offers the basic security services of encryption, source authentication, and integrity protection, for data exchanged over underlying unprotected networks. The SSL protocol is typically layered on top of TCP/IP layers of the protocol stack, and is either embedded in the protocol suite or is integrated with applications such as browsers. The SSL protocol consists of two main layers, as shown in Figure 3.4. The SSL record protocol provides the basic services of confidentiality and integrity to the higher-layer protocols: SSL handshake, SSL change cipher and SSL alert. Let us now examine how the SSL record protocol is used to encrypt application data. The first step involves breaking the application data into smaller fragments. Each fragment is then compressed, if compression options are enabled. The next step involves computing a message authentication code (MAC), which facilitates message integrity. The compressed message plus MAC is then encrypted using a symmetric cipher. If the symmetric cipher is a block cipher, then a few padding bytes may be added. Finally, an SSL header is attached to complete the assembly of the SSL record. The header contains various fields including the higher-layer protocol used to process the attached fragment.

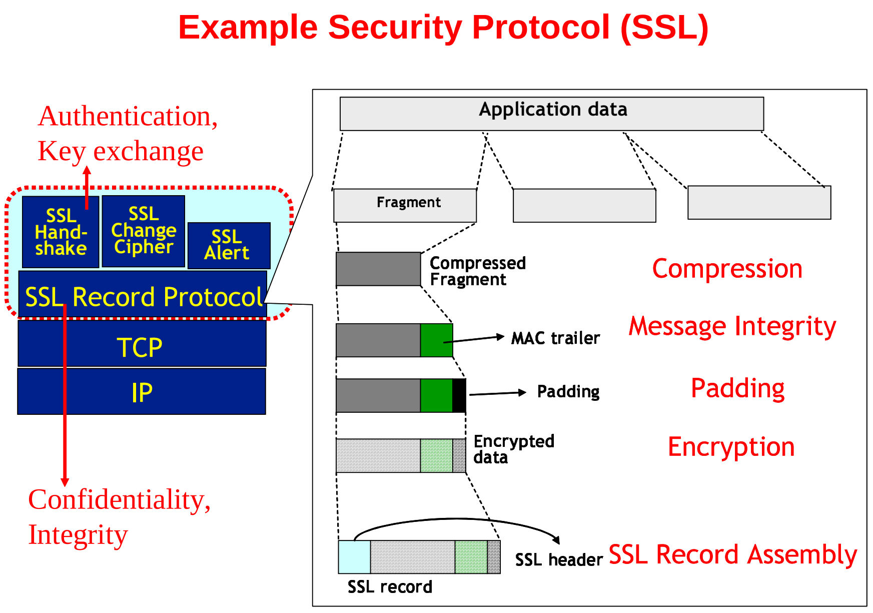


Figure 3.4: The SSL protocol, with an expanded view of the SSL record protocol

Of the three higher-layer protocols, SSL handshake is the most complex and consists of a sequence of steps that allows a server and client to authenticate each other and negotiate the various cipher parameters needed to initiate a session. For example, the SSL handshake is responsible for negotiating a common suite of cryptographic algorithms (cipher-suite), which can then be used for session key exchange, authentication, bulk encryption and hashing. The cipher-suite RSA-3DES-SHA1, for example, indicates that RSA can be used for key agreement (and authentication), while 3DES and SHA1 can be used for bulk encryption and integrity computations, respectively. More than 30 such cipher suite choices exist in the OpenSSL implementation [67] of the SSL protocol, resulting from combinations of various cipher alternatives for implementing the individual security services.

Finally, the SSL change cipher protocol allows for dynamic updates of cipher suites used in a connection, while the SSL alert protocol can be used to send alert messages to a peer. Further details of the SSL protocol can be found in [4].

3.4 Experimental setup

Figure 3.5 describes the experimental setup used to execute secure client-server interactions, and the testbed developed to quantify the energy consumption of the various constituent security protocols.

The experimental setup for secure client-server communication consists of a client that connects to a LAN through a wireless access point, while the server is a PC that is wired to the LAN. The handheld used in the experiment is a Compaq iPAQ H3670, which contains an Intel SA-1110 StrongARM processor clocked at 206MHz. It is provided with 64MB of RAM and 16MB of FlashROM, and has an expansion sleeve which allows for memory expansion using compact flash cards. It connects to the wireless access point using a Cisco Aironet 350 series WLAN card. The handheld also supports additional communication capability through a serial port, a USB port and IrDA at 115.2 Kbps. It is powered by a Li-Polymer battery with a 950 mAh rating. The handheld uses the Familiar distribution [68] of Linux as its operating system (OS). The server is a PC equipped with a 700MHz Intel Pentium III having 256MB of RAM and running the RedHat Linux OS. The security of client and server interactions is provided by the SSL software from the OpenSSL [67] open-source project.

The energy consumption values for individual cryptographic algorithms are obtained by running their implementations on the client, and measuring the current drawn from the power supply. Figure 3.5 also shows the arrangement used for measuring the energy consumption of the cryptographic algorithms. The energy measurement is done using LabVIEW [69], a GUI-based data acquisition, measurement analysis, and presentation software. The data acquisition software runs on a PC (called a power measurement system), which is also directly connected to the handheld through its serial port. This enables the handheld to send synchronization signals to the data acquisition unit to start and stop the energy measurements. This signaling mechanism allows us to precisely measure the energy dissipated by the chosen software kernels. The current drawn by the client is measured by connecting a sense resistor in series between the handheld and the energy source, *i.e.*, the battery. The voltage drop across the sense resistor is measured using an SCB-68 I/O con-

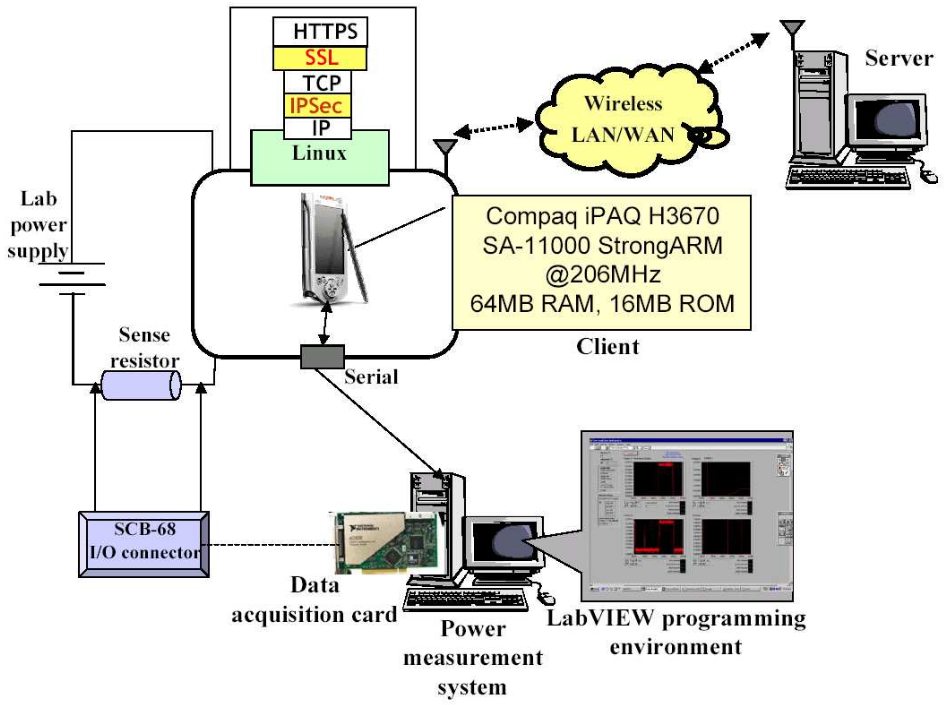


Figure 3.5: Secure client-server configuration and the energy measurement testbed

nector block [69]. This block interfaces to the data acquisition software, LabVIEW, through a data acquisition card in the PC running the LabVIEW software. LabVIEW is used to calculate the energy supplied to the handheld by integrating power over the time interval between the start and stop synchronizing signals.

3.5 Experimental results

In this section, we present a comprehensive empirical analysis of the energy consumption characteristics of cryptographic algorithms (Section 3.5.1) using the experimental set-up described in Section 3.4. We also present a comprehensive energy analysis for various stages of the SSL protocol (Section 3.5.2).

3.5.1 Energy analysis of cryptographic algorithms

We analyze variations in the energy consumption of various asymmetric, hash, and symmetric algorithms used for the purposes of authentication, integrity and secrecy of data transactions, respectively (Sections 3.5.1.1 - 3.5.1.3). In Section 3.5.1.4, we investigate the influence of commonly used software implementation techniques of cryptographic algorithms on their energy consumption. We conclude this section by illustrating energy consumption versus security-level trade-offs realized by varying the operational parameters of cryptographic algorithms (Section 3.5.1.5). The implementations of the cryptographic algorithms were obtained from a standard cryptographic library used in the widely deployed OpenSSL package [67], and run on a SA-1110 based energy testbed described in the previous section. Since all the implementations are derived from the same standardized library, we assume that more or less similar software engineering techniques were uniformly employed throughout the software package. Thus, we can consider that the relative computational differences between various algorithms belonging to the same class (asymmetric, hash, and symmetric), are largely due to disparities in the complexity of their constituent operation steps, and are not strongly tied to the data structures and software programming techniques used. In addition, most of the hardware platforms used in embedded devices are either SA-1110 based or very similar to it. Therefore, the conclusions drawn here are broadly applicable to other protocols such as WTLS, IPSec, *etc.*, since they use the same cryptographic algorithms. However, there are some limitations to this approach arising out of usage of particular number-theoretic algorithms. For each number-theoretic operation, there exist many different algorithms to perform it, and they have varying performance. Though, the library chooses the optimal mathematical algorithms for the various operations, the state-of-the-art keeps changing. For example, modular multiplication is the basic operation in modular exponentiation-based public-key algorithms, and in our experiments, we employed Montgomery multiplication which is one of the most efficient algorithms available. However, if a different algorithm is employed, then our results would differ, in the best case, by a scaling factor. Thus, the observations from our empirical analysis would be relevant in the context of the algorithms employed for different mathematical operations.

3.5.1.1 Asymmetric algorithms

Computationally hard mathematical problems form the basis of public-key cryptosystems. RSA is based on the hardness of integer factorization, while digital signature algorithm (DSA) and Diffie-Hellman (DH) are based on that of the discrete logarithm problem in integer fields. Elliptic curve digital signature algorithm (ECDSA) and elliptic curve Diffie-Hellman algorithm (ECDH) provide security based on the discrete logarithm problem defined on elliptic curves. The basic mathematical operation in RSA, DSA and DH is modular exponentiation, and point multiplication on elliptic curves forms the core arithmetic operation in ECDSA and ECDH [70]. If $Security_{integer}(k1)$ and $Security_{elliptic}(k2)$ denote the security provided by RSA/DSA/DH algorithms employing a key of $k1$ bits and ECDSA/ECDH algorithms using a key of size $k2$ bits, respectively, then research has shown that the following equivalence exists between them [71]: $Security_{elliptic}(163) \equiv Security_{integer}(1024)$, $Security_{elliptic}(283) \equiv Security_{integer}(3072)$, and $Security_{elliptic}(409) \equiv Security_{integer}(7680)$. For example, a 1024-bit modulus RSA offers the same level of protection from cryptanalytic attacks as a 163-bit ECDSA.

Table 3.1: Energy cost of digital signature algorithms

Algorithm	Key size (*bits*)	Key generation (*mJ*)	Sign (*mJ*)	Verify (*mJ*)
RSA	1,024	270.13	546.50	15.97
DSA	1,024	293.20	313.60	338.02
ECDSA	163	226.65	134.20	196.23
ECDSA	193	281.65	166.75	243.84
ECDSA	233	323.30	191.37	279.82
ECDSA	283	504.96	298.86	437.00
ECDSA	409	1034.92	611.40	895.98

Table 3.1 compares the energy consumed by the three federal information processing standard (FIPS)-approved asymmetric algorithms for generating and verifying signatures in security protocols: RSA, DSA and ECDSA. We show the energy consumed by ECDSA for different elliptic key sizes, in addition to 1024-bit RSA and DSA. The energy values are reported for the three main steps associated with digital signature algorithms: key generation, signature creation (sign) and signature verification (verify). We assume *a priori* generation of the parameters used in the key generation process, as is the case in resource-constrained devices. We can see that 163-bit ECDSA is energy-efficient compared to 1024-bit DSA.

However, 163-bit ECDSA and 1024-bit RSA digital signature algorithms have complementary energy costs. RSA performs signature verification efficiently, while ECDSA imposes a smaller cost for signature generation. We can see that the energy costs of sign and verify are much more symmetric in ECDSA than in RSA. ECDSA uses point multiplication where a scalar of the order of the degree of the curve is multiplied with a fixed point (called the base point) on the elliptic curve to get another point on the curve. Point multiplication is employed by both the sign and verify operations. In ECDSA, the verify operation requires some extra steps involving modular multiplication, for validating the signature, and therefore consumes more energy than the sign operation. The huge discrepancy in the energy costs of sign and verify operations in RSA results from the significant difference in the sizes of the keys employed (which are used as exponents in the modular exponentiation operation). In the sign operation, the private key is used which has the same size as the modulus, and the much smaller public key (it is usually 3 or 17 due to them having fewer number of ones in their binary representation) is used in the verify operation.

Asymmetric algorithms are also widely used for performing key exchange. Table 3.2 compares the standard algorithms used for key exchange, Diffie-Hellman (DH) and its elliptic curve analogue (ECDH). We observe that a 163-bit ECDH consumes much less energy than a 1024-bit DH key exchange. The energy cost of the DH algorithm can be drastically reduced by decreasing the size of keys from 1024 bits to 512 bits. However, this benefit does come at the cost of reduced security.

Table 3.2: Energy cost of key exchange algorithms

Algorithm	Key size ($bits$)	Key generation (mJ)	Key exchange (mJ)
DH	1,024	875.96	1,046.5
ECDH	163	276.70	163.5
DH	512	202.56	159.6

Modular exponentiation and point multiplication can be done in different ways [70]. In the implementations used in our experiments, modular exponentiation was performed by combining Montgomery reduction with a sliding window exponentiation. The window size was set at 6 bits. There were two alternatives available for realizing point multiplication: Montgomery without precomutation (MWP) method, and width-w non-adjacent (wNAF) method. We studied the energy consumption of the two algorithms and found that the wNAF method is efficient only when multiple point multiplications are performed with respect to the same point.

3.5.1.2 Hash algorithms

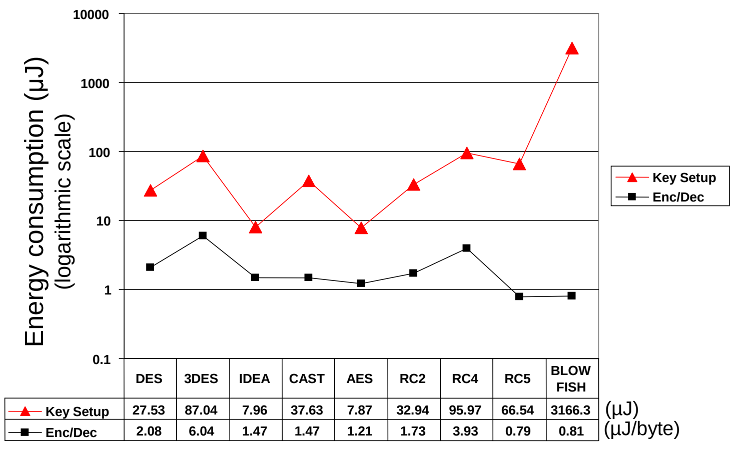
Table 3.3 summarizes the energy cost of commonly-used hashing algorithms. In general, hash algorithms are the least complex of the cryptographic algorithms, and should intuitively incur the least energy cost. From Table 3.3, MD2 and HMAC are observed to be more compute-intensive than the rest of the hash algorithms. MD2 is byte oriented, while MD4 and MD5 are word oriented and much faster on 32-bit processors. In MD2, the input is broken into 16 byte blocks which are transformed using a function derived from digits of pi, and the transformed result is mangled to produce the output. HMAC is a keyed hash, and as the bit-width of the key is increased from 0 (no key) to 128 bits, the energy cost varies by a very small amount. SHA and SHA1 are newer hash algorithms, and have a larger number of steps than MD4 and MD5. Also, SHA and SHA1 are supposed to have better collision resistance, *i.e.*, probability of two inputs mapping to the same hash value, than MD4 and MD5. These benefits of SHA (and SHA1) come at the cost of a slightly higher energy cost than MD4 and MD5.

Table 3.3: Energy consumption characteristics of hash functions

Algorithm	MD2	MD4	MD5	SHA	SHA1	HMAC
Energy (μJ/B)	4.12	0.52	0.59	0.75	0.76	1.16

3.5.1.3 Symmetric ciphers

Symmetric ciphers can be chosen from two classes, *block* and *stream*, for use in a security protocol. Block ciphers operate on similar-sized blocks of plaintext and ciphertext. Examples of block ciphers include DES, 3DES, AES, *etc.* Stream ciphers, such as RC4, convert a plaintext to ciphertext one bit (or byte) at a time. Before a block or stream cipher starts the encryption/decryption operation, the input key (usually, 64 bits) is expanded in order to derive a distinct and cryptographically strong key for each round (*key setup*). Encryption or decryption in symmetric algorithms then proceeds through a repeated sequence (rounds) of mathematical computations. We begin by comparing the energy consumption of various symmetric algorithms and go on to examine the energy costs of features that are particular to symmetric algorithms, and their consequences.



	DES	3DES	IDEA	CAST	AES	RC2	RC4	RC5	BLOW FISH	
Key Setup	27.53	87.04	7.96	37.63	7.87	32.94	95.97	66.54	3166.3	(µJ)
Enc/Dec	2.08	6.04	1.47	1.47	1.21	1.73	3.93	0.79	0.81	(µJ/byte)

Figure 3.6: Energy consumption data for various symmetric ciphers

Figure 3.6 shows variations in energy consumption due to the use of different symmetric ciphers. Energy numbers for the key setup phase and energy-per-byte numbers for encryption/decryption phases are shown for each cipher. The results are reported for one specific mode of each block cipher – ECB or electronic code book, where a given plaintext block always encrypts to the same ciphertext block for the same key (the impact of different modes on energy is explored later in Section 3.5.1.4). The only exception is RC4, which is a stream cipher. From the results displayed in Figure 3.6, we make the following observations:

- RC4 is supposed to be a fast and efficient stream cipher, which is suitable for encrypting data in high-speed networking applications. However, we see that it has a significant energy cost for encryption compared to other symmetric ciphers. Further analysis of the operation of the algorithm shows that majority of the energy is consumed in memory accesses resulting from cache misses. In the 3.93 μJ used in encrypting a byte of data, 3.44 μJ (87.53%) is spent in memory-related operations and the remaining 0.49 μJ (12.47%) on computations.

- Blowfish exhibits the greatest contrast between the energy costs of key setup and encryption/decryption: the energy cost of key setup is the highest, while that of encryption/decryption ranks as one of the lowest. Blowfish is a 64-bit cipher which performs encryption using simple operations, and is designed to be efficient on 32-bit processors with a reasonably-sized data cache. On the other hand, key setup is a complex operation involving 521 iterations in which subkey arrays totaling 4,168 bytes are generated. This algorithm is suitable for applications where the key is not changed frequently (thereby allowing the significant overhead of key setup to be amortized by the low encryption cost).

- In terms of energy of key setup and encryption, IDEA is on par with AES. IDEA is a 64-bit cipher where the constituent operations in encryption are performed on 16-bit blocks. IDEA is supposed to have very good cryptanalytic properties, thereby combining efficiency with acceptable security.

- AES has competitive energy costs, and its cryptanalytic properties have been well-studied. The round operations in AES operate on 8-bit data blocks and are amenable to implementation efficiency on 8-bit processors. However, optimizations exist to make AES run extremely fast on 32-bit processors at the cost of some space overhead (upto 4KB) [72]. In this case, the round operations are transformed into table lookups. The AES implementation under study has this optimization (discussed in detail in Section 3.5.1.4). Moreover, the table lookups can be done in parallel and this feature can be exploited by multi-threaded processors to get further gains in performance.

The encryption energy overhead of symmetric algorithms is also considerably influenced by their operational characteristics, like key size, cipher mode, *etc.* These effects are investigated in a later section.

3.5.1.4 Energy costs of implementation choices

Most symmetric ciphers (block) perform encryption/decryption by passing the input data through multiple iterations of a fixed sequence of operations. This fixed sequence of operations is collectively referred to as a *round*. In software implementations of symmetric ciphers, some characteristics of the operations which make up the *round* are exploited to enhance the performance of the implementation. Two popular techniques employed for improving performance are table look-ups and loop unrolling. In this section, we evaluate the effect of these popular optimizing techniques on the energy consumption by studying them in the context of AES.

Some of the mathematical transformations used in a round can be implemented as pre-determined tables. Table look-ups allow faster execution of the corresponding round operations at the expense of increase in program data size. The tables achieve this by substituting lookups into pre-computed arrays in memory for online computations in the processor. In loop unrolling, code implementing the round operations is expanded across the loop iterations, *i.e.*, the body of the loop is replicated once for every reduction by two times in the number of loop iterations. Loop unrolling increases the number of instructions relative to the branch and overhead instructions present in a loop implementation. This results in a better scheduling of instructions in the processor pipeline, and thus improved performance. However, unrolling results in increase in program code size. In constrained environments, like embedded systems, which usually have a small-footprint on-chip memory, bloating of program size raises the number of capacity misses in the cache, thereby increasing the number of memory accesses which are expensive with respect to energy and performance.

A round in AES consists of four operations, namely ByteSub, ShiftRows, MixColumns and AddRoundKey. Among the four operations, ByteSub is most suitable for being implemented as a table look-up. When performance is an issue, the four operations in a round can be realized as four parallel table look-ups on 32-bit processors [72]. Similarly, the degree of loop unrolling can be varied. In our experiments, we observed the energy consumed in encrypting and decrypting a 60KB data file using a 128-bit key as the number of tables per round is varied (none, one and four), in addition to altering the degree of loop unrolling (none, partial and full). In the case of one table per round, only the ByteSub operation is implemented as a table, and in partial unrolling the loop is unrolled for half the number of times it is in full unrolling. The results of the experiments are shown in Figure 3.7 based on which we make the following observations:

- Full unrolling consumes the maximum energy among the three degrees of unrolling. The increase in code size due to full unrolling has a negative impact on the cache behavior, thereby resulting in an increase in expensive memory accesses. The presence of table look-ups further increases the memory traffic, and therefore we see energy consumption increasing with the number of tables.

- Partial unrolling gives the most energy-efficient behavior among the three types of unrolling. We can see that partial unrolling extracts the benefits of loop unrolling without appreciably affecting the cache behavior. The inclusion of a single table further improves the energy efficiency, however, when the number of tables is increased to four it worsens (due to an increase in memory traffic).

- Energy consumption behavior in absence of any form of loop unrolling is similar to that in presence of partial loop unrolling, *i.e.*, the energy efficiency improves from no tables to one table per round, and deteriorates when the number of tables is increased to four. However, the energy consumption in the presence of one and four tables is greater than for similar configurations with partial loop unrolling.

From the observations above, we can see that partial unrolling gives better energy efficiency than no and full loop unrolling. Furthermore, in the presence of partial loop unrolling a single table look-up was observed to be the most energy-efficient option with respect to the number of table look-ups. The reason for this is substantiated in Figure 3.8 which shows the energy dissipated in the processor and memory in the presence of full loop unrolling (3.8(a)) and partial loop unrolling (3.8(b)), as the number of tables per round is varied from zero to four. Tables substitute computation in arithmetic and logic units of the processor for look-up operations into pre-computed arrays in memory. Thus, in both the cases we see the energy dissipated in the processor decreasing steadily as the number of tables is in-

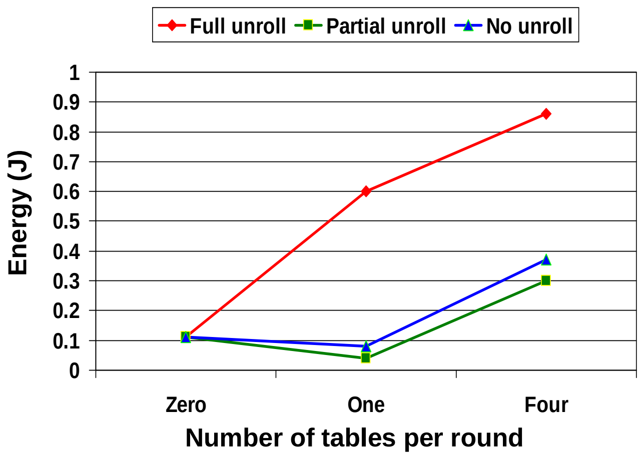


Figure 3.7: Energy consumption of AES as a function of table look-ups

creased. On the other hand, tables increase the number of accesses to memory. For both full and partial loop unrolling, using four tables per round disturbs caching behavior, and the resulting increase in memory energy consumption (due to more memory accesses) is much higher than reduction in processor energy dissipation (the magnitude of this difference in much higher for full loop unrolling). However, the memory energy consumption behavior in the presence of one table is drastically different for full loop unrolling and partial loop rolling. Upon increasing the number of tables from zero to one, memory energy consumption decreases for partial loop unrolling whereas it increases for full loop unrolling. We can see that full loop unrolling negatively affects the caching behavior (resulting in more memory accesses, and accompanying increase in energy dissipation), and introducing tables further worsens the energy consumed. On the other hand, partial loop unrolling does not affect caching behavior, and this behavior remains unchanged with the introduction of a single table. Thus, partial loop unrolling with one table gives the best performance.

3.5.1.5 Energy consumption versus security trade-offs

If different security levels can be provided by a cryptographic algorithm, each with its associated energy consumption characteristic, a security protocol has the option to adapt the level of security commensurate with the current state of the battery of the system with a view of extending its life. This is best exemplified in symmetric algorithms where the

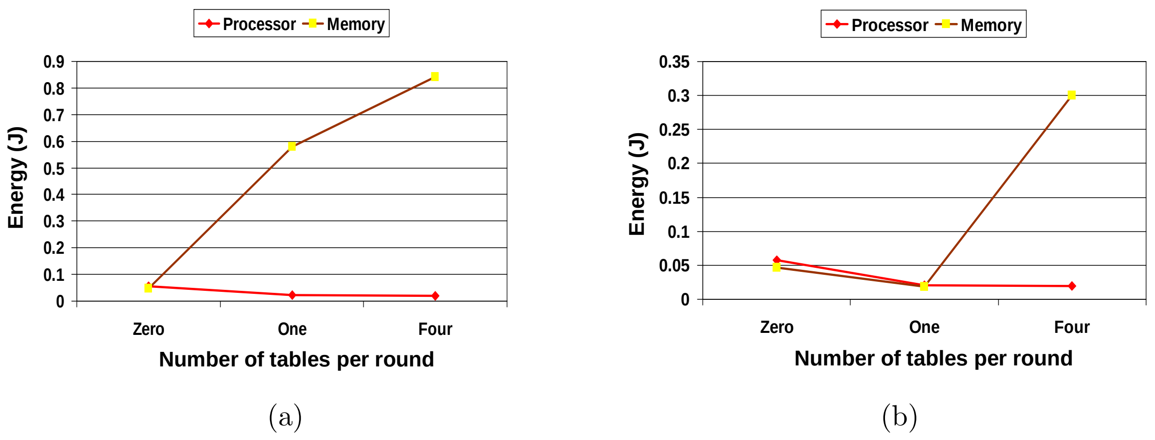


Figure 3.8: Comparison of processor and memory energy consumption in (a) full unrolling and (b) partial unrolling.

security level can be altered by adjusting functional parameters, like cipher modes, key size, and number of rounds. We show that each of these parameters has a considerable effect on the energy cost of the algorithms, thereby resulting in energy-security tradeoffs of practical interest.

The different cipher modes of operation of a block cipher result in algorithmic variants with different energy consumption characteristics and also security levels. We illustrate this fact in the context of AES. The simplest mode is the ECB, but it is susceptible to cryptanalytic attacks. The remaining modes (cipher block chaining (CBC), cipher-feedback mode (CFB), and output-feedback mode (OFB)) employ a feedback mechanism so that the encryption of a plaintext block is made dependent on the results of encryption of previous plaintext blocks. These modes differ in the manner in which the feedback loop is realized. Due to the feedback mechanism, even for the same key, a given plaintext will not always map to the same ciphertext. Thus, CBC, CFB and OFB offer greater resistance to cryptanalytic attacks than ECB [4]. Also, the size of the key has an effect on the security offered by an algorithm: the larger the key size, the greater the security offered [4]. Table 3.4 presents the energy consumption of the AES algorithm for various operating modes and key sizes. From the table, we can observe that

- The energy consumption for the key set-up phase and encryption increases with the key size.

- ECB is the most energy-efficient mode for encryption, and the energy cost of encryption increase across CBC, OFB, and peaks for the CFB mode.

Previous studies have shown similar results about the relationship between key size and energy consumption in block ciphers [7]. The results on the effect of cipher mode and energy consumption are new, and we expect them to be true across various block ciphers.

Table 3.4: Energy costs of AES variants

Key size (bits)	Key setup (μJ)	ECB (μJ/B)	CBC (μJ/B)	CFB (μJ/B)	OFB (μJ/B)
128	7.83	1.21	1.62	1.91	1.62
192	7.87	1.42	2.08	2.30	1.83
256	9.92	1.64	2.29	2.31	2.05

The number of rounds of execution has a proportional effect on the security of the algorithm. Table 3.5 identifies different security levels for the RC5 cipher, obtained by changing the number of rounds used in the cipher, for a given key and block size (128 bits). Each entry indicates the data (number of attempts) needed for a successful attack against RC5 using differential and linear cryptanalysis techniques. Symbol > denotes the case when the attacks are deemed impossible even theoretically.

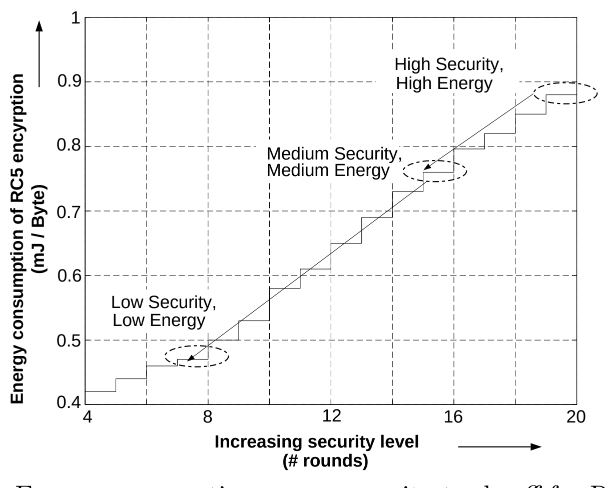


Figure 3.9: Energy consumption versus security trade-off for RC5 encryption

We measured the energy consumption of RC5 for various security levels, and the detailed energy versus security trade-off curve is shown in Figure 3.9. This shows a scheme for lowering the energy consumption by adjusting the security level from high to mid to low, achieved by changing the number of RC5 rounds from 20 to 16 to 8, respectively. We also analyzed the combined effect of key size and number of rounds on the energy cost of key setup for RC5. From Table 3.6, we can see the cost of key setup steadily increasing with key size and number of rounds.

Table 3.5: Multiple levels of cryptanalytic difficulty in RC5 [1]

Rounds	4	6	8	10	12	14	16
DC-C	2^{19}	2^{42}	2^{58}	2^{83}	2^{106}	2^{123}	$>$
DC-K	2^{74}	2^{86}	2^{94}	2^{106}	2^{118}	$>$	$>$
LC	2^{47}	2^{95}	2^{119}	$>$	$>$	$>$	$>$

* *DC-C: Differential cryptanalysis (chosen plain-text), DC-K: Differential cryptanalysis (known plain-text), LC: Linear cryptanalysis (known plain-text)*

Table 3.6: Energy consumption of RC5 key setup

Key size (bits)	Rounds	Energy (μJ)
64	8	36.60
	12	51.00
	16	68.53
128	8	65.80
	12	66.96
	16	71.48
256	8	122.34
	12	126.10
	16	131.12

3.5.2 Energy analysis of the SSL protocol

Figure 3.10 shows the typical (client-side) sequence of operations for a secure session that uses the SSL protocol. The first stage involves loading the client certificate from local storage, optionally decrypting it using a symmetric cipher and performing an integrity check. Note that the above cryptographic operations are optional and are not part of the SSL protocol. This is only a means of ensuring that the user's certificates stored on his local machine were not tampered with by some unauthorized agents. Once the SSL handshake initiates a session, the client and server begin a sequence of exchanges which result in the client-side operations shown in the figure. The operations include (i) *server authentication*, where the client verifies the digital signature of the trusted certificate authority (CA) on the server certificate through decryption using the public key of the CA, followed by an integrity check, (ii) *client authentication*, where the client generates a digital signature by hashing some data using the MD5 and SHA-1 algorithms, concatenating the digests, and encrypting the result with its private key, and (iii) *key exchange*, where the client generates a 48-byte pre-master secret (used to generate the secret key for the record stage) and encrypts it with the public key of the server. Once the connection is established, secure transmission of data proceeds through the SSL record stage.

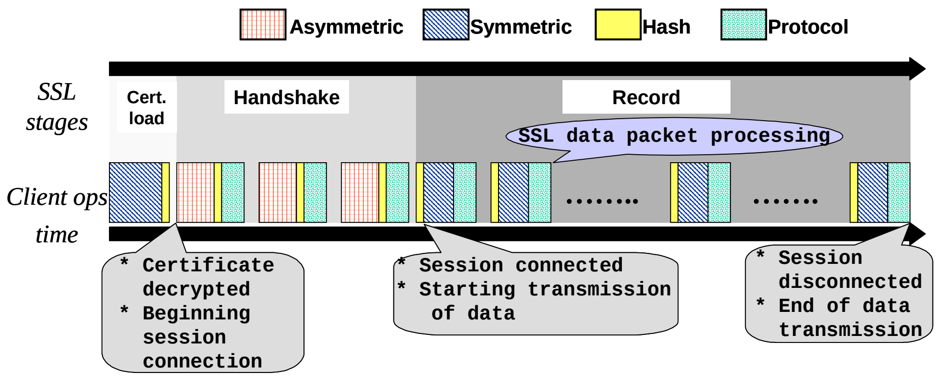


Figure 3.10: Sequence of client-side operations for an SSL session

In the next two sub-sections, we discuss the energy consumption of the SSL protocol with respect to that consumed by computation and communication, respectively. In the concluding sub-section, we discuss the techniques for optimizing the energy consumption of different stages of the SSL protocol, *i.e.*, handshake and record stages.

3.5.2.1 Energy cost of computation in SSL protocol processing

The computation in SSL protocol processing is divided among the operations in the handshake and record stages. Figure 3.11 examines the energy consumption contributions from the handshake and record stages of the SSL protocol for various transaction sizes. We can see that for small transaction sizes (upto 256KB), the SSL handshake protocol dominates the overall energy consumption (*e.g.*, 98.9% for 1KB transactions), while for

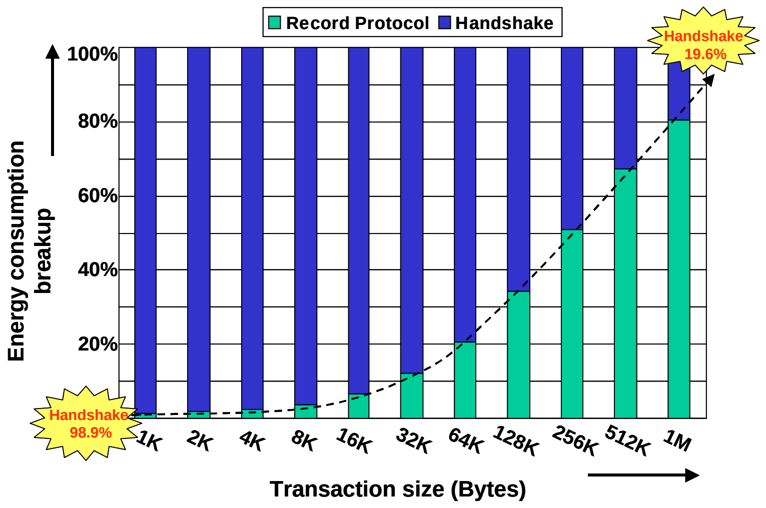


Figure 3.11: Variation of energy consumption contributions from the SSL handshake and record stages with increasing transaction sizes

large transactions, the energy consumption of the SSL record protocol is significant (*e.g.*, 80.4% for 1MB transactions). Therefore, we have to optimize the energy dissipated by the handshake protocol to improve the energy efficiency of small transactions, and similarly target the record protocol for significantly improving the energy consumption behavior of large transactions. Usually, the operating environment of a device determines whether the small transactions dominate or the large ones do.

The SSL handshake and record stages include cryptographic and non-cryptographic operations. It is of interest to find out how the total energy consumed in SSL data transactions is divided between cryptographic and non-cryptographic processing. This information will enable us to calculate the upper bound on the energy savings we can achieve by improving the energy efficiency of cryptographic algorithms. Figure 3.12 summarizes our findings on the energy consumption of the cryptographic and non-cryptographic components of the SSL computation for three different transaction sizes (1KB, 100KB, 1MB). Cryptographic processing includes processor cycles spent in the execution of symmetric, asymmetric and hash algorithms as part of SSL protocol execution. Non-cryptographic processing encompasses all the system functions that are necessary for sending and receiving data over a network (shown as protocol processing in Figure 3.12). Examples of this would be net-

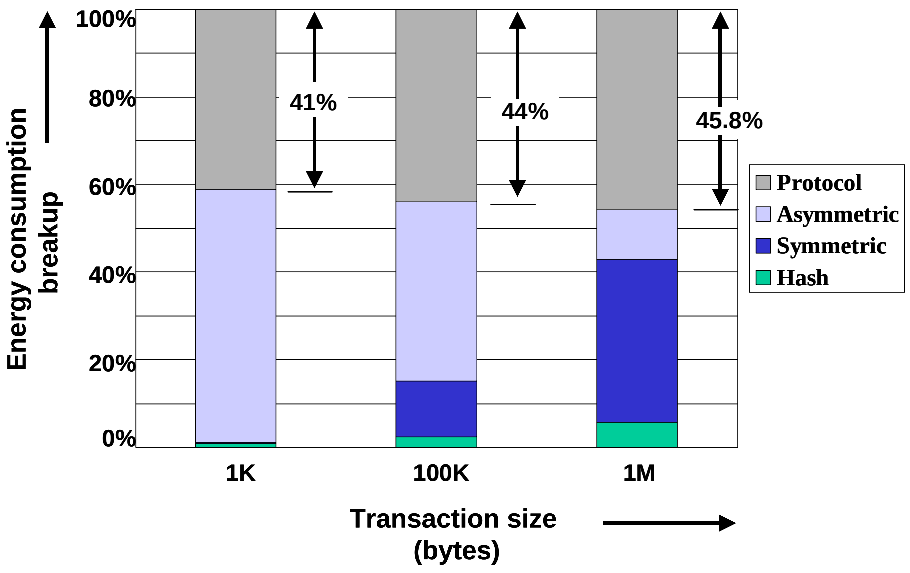


Figure 3.12: Break-up of SSL energy consumption into cryptographic and non-cryptographic components

working functions operating at different layers of the protocol stack (socket, TCP, IP and network interface), memory management for buffering packets after reception and before transmission, *etc.* Based on the data in Figure 3.12, we make the following observations:

- The energy used by cryptographic processing contributes a significant percentage to the total energy dissipated. For example, in a 1KB SSL-enabled data transaction, 58% of the total energy dissipated is due to cryptographic algorithms. We also note that the energy contribution from cryptographic processing steadily decreases with the size of data transactions.

- For small-sized data transactions, the energy dissipated by cryptographic processing is made up almost entirely of contributions from asymmetric algorithms. However, as the size of the transaction increases, symmetric algorithms replace asymmetric algorithms as a dominant contributor to the total energy dissipated by cryptographic processing. The energy contribution of hash algorithms also increases with data size, but remains a minor fraction of the total energy consumption. For example, in a 1KB data transaction, the energy dissipated by asymmetric algorithms forms more than

90% of the energy consumption of cryptographic processing (and roughly 56% of the overall energy consumption). When the data transaction size is increased to 1 MB, the contribution of asymmetric algorithms to the energy consumed by cryptographic processing reduces to 23% (12% of overall consumption) and that of symmetric algorithms increases to 67% (37% of overall consumption).

Thus, smaller transactions benefit greatly from the optimization of asymmetric algorithms for energy, while an improvement in energy efficiency of symmetric algorithms significantly improves the energy cost of large transactions.

3.5.2.2 Communication energy cost in SSL protocol processing

We now analyze the communication energy overhead resulting from the transmission and reception of extra bytes resulting from SSL-related security processing. The extra bytes that are transmitted or received by the client are part of the packets which belong to one of the following four protocol layers of SSL, namely: *handshake*, *record*, *alert*, and *change cipher spec*. The packets belonging to the SSL handshake, SSL change cipher spec, and SSL alert layers are exchanged for the purpose of initiating, maintaining and closing the SSL connection, whereas the SSL record layer is responsible for securely transmitting the data. Therefore, the extra bytes result from two sources: first, from the protocol management packets exchanged by the non-record SSL layers, and, second, from the extra information (like the hash value, SSL header, *etc.*) appended to the data in the packets by the SSL record layer. In Table 3.7, we show the actual number of bytes, categorized according to the SSL protocol layer to which they belong, that are transacted between a client and server for four different files (having sizes 1KB, 10KB, 100KB and 1MB) in an SSL connection. These numbers were obtained by using *ssldump* [73], a software package which can be used to read SSL traffic on networks.

In Table 3.7, the first column gives the size of the transaction. The second column shows the direction of the flow of bytes exchanged, *i.e.*, "$C \rightarrow S$" stands for "client to server", and similarly, "$S \rightarrow C$" means "server to client". Therefore, in the "$C \rightarrow S$" flow, the bytes

Table 3.7: Profile of the data exchanged between the client and server in an SSL connection

Transaction size (*bytes*)	Direction	Handshake (*bytes*)	Record (*bytes*)	Change cipher spec (*bytes*)	Alert (*bytes*)	Total (*bytes*)
1K	C $\rightarrow$ S	183	20	32	1	236
	S $\rightarrow$ C	1,627	1,088	32	1	2,748
10K	C $\rightarrow$ S	183	20	32	1	236
	S $\rightarrow$ C	1,627	10,432	32	1	12,092
100K	C $\rightarrow$ S	183	20	32	1	236
	S $\rightarrow$ C	1627	104,000	32	1	105,660
1M	C $\rightarrow$ S	183	20	32	1	236
	S $\rightarrow$ C	1,627	1,064,960	32	1	1,066,620

are transmitted by the client, whereas, in the "$S \rightarrow C$" flow, it receives them. The next four columns show the amount of data generated by the four layers of the SSL protocol. The last column gives the total number of bytes transmitted and received by the client. The iPAQ, acting as the client, consumes different amounts of energy for transmission and reception, which are 3.36 $uJ/Byte$ and 2.64 $uJ/Byte$, respectively. Table 3.8 shows the energy overhead incurred by the client due to the transaction of the extra bytes resulting from SSL processing. The first column gives the size of the file, and the second column gives the energy expended in receiving this file from the server without using SSL. The fourth column shows the actual number of bytes handled by the client, in order to receive the file, of size as given by the entry in the first column, from the server using SSL. The fifth column gives the energy consumed by the client in a file transaction using SSL. Finally, the last column shows the communication energy overhead of SSL security processing which is the difference between the energy values given in the fifth and second columns.

From Table 3.8, we can see that the ratio of the communication energy overhead to the theoretically needed energy is very high for small-sized data transactions. Thus, the communication energy cost of security imposes a significant burden for small data transactions. This is primarily due to the large number of bytes exchanged in the non-record layers of the SSL protocol. However, as the size of the data transaction increases, the communication overhead of security forms a smaller and smaller fraction of the theoretically required energy, *i.e.*, the impact of the communication energy overhead is reduced. This trend is illustrated in Figure 3.13. In this figure, the two tuples indicate the transaction size and percentage communication energy overhead. We can see that for a transaction of size 1KB, the percentage communication energy overhead is a very significant 192.69 %, and it falls to 1.75 % for a 1M-sized transaction. Also, the SSL client needs to update its SSL certificate database from time to time by including new ones, and the energy is expended in downloading them over the wireless interface. A typical SSL certificate provided by a certificate authority is around 1,200 bytes, and the energy cost of updating a certificate comes to 3.168 mJ.

Table 3.8: Communication energy overhead in the client due to SSL security processing

Transaction size (*bytes*)	Energy needed without SSL overhead (*uJ*)	Action	Data exchanged when using SSL (*bytes*)	Energy spent with SSL overhead (*uJ*)	Energy overhead of SSL usage (*uJ*)
1K	2,703.36	Tx	236	8,047.68	5,344.32
		Rx	2,748		
10K	27,033.6	Tx	236	32,715.84	5,682.24
		Rx	12,092		
100K	270,336.00	Tx	236	279,735.36	9,399.36
		Rx	105,660		
1M	2,768,240.64	Tx	236	2,816,669.76	48,429.12
		Rx	1,066,620		

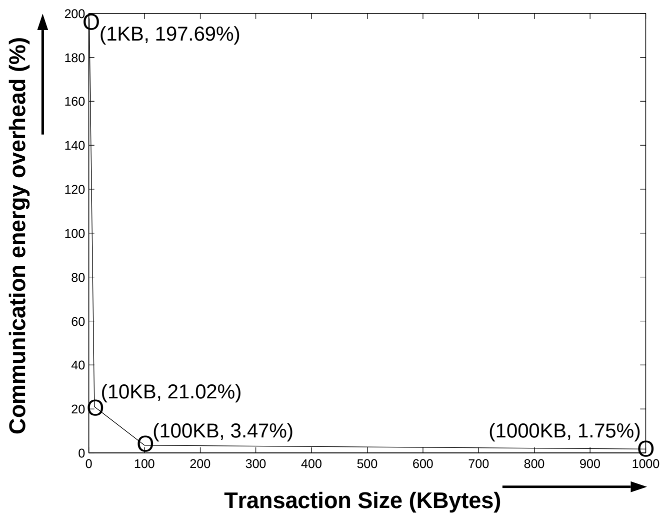


Figure 3.13: Percentage communication energy overhead of SSL usage for varying transaction sizes

3.5.2.3 Scope for optimizing SSL protocol processing

Having examined the energy consumption characteristics of the SSL protocol, both with respect to computation and communication, we now analyze how the energy consumption of the handshake and record stages is affected by various protocol-level services as well as cryptographic algorithm parameters. Specifically, we describe how the use of client authentication impacts the energy consumption due to SSL handshake and how the choice of cipher-suite affects the energy consumption of SSL handshake and record stages, respectively. We conclude this sub-section by enumerating the various ways in which the energy consumption of SSL protocol processing can be optimized.

3.5.2.3.1 Impact of client authentication and asymmetric cipher choice on SSL handshake

We investigated the energy cost of the SSL handshake protocol using the RSA algorithm and the ECC algorithms (ECDSA/ECDH) to implement various public-key operations. The results of our analysis are presented in Figure 3.14. The SSL handshake can be performed between a server and client with or without client authentication. In the case of handshake without client authentication, the following operations are performed by the client and server:

- **RSA-based handshake**: The client performs two RSA public key operations (verify and encrypt), and the server performs an RSA private key operation (decrypt).

- **ECC-based handshake**: The client performs verification using ECDSA, and an ECDH operation is performed to compute the shared secret. The server performs an ECDH operation to calculate the shared secret.

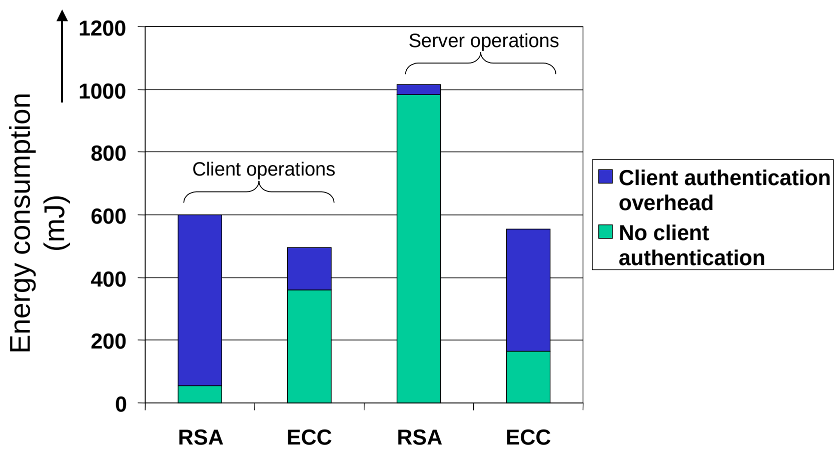


Figure 3.14: Energy consumption for client and server operations in SSL handshake under the presence or absence of client authentication

If client authentication is required, some extra operations need to be performed by the client and server. These are:

- **RSA-based handshake**: The client performs an RSA private key operation (sign). The server performs two extra RSA public key operations (verify).

- **ECC-based handshake**: The client performs an additional signing operation using ECDSA, and the server performs two extra verification operations using ECDSA.

Figure 3.14 shows the energy consumed by the SSL handshake process using RSA or ECC algorithms for the handheld functioning as a client or a server. Though the handheld typically behaves as the client in a majority of transactions, it may sometimes be required to play the part of the server. In order to investigate this scenario, we allowed the handheld to perform the server operations for collecting the corresponding energy data. Energy data were also collected for studying the impact of client authentication in all the cases. With respect to the client energy cost, we can see from the figure that RSA-based handshake is much more efficient than ECC-based handshake when there is no client authentication in the SSL handshake stage. However, in the presence of client authentication in SSL handshake, ECC-based handshake consumes less energy than RSA-based handshake. In general, we believe that various protocol-level parameters have interdependent effects on energy, leading to many interesting trade-offs.

3.5.2.3.2 Impact of cipher-suite choice on SSL record energy consumption

The energy cost of the SSL record stage is mainly determined by the amount of bulk data that is transmitted. An analysis of the cipher suites shows that careful choices of cryptographic algorithms need to be made, in order to optimize energy during the record stage. Consider the following two cipher suites, ECC-BLOWFISH-SHA1 and ECC-AES-SHA. A cursory examination would conclude that the second cipher suite is more energy-efficient, given the very high cost of key setup in BLOWFISH. However, Figure 3.15 shows that if the amount of data transacted is greater than 7.9KB, then, in fact, the first cipher suite is more efficient. This is because the cost of key setup in BLOWFISH is gradually amortized, and the advantages of BLOWFISH come into play.

Figure 3.15 illustrates the energy consumption of two cipher suites, RSA-RC5-SHA1 and ECC-3DES-SHA. The public-key algorithm (RSA or ECC) is used in the SSL handshake stage and the symmetric-key algorithm (RC5 or 3DES) is used for bulk encryption in the SSL record stage. The figure shows that for data sizes smaller than 21KB, ECC-3DES-SHA is more energy-efficient because ECC is simpler than RSA (and asymmetric energy consumption dominates that of small data transactions). However, for transactions where there are significant bulk data (greater than 21KB) to encrypt, RSA-RC5-SHA1 consumes less energy, because for large data transfers, the energy consumption of symmetric ciphers dominates the total energy spent, and RC5 is much simpler than 3DES. This shows that a judicious choice of cryptographic algorithms can greatly reduce the amount of energy consumed.

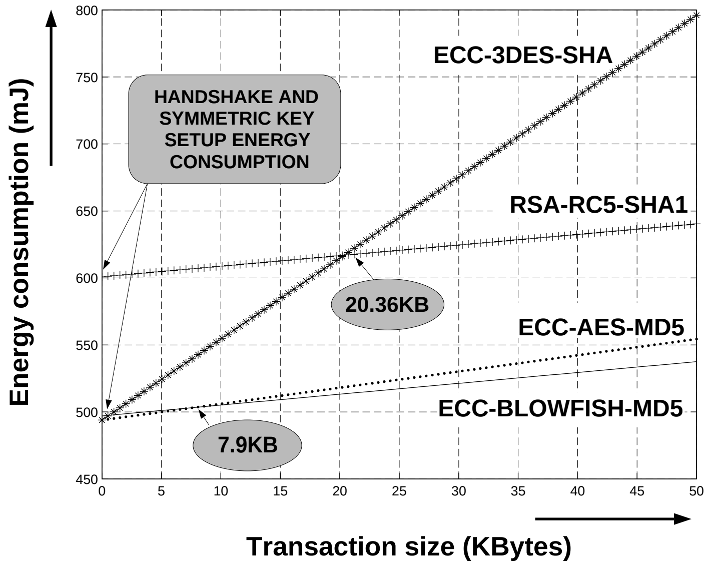


Figure 3.15: The impact of cipher suite selection on energy consumption during the SSL handshake and record stages

3.5.2.3.3 Scope for optimizing SSL

The energy analyses of the SSL protocol and the constituent cryptographic algorithms provide us with insights into the opportunities available for saving energy during protocol execution. These opportunities emerge from (a) the presence of security levels (ranging from high to low) for each cryptographic algorithm and security protocol with each level correlated with distinct energy consumption characteristics, (b) the availability of parameters in a cryptographic algorithm such as key sizes, number of rounds, *etc.*, that can be tuned for energy efficiency and security level, and (c) the availability of parameters in a security protocol such as services provided, cipher/cipher-suite used for a given security service, *etc.* It should be remembered that for the above optimizations to be practical, the capabilities of both the interacting parties (server and client) should be enhanced.

We now examine the security *vs.* energy tradeoffs that result from variation of key size, cryptographic algorithm, cipher parameter implementation choices and protocol steps.

- **Key size**: The size of the keys used in cryptographic primitives is flexible. The key size determines the strength of the cryptographic primitive, and also the amount of computation done. This is *especially* true for asymmetric algorithms. For example, in modular exponentiation (ME)-based asymmetric algorithms, like RSA *etc.*, if k is the number of bits in the key, the number of modular multiplications necessary to compute one ME operation is $1.5k$ (when the LR binary algorithm is used). Since small-sized keys can be easily broken by an adversary, the keys cannot be made arbitrarily smaller. Cryptographic studies [74] have shown that to maintain an acceptable level of security, key sizes should be greater than or equal to 64 bits, 512 bits and 163 bits for *symmetric algorithms*, *RSA and discrete logarithm (DL)-based*, and *elliptic curve (EC)-based asymmetric algorithms*, respectively. Table 3.9 shows the key sizes advisable for the two levels of security for various types of algorithms. Key size has a significant effect on the energy consumed by asymmetric algorithms as observed in the case of DH key exchange, where the energy consumption for key exchange increases from $159.6mJ$ to $1046.5mJ$, in going from a 512-bit key to a 1024-bit key (Table 3.2).

Table 3.9: Key sizes for the two security levels

Type of algorithm	Keysize in bits	
	Low security	High security
Symmetric	64	$\geq$ 128
RSA and DL	512	$\geq$ 1,024
EC	163	$\geq$ 190

- **Choice of algorithm**: Different cryptographic primitives exist to realize the same security functionality. It is known that the different choices give the same level of protection with *equivalent sized keys* [74]. For example, in asymmetric algorithms, a 1024-bit RSA key is calculated to give the same level of protection as a 163-bit ECC key. However, in spite of this observation, the choice of algorithm to be used in a session is still open. This is due to a variety of reasons, such as the algorithms supported by the end parties, the perceived confidence in the algorithms by the users[2] and the nature of the applications. The following observations can be made regarding algorithm selection for achieving energy efficiency:

1. AES offers a good mixture of security and energy efficiency. Its security properties have been well-studied, and it was found to offer high resistance to linear and differential cryptanalysis. In addition, it also has a low energy cost for both key setup and encryption. However, recent studies have pointed out that it might be susceptible to algebraic attacks [75]. In terms of high resistance to cryptanalytic attacks, other algorithms, which fare well, are 3DES and IDEA [3]. When lower energy consumption is a higher priority, RC5 and Blowfish serve as possible candidates (Figure 3.6). Blowfish is the ideal choice when large amounts of data are to be transmitted with a low frequency of key refreshes.

2. When there is a choice in the symmetric algorithm to be used (as in the latter case mentioned above), the size of the data to be encrypted should be an important factor in making a decision. Within the available set of algorithms, the one which can transmit the data with the least energy consumption should be selected. The energy cost of symmetric encryption, Si, can be estimated by a simple equation, given by:

$$Energy\_cost_{Si} = Key\_setup(Si) + Energy\_per\_byte(Si) \times Data\_size$$

   where $Key\_setup(Si)$ is the energy cost of expanding the symmetric key for algorithm Si, energy expended per byte for encryption/decryption using algorithm Si is given by $Energy\_per\_byte(Si)$, and $Data\_size$ is the total size of the data to be encrypted by algorithm Si. Figure 3.6 gives the Key_setup and $Energy\_per\_byte$ values for different symmetric algorithms.

[2]The cryptographic algorithms are as secure as the limitations of the present cryptanalytic techniques. A new discovery, either in mathematics or cryptanalysis, can undermine the security of a cryptographic algorithm which was hitherto considered to be unbreakable. Usually, the longer an algorithm stays unbroken, the greater is the trust placed in it. For example, though ECC is eminently suited for small form factor devices and is gaining wide acceptance, some parties prefer to use RSA just because it has been around longer, and no successful attacks have been carried out against it.

3. For digital signatures, RSA consumes much less energy for verification compared to signature generation, while ECDSA displays a complementary behavior. In a battery-constrained handheld, if the frequency of signature generation is much *smaller* than verification then it is advisable to use RSA, while ECDSA is suggested if the frequency of signature generation is much higher. However, if both the operations are performed with similar frequencies on the handheld then it might be a good idea to use ECDSA because the total energy dissipated for both signature generation and verification is less than in the case of RSA and DSA (Table 3.1).

- **Cipher parameters of an algorithm**: The cipher parameters of an algorithm influence the manner in which the ciphertext is produced. The cipher parameters have a significant influence on symmetric algorithm execution (and, thereby, the energy dissipation), and include the number of rounds, and the mode of operation (ECB, CBC, CFB and OFB). The recommendations, in terms of selecting the cipher parameters, for obtaining energy efficiency by an appropriate selection of cipher parameters are listed next:

  1. For a high security level, the number of rounds in a symmetric algorithm execution should be more than for a low security level. In RC5, where the number of rounds is variable, for low security the number of rounds is set to eight, and for high security the number of rounds can be set at 12 or higher (Figure 3.9).
  2. The choice of mode is dictated by the nature of the application, and its desired security level. The ECB mode consumes the least energy, but is susceptible to statistical attacks using the plaintext. Therefore, it is suited for encrypting short random data with a low security level. For a high security level and sending normal data, it is advisable to use one of the CBC, CFB or OFB modes. They provide more security compared to the ECB mode at the expense of a higher energy consumption. In the CBC mode, the present plaintext block is XORed with

the previously generated ciphertext block to provide better protection against statistical attacks. In CFB and OFB modes, the amount of ciphertext given as feedback to be XORed with the present plaintext block can be varied in sizes having increments of 8 bits. These two modes are suited for networking applications where there is streaming data and, therefore, one does not need to wait for the entire block to arrive to do encryption (as is necessary in CBC). Table 3.4 gives the energy consumption values for these modes in the case of the AES algorithm.

- **Implementation choices**: In Section 3.5.1.4, we showed that memory accesses have a considerably higher energy cost than computations in the datapath of the processor. However, techniques like table lookups and loop unrolling are frequently employed for performance reasons. Usually, aggressive use of these measures results in an increase in the number of memory accesses, thereby decreasing energy efficiency. Thus, for the sake of energy, we should be judicious in the extent to which we employ these techniques that increase memory traffic. Depending on the energy and performance constraints, the right balance between the fraction of computation, which can done on the processor, and that which is implemented though table lookups should be achieved (Figure 3.8).

- **Protocol steps**: The operations in security protocols can be divided into two mutually exclusive subsets: compulsory and optional. The optional steps are usually included for extra security. Depending on the security requirements of a task, the optional steps are either included in the protocol execution or not. In SSL handshake, the *client authentication* step is optional. The decisions with regard to execution of protocol steps, necessary to achieve energy efficiency can be stated as follows:

  1. The following energy saving measures are valid when the handheld is acting as a client (Figure 3.14):

     – If a high level of security is desired, client authentication should be included in the handshake. Using ECC is much more energy-efficient than RSA.

  – When the security requirements are low, it is advisable to skip client authentication in SSL handshake. In this case, using RSA results in significantly higher energy efficiency than ECC.

2. When the handheld is acting as a server, using ECC results in better energy efficiency both in the presence and absence of client authentication.

Implementing the suggestions proposed for optimizing the energy efficiency of the SSL protocol do not entail making major changes to the existing structure and implementation of the protocol. They determine the parameters for energy-efficient protocol execution based on some characteristics of the secure session, like size of the data transaction, amount of security desired, whether client authentication is required, amount of charge left in the battery, *etc.* The information governing energy-efficient parameter selection can be easily stored as a set of parameter values, and a set of rules which specify the conditions under which the parameters are applicable. This decision-making system can be naturally incorporated into the SSL Handshake protocol which determines the parameter values applicable to the session to be established, and fixes them for the whole length of the session. This approach has an obvious limitation that both the client and server wishing to establish a secure connection should be able to support the protocol parameters aimed at acheiving energy efficiency. Except for this issue, we cannot foresee any practical matters related to existing protocol implementations which will limit the usage of the proposed recommendations.

3.6 Chapter summary

In this work, we presented an analysis of the energy consumption of cryptographic algorithms and security protocols for an embedded system. We examined several cryptographic algorithms from the three main classes — asymmetric, symmetric and hash, and observed that (i) asymmetric and hash algorithms have the highest and least energy costs, respectively, (ii) the energy cost of asymmetric algorithms is dependent on the key size, while that of symmetric algorithms is not significantly affected by the key size, (iii) the energy consumption of a symmetric algorithm depends not only on the bulk data encryption/decryption cost but also on the key set-up cost, (iv) wide variations in energy consumption exist within the same family of cryptographic algorithms, and (v) the level of security provided by a cryptographic algorithm can be traded off for energy savings by tuning parameters such as key size, number of rounds, *etc.*

We also studied the energy consumption profile of the SSL protocol and saw that the energy costs of the handshake and record stages of the SSL protocol vary depending on parameters like the functionality desired in the handshake, size of bulk data transacted, *etc.* Furthermore, we used our analysis to detail the opportunities available for making SSL (and other security protocols) energy-efficient.

Chapter 4

Optimizing IPSec Protocol Execution by Extending and Configuring an Embedded Processor

Security protocols, such as IPSec and SSL, are being increasingly deployed in the context of networked embedded systems. The resource-constrained nature of embedded systems and, in particular, the modest capabilities of embedded processors make it challenging to achieve satisfactory performance while executing security protocols.

A promising approach for improving performance in embedded systems is to use application-specific instruction set processors (ASIPs) that are designed based on configurable and extensible processors. In this work, we perform a comprehensive performance analysis of the IPSec protocol on a state-of-the-art configurable and extensible embedded processor (Xtensa from Tensilica, Inc.). We present performance profiles of a lightweight embedded IPSec implementation running on the Xtensa processor, and examine in detail the various factors that contribute to the processing latencies, including cryptographic and protocol processing. In order to improve the efficiency of IPSec processing on embedded devices, we then study the impact of customizing an embedded processor by synergistically (a) configuring architectural parameters, such as instruction and data cache sizes, processor-memory interface width, write buffers, etc., and (b) extending the base instruction set of the processor using custom instructions for both cryptographic and protocol processing. Our experimental results demonstrate that 3.2X speedup in IPSec processing is possible over a popular embedded IPSec software implementation. This chapter is based on results presented in [40, 41].

4.1 Introduction

Embedded systems are designed under a wide range of constraints, including cost, performance, and power consumption. Security has traditionally been an important consideration in the design of specific embedded systems, such as smart cards. As embedded systems are used in increasingly diverse applications to perform critical functions and access sensitive information, security has become a widespread concern in their design. Due to the networked nature of many modern embedded systems, they are exposed to the myriad security threats that we have experienced with personal computers (PCs) and the Internet.

Conventional security measures, such as cryptographic algorithms, secure communication protocols, and secure computation and storage mechanisms, can also be applied to embedded systems. However, various design constraints and usage scenarios that are unique to embedded systems usher in new challenges. For example, due to cost and power constraints, many embedded systems do not possess the computing power of the general-purpose microprocessors used in PCs or workstations. This leads to a “security processing gap,” or a disparity between the computational requirements for security and the capabilities of embedded processors. For example, performing 3DES symmetric encryption plus SHA-1 hashing at a data rate of 2Mbps requires a computational capability of around 130 Million Instructions per Second (MIPS) [6], which is beyond the capabilities of many low-end embedded processors.

A promising architectural approach to obtain satisfactory performance with low cost and low power is to use processors that are customized to the application or application domain targeted by the embedded system. ASIPs combine the performance and power advantage of application-specific integrated circuits (ASICs) with the versatile programmability of general-purpose processors, without having to deal with the time-to-market issues inherent in custom ASIC design. While ASIPs offer a good tradeoff between flexibility and efficiency, the primary challenge to their adoption has been the complexity of designing a processor and its supporting compiler and software tool chain from scratch. *Configurable* and *extensible* processors have recently emerged as an effective way to design ASIPs by leveraging a pre-designed customizable base processor. Several established and emerging companies have developed embedded processors with configurable and extensible features [76–81]. Configurability refers to the ability to choose high-level architectural parameters, such as functional units (multiplier, accumulator, *etc.*), register file size, cache architecture, memory configuration, *etc.*, to best suit the needs of the target application. The provision for extending the instruction set of the configured processor by adding some special-purpose instructions is termed extensibility. A synergistic use of configurability and extensibility can give rise to ASIPs that are very efficient in satisfying the design constraints imposed by the end application. The following is the summary of extensible and configurable features in the embedded processors listed above:

- Xtensa processor [76]: Under configurable options, it offers optional functional units (multipliers, 16-bit MAC, FPU and DSP units), memory protection features (region protection, memory management unit), ability to choose processor options (write buffer size, endian byte ordering, exceptions, register file size, and cache parameters (size, associativity, and line width), and width of processor to memory interface. Also, it allows extensibility in terms of support for introducing custom instructions.

- ARC processor [77]: The configurable options include DSP extensions (16-MAC, 32-bit MAC and saturating arithmetic instructions), floating point extensions, cache parameters (size, associativity and line width), interrupts, and timers. It allows designers to define custom instructions.

- MIPS32 M4K processor [80]: The configurable options include optional multiply/divide units, coprocessor interface to plug-in a custom defined coprocessor, and JTAG controller. It allows users to define custom instructions.
- PICO express [78]: It allows design of configurable cores where algorithm description in C language is mapped to an array of highly optimized pipelined processing units.
- LISAtek solution [81]: LISAtek provides a framework for automated design of embedded processors optimized for a target application. It provides a language for describing processor cores at a high level of abstraction in terms of registers, memory, and its instruction set. Generators exist to map the processor model to a synthesizable core, and generate the supporting software tool suite.

In this work, we analyze the performance of the popular network security protocol, IPSec, executing on a commercial configurable and extensible embedded processor (Xtensa from Tensilica, Inc.). We investigate the performance tradeoffs resulting from architectural configurability and extensibility, and demonstrate how they can be explored to achieve high performance with minimal hardware complexity.

We chose the IPSec protocol since it is relevant to networked embedded systems, which represent a major and rapidly growing part of the embedded systems market. IPSec is a network-layer security protocol, which provides for confidentiality and integrity of the communicated data. The key benefits of IPSec include transparency to applications (*i.e.*, applications can make use of the security services offered by IPsec without any changes), and flexibility (it can be used in different modes: end-to-end for securing traffic between two hosts, route-to-route for protecting a certain set of network links, or edge-to-edge for forming a secure tunnel between two trusted networks through an untrusted network).

4.1.1 Contribution of this work

Previous work aimed at improving security performance tended to target the acceleration of the cryptographic algorithm computations, ignoring the overhead of security protocol processing [23, 25, 26]. However, this approach does not suffice for security protocols such as IPSec, which also contain a significant amount of non-cryptographic computation in the form of packet processing. There is a difference in non-cryptographic (protocol) processing in IPSec and SSL protocols (shown in Fig. 4.2 and Fig 3.12, respectively). Unlike, IPsec processing which is applied to all packets leaving a host machine, SSL processing is applied on a per application basis. Also, non-cryptographic processing in SSL is much simpler than in IPSec and consists largely of building and reading SSL-specific packet headers (protocol processing in Figure 3.12 includes cycles consumed in non-cryptographic processing in SSL and in networking protocol whereas protocol processing in Figure 4.2 comprises only of non-cryptographic processing in IPSec). Thus, non-cryptographic processing in IPSec is much more complex and compute-intensive than in SSL. Packet processing operations, like building and removing packet headers, looking up policy databases *etc.*, have been shown to be memory-intensive [82], and they pose a considerable bottleneck in IPSec operation as the required data rates increase. This trend will become worse as the gap between the processor and memory speeds widens in the future [83].

In this chapter, we investigate the design space of a configurable and extensible embedded processor platform for improving the performance of IPSec-like security protocols. Our analysis identifies points in the design space that simultaneously improve the performance of both the compute-intensive cryptographic components, and the memory behavior of the protocol processing part, in a cost-effective manner on an embedded processor. *To the best of our knowledge, this is the first work to present a systematic analysis of IPSec execution on configurable and extensible processors.*

The rest of this chapter is organized as follows. Section 4.2 provides a basic overview of the IPSec protocol. Section 4.3 provides various performance statistics for IPSec processing, and motivates the opportunities for using configurability and extensibility to improve performance. Section 4.4 describes our performance analysis and architectural exploration framework, and studies various optimizations for accelerating IPSec on embedded processors. Section 4.5 concludes this chapter.

4.2 IPSec protocol

IPSec is implemented below the Internet protocol (IP) layer of the protocol stack, and transparently provides a variety of security services, *e.g.*, (1) connectionless integrity, (2) data origin authentication, (3) rejection of replayed packets, (4) confidentiality, and (5) limited traffic flow confidentiality [4]. Authentication header (AH) and encapsulated security payload (ESP) can be used in conjunction to obtain all the security services.

An important mechanism for enabling AH and ESP protocols is a security association (SA). The parameters maintained by an SA include protocol(s) used (AH or ESP or both), cryptographic algorithms to implement AH (MD5 or SHA) or ESP (3DES or AES) or both, keys used by the cryptographic algorithms, replay protection window, lifetime of the association, *etc.* For secure communication between two parties, two separate SAs have to be established: one in each direction. All the SAs are stored in a security association database (SAD). The entries in the SAD are indexed using either the security parameter index (SPI) and IP destination address (address of the receiving end of the security association) obtained from the fields of an incoming packet, or the SA index returned by the security policy database (SPD) for an outgoing packet. Normally, security policy decisions, *i.e.*, whether to use security services or not, are made at the flow granularity rather than on a per-packet basis. A flow can be uniquely identified by a four-tuple: source IP address, destination IP address, source port number, and destination port number (collectively called *selectors*). Multiple flows can exist between two networked machines, with possibly different security policies. The security policies of the different flows are stored in the SPD. A flow security policy could specify one of the following: APPLY, BYPASS, and DISCARD.

The processing of IPSec packets is illustrated in Figure 4.1. The sequence of steps applied to an outgoing packet by the IPSec device driver are as follows:

- A packet output from the IP layer is captured by the IPSec device driver.
- The IPSec device driver extracts the selectors from the packet, and uses them as indices to perform a table lookup in the SPD. The corresponding SPD entry specifies

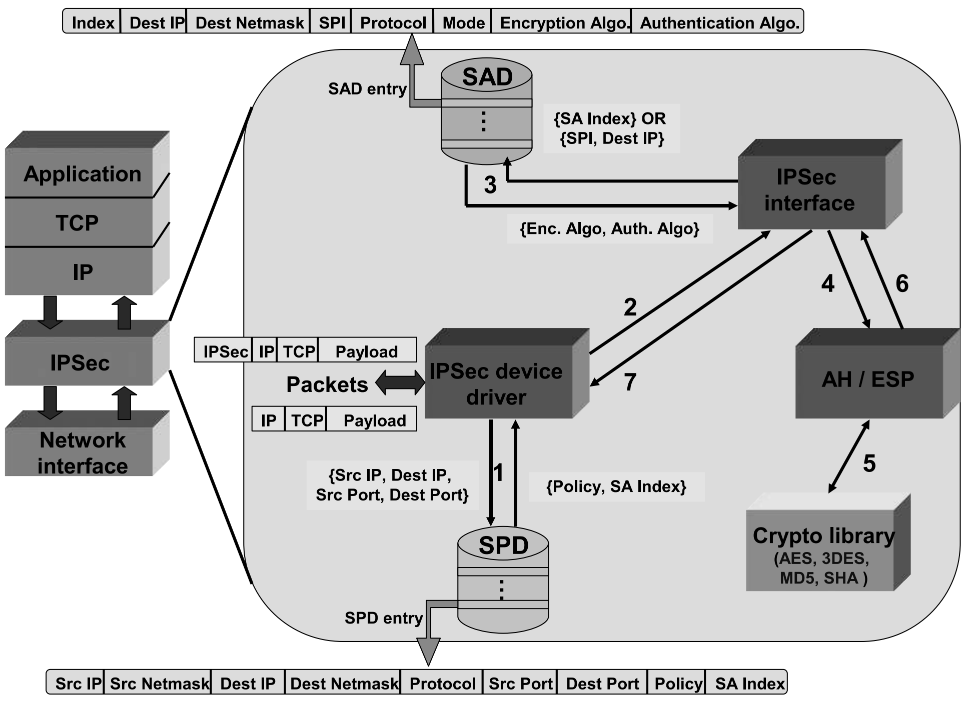


Figure 4.1: Steps in IPsec processing

the security policy for the flow, and also an index into the SAD, if applicable (Arrow 1 in the figure). The fields in an SPD entry are shown at the bottom.

- If the policy says BYPASS, then the packet is passed unmodified to the network interface. In case of DISCARD, the packet is discarded. In both cases, the next packet is captured for processing. However, if the policy mentions APPLY, then the packet is sent to the IPSec interface for further processing. Goto the next step (Arrow 2).

- The IPSec interface performs a table lookup in the SAD, in order to extract information, *e.g.*, which protocol to apply (ESP, AH or both), and the corresponding cryptographic parameters. The SA index returned by the SPD is used to find the correct entry in the SAD (Arrow 3).

- Depending on the SA, the packet is sent for processing using AH or ESP or both (Arrow 4).

- The required ESP and/or AH headers are appended to the packet, and a subset of the packet contents are hashed and/or encrypted. The checksum of the packet data is computed. Hashing/encryption operations are performed using functions provided by a cryptographic library (Arrow 5).

- The packet is passed back to the IPSec interface where a new IP header is constructed (for tunnel mode) (Arrow 6).

- The IPSec interface passes the processed packet to the IPSec device driver which, in turn, sends it to the network device driver for transmission (Arrow 7).

The sequence of steps is slightly different for incoming packets (input by the network device driver) since they already have the SPI embedded in the AH or ESP header. This value is used to directly index the SAD, and inverses of the operations performed on the outgoing packets are performed.

4.3 Motivation

In this section, we analyze the performance of IPSec, and motivate how the configurability and extensibility of an embedded processor can be used to improve it.

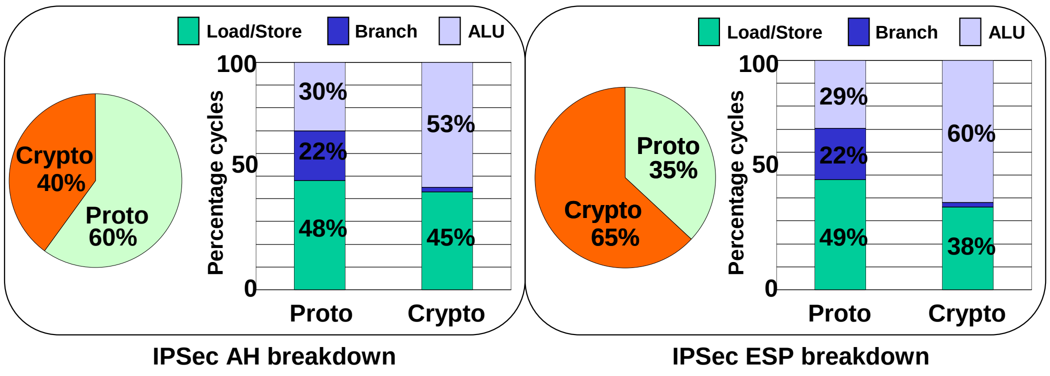


Figure 4.2: Breakdown of IPSec processing into cryptographic and protocol components, and into various instruction categories

4.3.1 Cryptographic and protocol components of IPSec processing

Figure 4.2 breaks down the performance of IPSec processing on the client side of a secure client-server transaction. We separately consider the Authentication Header (AH) and Encapsulating Security Payload (ESP) modes of IPSec. The client is a web browser linked with embedded IPSec [42] and a lightweight network protocol stack (lwIP) [84], and is profiled on Tensilica's Xtensa T1050.1 embedded processor [76]. In these experiments, IPSec used the AES cipher for data encryption/decryption, and the MD5 algorithm for message authentication. Prior to invoking IPSec, pre-negotiated session keys were made available on both the client and server. Further details of our experimental framework are provided in Section 4.4.1. All the experiments were performed using Xtensa, a state-of-the-art extensible and configurable embedded processor, executing a IPSec-enabled network protocol stack, and taking measurements.

First, we measured the number of cycles spent by the client in IPSec-related protocol and cryptographic processing. The piecharts show that nearly 48% (averaged over multiple traces, and across both modes) of the IPSec processing cycles are spent in protocol processing. We further analyzed the cryptographic and protocol components of IPSec processing, and broke them down into cycles spent in executing different types of instructions. The results are shown in the bar graphs of Figure 4.2. We can see that protocol and cryptographic processing have complementary behavior: protocol processing spends majority of its cycles in loads/stores and branch instructions, while cryptographic processing cycles are primarily spent in ALU instructions. This shows that cryptographic operations primarily comprise computational tasks, while protocol functions spend most of their cycles in memory transfers and issues arising out of transferring control in program execution. Thus, we observe that the cryptographic tasks may be more suited for optimization using custom instruction extensibility, whereas the performance of protocol functions may be more suitable improved by judiciously selecting values of the configurable parameters.

4.3.2 Functional analysis of protocol and cryptographic processing

From the above experiments, we also gathered performance statistics of individual functions involved in protocol and cryptographic processing. The results are summarized in Figure 4.3.

The profile for protocol processing consists of four main components: data capture using the IPSec device driver, building and stripping headers (collectively called *header processing*), loading values into security databases and looking them up later, and checksumming. The first bar in Figure 4.3 shows the percentage breakup of protocol processing cycles among these functions. Except for checksum, which involves arithmetic computations, the other functions are dominated by memory interactions. Hence, in our work, the checksum operation is targeted by adding custom instructions, while the other operations are optimized using configurability. The percentage cycle contribution of the constituent functions of AES (used in the ESP protocol) and MD5 (used in the AH protocol) are shown by the second and third bars in Figure 4.3, respectively. All these functions involve intensive computations and, hence, can be sped up by custom instructions.

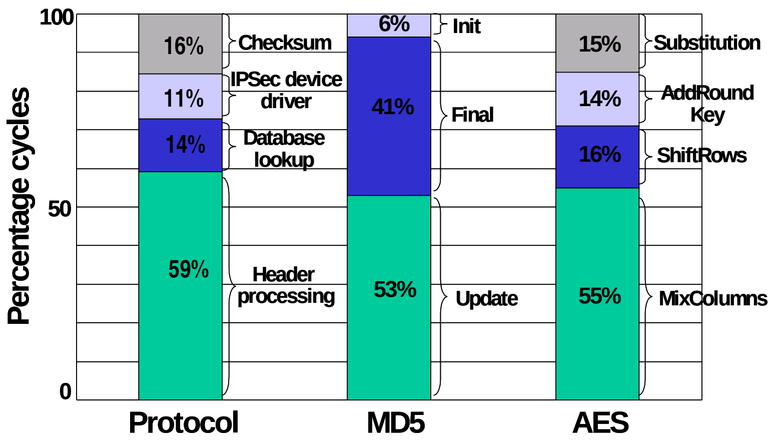


Figure 4.3: Contributions of core functions in cryptographic and protocol components of IPSec processing

4.4 Architectural exploration and performance analysis

In this section, we first describe the architectural exploration and performance analysis framework used in this work (Sections 4.4.1 and 4.4.2). We then evaluate the impact of tuning various extensible and configurable parameters of an embedded processor on the performance of IPSec (Sections 4.4.3 and 4.4.4). Finally, we discuss the implications of our experiments (Section 4.4.5).

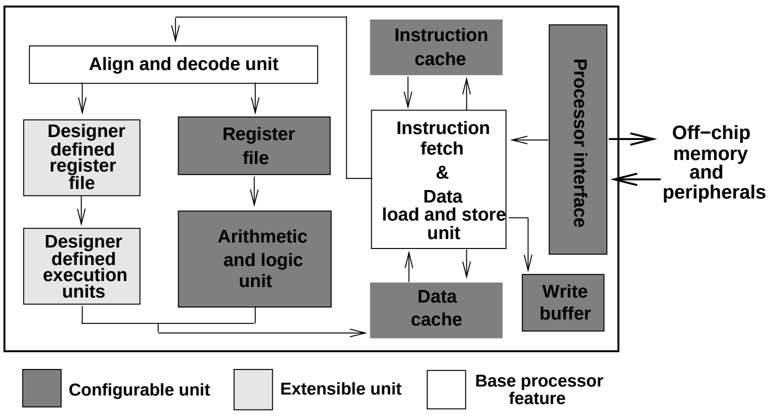
4.4.1 Architectural design space

Tensilica's Xtensa [76] is a high-performance embedded processor with a 32-bit reduced instruction set computer (RISC) architecture. It has a five-stage pipeline, and 32 general-purpose registers. It uses 16-bit and 24-bit instruction words for a denser encoding for reducing the processor code size. It employs register windows for faster handling of procedure calls. Figure 4.4(a) shows the architectural features of the Xtensa processor partitioned into three categories. They include base instruction-set architecture (ISA) features that cannot be altered, configurable options, and extensible options. Configurable options allow for settings to the micro-architectural parameters, such as cache size, associativity, and line size, bi-directional processor interface (PIF) width, *etc.* Extensible options allow the designer to extend the base ISA using special instructions that invoke custom application-specific functional units and registers integrated into the processor's datapath.

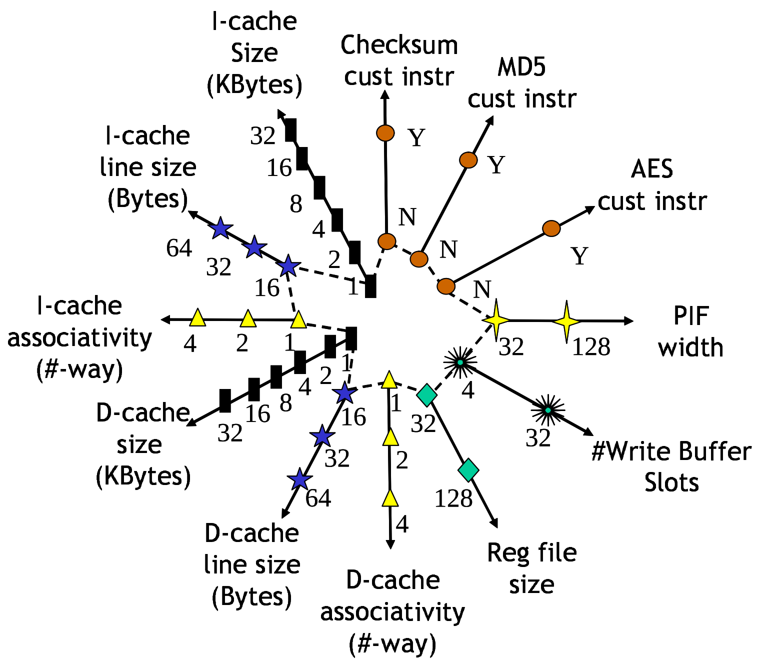
Figure 4.4(b) summarizes the design space of configurable and extensible options explored in this work. Each arrow represents a distinct option, *i.e.*, a dimension in the design space. The specific configurable options include (i) instruction and data cache parameters such as cache size (from 1 to 32 KB), line size (from 16 to 64 B), and associativity (1- to 4-way), (ii) register file size (32 or 128), (iii) number of write buffer slots (4 or 32), and (iv) PIF width (32 or 128). Presence or absence of custom instructions for AES, MD5 and checksum form the list of extensible options considered. An example Xtensa configuration is shown in dotted lines. The goal of our work is to find the configuration in this design space that would provide the maximum performance improvement to IPSec while satisfying the constraints of typical of embedded systems.

4.4.2 IPSec performance analysis and optimization methodology

Figure 4.5 describes the methodology used in this work. The objective of this methodology is to explore the configurable and extensible design space for the Xtensa processor described in the previous section, and improve the performance of IPSec running on the base Xtensa platform. The methodology consists of three phases: (i) gathering real-world IPSec traces[1]


Align and decode unit
Designer defined register file
Register file
Designer defined execution units
Arithmetic and logic unit
Instruction cache
Instruction fetch & Data load and store unit
Data cache
Processor interface
Write buffer
Off−chip memory and peripherals
Configurable unit
Extensible unit
Base processor feature

(a)



(b)

Figure 4.4: (a) Xtensa's architectural features, and (b) design space of configurable and extensible options explored in this work

in a data collection phase, (ii) using the traces to analyze the performance of IPSec on the Xtensa processor through instruction-set simulation, and (iii) identifying performance hotspots and tuning the processor by setting various configuration parameters and/or selecting custom instructions. The result of step (iii) is a new processor, which is then fed back to step (ii) to evaluate the performance impact of the modifications made.

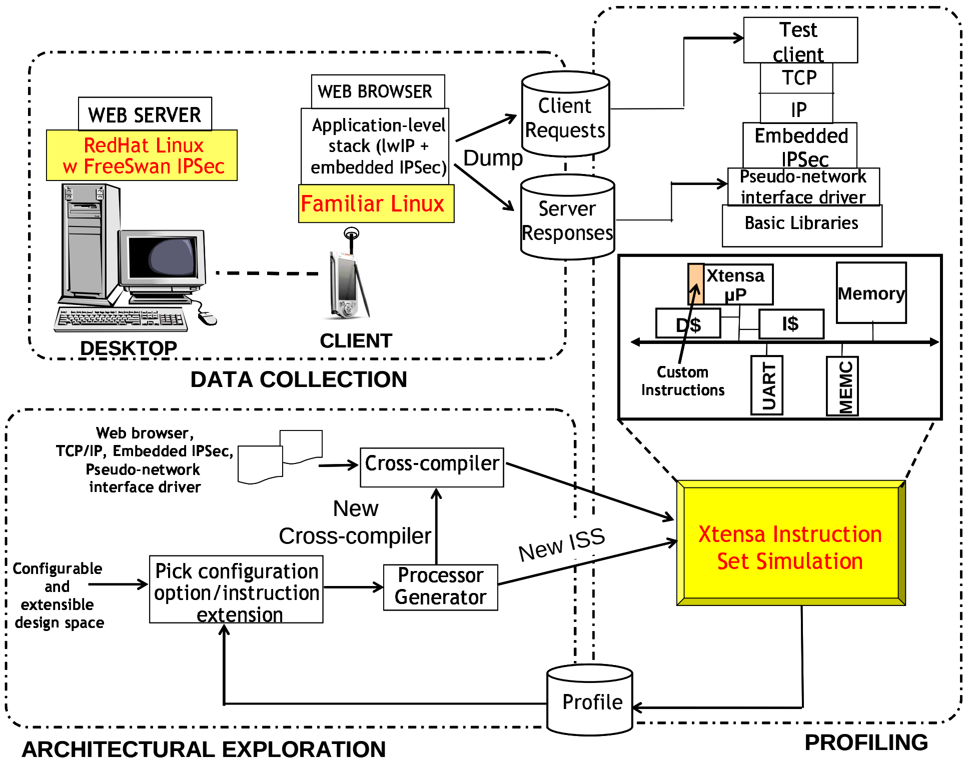

WEB SERVER
RedHat Linux w FreeSwan IPSec
WEB BROWSER
Application-level stack (lwIP + embedded IPSec)
Familiar Linux
Client Requests
Dump
Server Responses
DESKTOP
CLIENT
DATA COLLECTION
Test client
TCP
IP
Embedded IPSec
Pseudo-network interface driver
Basic Libraries
Xtensa µP
Memory
D$
I$
Custom Instructions
UART
MEMC
Web browser, TCP/IP, Embedded IPSec, Pseudo-network interface driver
Cross-compiler
New Cross-compiler
New ISS
Xtensa Instruction Set Simulation
Configurable and extensible design space
Pick configuration option/instruction extension
Processor Generator
Profile
ARCHITECTURAL EXPLORATION
PROFILING

Figure 4.5: The three phases of our design methodology: data collection, performance profiling and architectural exploration

We now examine each phase in further detail:

- **Data collection**: We use an IPSec connection between a web server and a client connected over a local area network as the testbed for data collection. The web server consists of a 700MHz PC with 256MB RAM running Redhat Linux. It runs the FreeSwan IPSec software. The client used in the experiment is a Compaq iPAQ H3670 PDA, which contains an Intel SA-1110 StrongARM processor clocked at 206MHz.

[1]Note that the trace files are a necessity since most instruction-set simulators (ISSs), including the Xtensa ISS, do not model the network interface, and hence, cannot support online simulation of client-server transactions.

The client uses the Familiar distribution [68] of Linux as its operating system (OS), and runs the lightweight TCP/IP stack (lwIP) [84]. lwIP is a thin, application-level protocol stack written for small form-factor devices. An IPSec software targeted toward embedded devices, called embedded IPSec [42] is incorporated into lwIP to provide IPSec support on the client.

The web browser program on the client initiates a connection to the server and sends it requests for web pages. Once a connection is established, the server supplies requested HTML pages to the client. The client requests and server responses are captured in trace files. Several trace files are captured for multiple IPSec-enabled web transactions having different traffic characteristics. The trace files comprise of webpages of various popular websites.

- **Profiling**: The client-side software used in data collection is cross-compiled to the Xtensa platform, and executed on Xtensa's ISS. Since the ISS has no networking support, we introduce a pseudo-network interface driver which sends server-side responses from a trace file to the client executing on the Xtensa ISS. Figure 4.6 illustrates the processes of trace file generation, and the pseudo device driver operation. On the top in Figure 4.6, we have the client and server exchanging message packets $(S_1, C_1, \ldots, S_n, C_n)$ during a transaction secured by IPSec. The messages sent by the server $(S_1, S_2, \ldots, S_n)$ are captured, and stored in a trace file. The bottom of Figure 4.6 shows the pseudo device driver operation. The client program along with IPSec-enabled application-level network protocol stack are executed on the ISS. The pseudo device driver reads a packet from the server trace file, and passes it to a client program through the IPSec-enabled network protocol stack. The client progam processes the server packet (S_i), and sends out the appropriate client response packet (C_i). The driver captures the client response packet, reads the expected server response (S_{i+1}) from the trace file, and sends it to the client program. This process continues until the end of the session. At the end of the session, the ISS outputs the cycles consumed in both the cryptographic and non-cryptographic processing components of the IPSec protocol. In general, care should be taken to ensure that server responses sent by the pseudo device driver are exactly the ones expected by the client program. This can be achieved by deterministically assigning values to some parameters in the client program which are randomly assigned values in each session (for example, certain values in the headers change from one transaction to the other, even though the same data are being exchanged). In this way, we can emulate the client-side operations on the ISS, and obtain IPSec's performance profile on an embedded client. The statistics gathered in this process include various measurements, *e.g.*, the number of processor cycles consumed by all the IPSec functions, instruction and data cache miss rates, memory penalty, *etc.* Table 4.1 gives the list of different microarchitectural features whose effects were examined on the processor cycle count.

- **Architectural exploration**: The performance profile derived in the previous step allows us to pick the configurable and extensible options relevant to this work. Each unique combination of options corresponds to an application-specific processor instance. For each processor instance, we used Xtensa's processor generator to generate a new RTL description of the processor and its corresponding software tool suite (cross-compiler and ISS). The profiling phase is then repeated to evaluate the impact of the selected options.

Table 4.1: Parameters examined in profiling

Microarchitectural parameters studied
Cache size (I-cache and D-cache)
Cache associativity (I-cache and D-cache)
Cache line width (I-cache and D-cache)
Register file size
Write buffer depth
Processor interface width

4.4.3 Tuning IPSec using extensible options

In order to improve the performance of IPSec processing, compute-intensive operations in IPSec protocol and cryptographic processing are converted into special-purpose instruc-

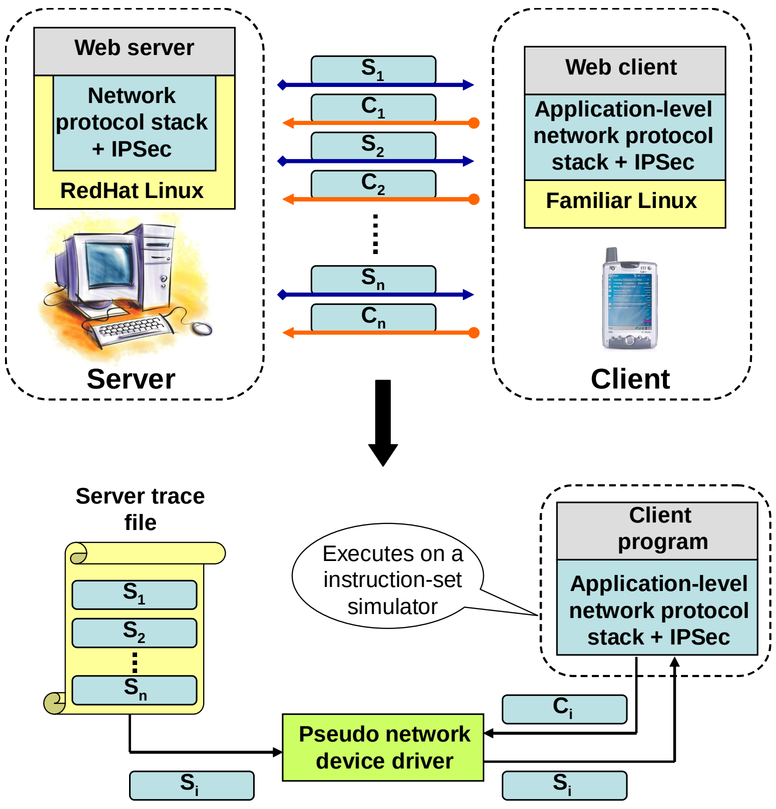

Web server
Network protocol stack + IPSec
RedHat Linux
Server
S_1
C_1
S_2
C_2
S_n
C_n
Web client
Application-level network protocol stack + IPSec
Familiar Linux
Client
Server trace file
S_1
S_2
S_n
Executes on a instruction-set simulator
Client program
Application-level network protocol stack + IPSec
C_i
Pseudo network device driver
S_i
S_i

Figure 4.6: Profiling using the pseudo device driver

tions. These compute-intensive operations (also called *hotspots*) are identified by studying the performance profile of IPSec protocol and cryptographic processing. Based on the performance analysis of IPSec presented in Section 4.3, we infer the following about the hotspots in protocol and cryptographic processing components of the IPSec protocol:

- **Protocol processing**: From Figure 4.3, we see that the *header processing* function consumes the most cycles. However, except for the *checksum* function, the protocol processing functions are dominated by memory transactions, and cannot be sped up with custom instructions. Thus, checksum is the only function in protocol processing which is suitable for custom instruction implementation.

- **Cryptographic processing**: Figure 4.3 shows the functions which consume the majority of the cycles in MD5 (in the AH mode), and in AES (in the ESP mode) and, therefore, can be targeted for custom instruction implementation to make cryptographic processing faster.

Custom instructions are implemented as custom-defined pipelined execution units which are introduced into the processor datapath as shown in Figure 4.7. Also, custom instructions read their inputs from and write the outputs into custom registers. Therefore, custom load and store instructions are defined in order to transfer input operands from memory into the custom registers, and subsequently, write out the results from the registers back to memory. Usually, transfer of values into and from the custom registers consumes appreciable number of processor cycles, and it should be taken into consideration while evaluating the speedup obtained by using a custom instruction. The following subsections present details of the custom instructions added to accelerate protocol and cryptographic processing in IPSec. We do not show the corresponding custom load and store instructions, but their impact is included in the performance results of the custom instructions.

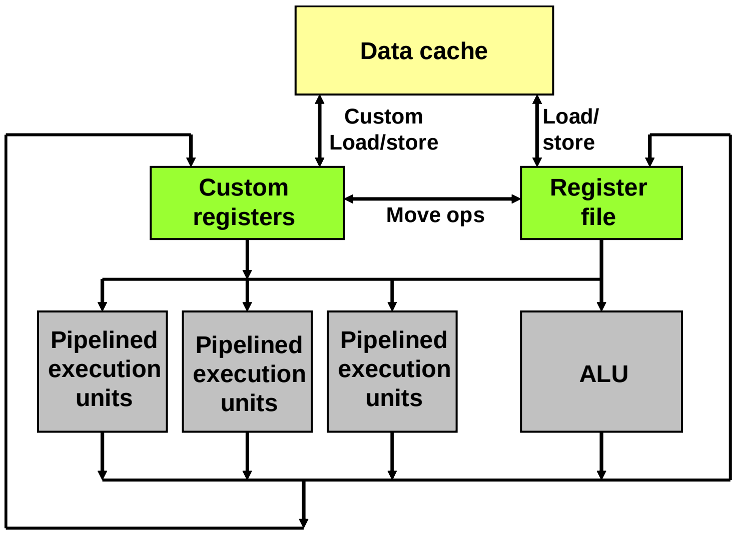


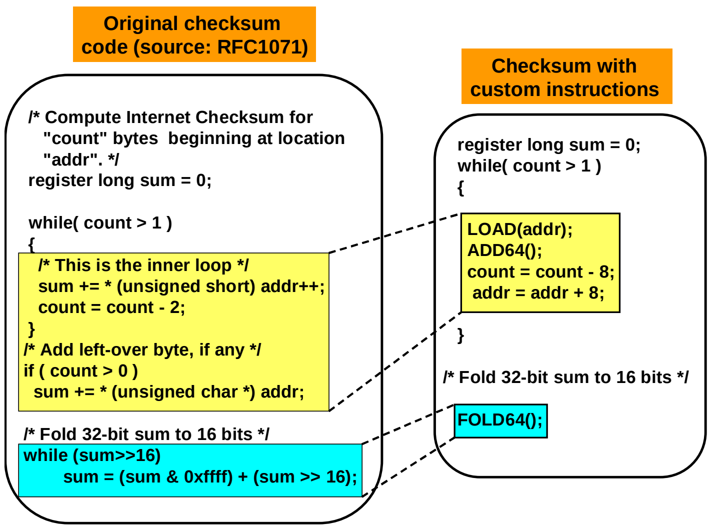
Figure 4.7: Datapath of an extensible processor

4.4.3.1 Checksum

The checksum operation is performed on data in every packet received. The RFC1071 [85] software implementation of the checksum operation accumulates 16-bit numbers into a 32-bit variable `sum`. The 32-bit sum in the accumulator is folded into a 16-bit result and complemented to get the final checksum value. In folding operation, any higher order bits generated due to carry are added back to the 16-bit number. The performance of checksum operation works out to 13 cycles/Byte on the base Xtensa processor. Figure 4.8 shows the original checksum code, and the modified version with new instructions.

The speed of the checksum operation is clearly limited by the serial addition of 16-bit data fetched from the memory. By introducing two 64-bit custom registers, we can provide new instructions that enable the addition of multiple 16-bit numbers of the input data in parallel. We designed two custom instructions, ADD64() and FOLD64(). ADD64() implements the parallel addition of four 16-bit numbers with an accumulated sum, while FOLD64() performs the final fold operation. We define a custom load instruction to read in four 16-bit numbers from the memory into the custom register, and a custom store to write the final checksum result back to the memory. The performance of checksum improves to 9 cycles/Byte, providing a speed-up of 1.4X. The instructions added are similar to the sub-word parallelism instructions [86]. The semantics of the instructions added are as follows:

- **ADD64(r1,r2)**: $r1[63:0] = r1[63:0] + r2[63:0]$.
- **FOLD64(r1,r2)**: $r2[15:0] = r1[63:48] + r1[47:32] + r1[31:16] + r1[15:0]$


Original checksum code (source: RFC1071)
Checksum with custom instructions

```
/* Compute Internet Checksum for
   "count" bytes  beginning at location
   "addr". */
register long sum = 0;

while( count > 1 )
{
  /* This is the inner loop */
  sum += * (unsigned short) addr++;
  count = count - 2;
}
/* Add left-over byte, if any */
if ( count > 0 )
  sum += * (unsigned char *) addr;

/* Fold 32-bit sum to 16 bits */
while (sum>>16)
      sum = (sum & 0xffff) + (sum >> 16);
```

```
register long sum = 0;
while( count > 1 )
{
  LOAD(addr);
  ADD64();
  count = count - 8;
   addr = addr + 8;
}

/* Fold 32-bit sum to 16 bits */

FOLD64();
```

Figure 4.8: Software implementation of the checksum operation (a) without, and (b) with custom instructions

4.4.3.2 AES

In AES encryption, the input data are divided into uniformly-sized input blocks of 128 bits, and a round transformation is applied multiple times to each input block [87]. An input block is referred to as the *state*, and is represented by a 4×4 matrix of bytes. The number of iterations of a round transformation applied on an input block is parameterized on two values: the input block size (128 bits) and the key length (128, 196 or 256 bits). Using the key as the input, a function is used to generate a different sub-key (called a *round key*) for each iteration of the round transformation. The round transformation is made up of the following operations:

- **Substitution**: Each byte in the state is replaced by a byte in the S-box obtained by using the original byte as an index to perform the table lookup, *i.e.*, byte x is replaced by S[x].
- **Row shifting**: The rows of the state are cyclically shifted by fixed offsets.
- **Column mixing**: The columns of the state are considered to be polynomials over GF(2^8). They are multiplied by a polynomial, $3x^3 + x^2 + x + 3$ modulo $x^4 + 1$. This operation is equivalent to multiplying by matrix

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

- **Key addition**: Bytes of a round key (derived from the private key) are XORed with those of the state.

From the performance profile of AES shown in Figure 4.3, we see that *column mixing* is the most time-consuming operation in AES encryption, followed by *row shifting*, *substitution*, and *key addition* which take almost the same time. Thus, column mixing operation seems to be one which should be targeted for speeding up AES. However, instead of making only column mixing faster, we employ an optimization suited for 32-bit processors, which combines all the four functions, in the round transformation, into a set of table lookups, thereby, obtaining a much higher speed-up in AES encryption [87]. The four operations in the AES round transformation can be represented as

$$\begin{bmatrix} e_{1,j} \\ e_{2,j} \\ e_{3,j} \\ e_{4,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} S[a_{1,j}] \\ S[a_{2,j\_C1}] \\ S[a_{3,j\_C2}] \\ S[a_{4,j\_C3}] \end{bmatrix} \oplus \begin{bmatrix} k_{1,j} \\ k_{2,j} \\ k_{3,j} \\ k_{4,j} \end{bmatrix}$$

where $\{e_{1,j}, e_{2,j}, e_{3,j}, e_{4,j}\}$, $\{a_{1,j}, a_{2,j}, a_{3,j}, a_{4,j}\}$, and $\{k_{1,j}, k_{2,j}, k_{3,j}, k_{4,j}\}$ represent the first, second, third, and fourth rows of the output, the state (input) and the round key, respectively. The terms a_{2,j_C1}, a_{3,j_C2}, and a_{4,j_C3} indicate left-rotation of the second, third and fourth rows of the state by pre-determined amounts $C1$, $C2$, and $C3$, respectively. In AES, the values of $C1$, $C2$, and $C3$ are one, two and three, respectively. For example, let $a_{2,j} = (a_{2,1}, a_{2,2}, a_{2,3}, a_{2,4})$ be the second row of the state matrix. It has four entries $a_{i,j}$, $1 \le j \le 4$. Then, $a_{2,j\_C1} = (a_{2,2}, a_{2,3}, a_{2,4}, a_{2,1})$. $S[a_{i,j}]$ is the result of S-box substitution of byte $a_{i,j}$ (S-box S[] has 256 entries). For example, in the equation above, $S[a_{1,j}]$ represents substitution of all the bytes in the first row. The above equation can be rearranged as

$$\begin{bmatrix} e_{1,j} \\ e_{2,j} \\ e_{3,j} \\ e_{4,j} \end{bmatrix} = S\left[a_{1,j}\right] \begin{bmatrix} 2 \\ 1 \\ 1 \\ 3 \end{bmatrix} \oplus S\left[a_{2,j\_C1}\right] \begin{bmatrix} 3 \\ 2 \\ 1 \\ 1 \end{bmatrix} \oplus S\left[a_{3,j\_C2}\right] \begin{bmatrix} 1 \\ 3 \\ 2 \\ 1 \end{bmatrix} \oplus S\left[a_{4,j\_C3}\right] \begin{bmatrix} 1 \\ 1 \\ 3 \\ 2 \end{bmatrix} \oplus \begin{bmatrix} k_{1,j} \\ k_{2,j} \\ k_{3,j} \\ k_{4,j} \end{bmatrix}$$

The multiplication of $S[a_{i,j}]$ and the column of constants can be converted into a table lookup with 256 entries by pre-computing the results for all the 256 possible values of byte

$a_{i,j}$. Thus, the four multiplications can be converted to four table lookups defined as follows.

$$T_1[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] * 2 \\ S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] * 3 \end{bmatrix}, T_2[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] * 3 \\ S[a_{i,j}] * 2 \\ S[a_{i,j}] \\ S[a_{i,j}] \end{bmatrix},$$

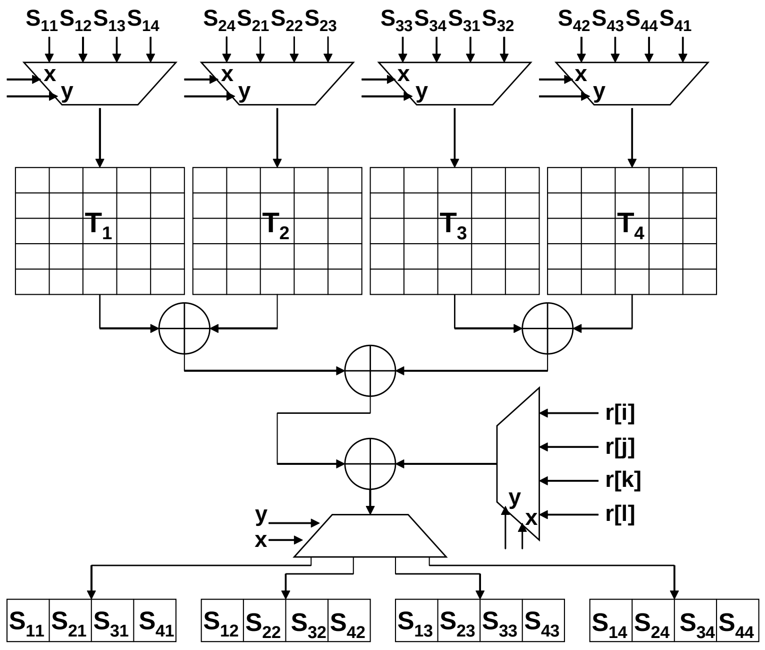
$$T_3[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] * 3 \\ S[a_{i,j}] * 2 \\ S[a_{i,j}] \end{bmatrix}, T_4[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] * 3 \\ S[a_{i,j}] * 2 \end{bmatrix}$$

Thus, the round transformation on one column of the state matrix can be implemented with four table lookups and four XORs as shown below.

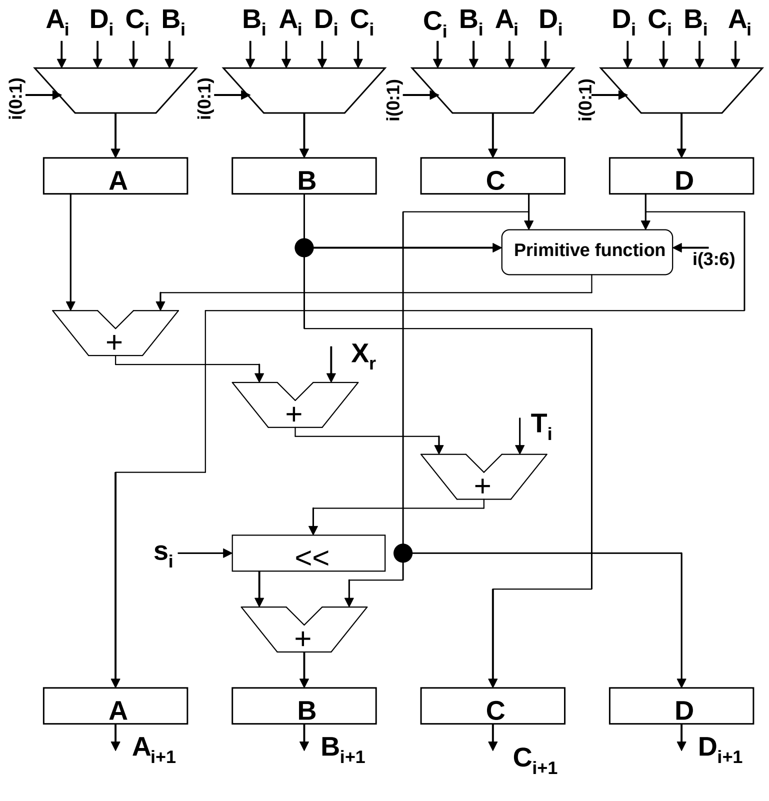
$$\begin{bmatrix} e_{1,j} \\ e_{2,j} \\ e_{3,j} \\ e_{4,j} \end{bmatrix} = T_1[a_{1,j}] \oplus T_2[a_{2,j}] \oplus T_3[a_{3,j}] \oplus T_4[a_{4,j}] \oplus \begin{bmatrix} k_{1,j} \\ k_{2,j} \\ k_{3,j} \\ k_{4,j} \end{bmatrix}$$

The round transformation on the 4×4 *state* matrix is realized using 16 table lookups and 16 XOR operations. This optimization greatly speeds up the rate of AES encryption/decryption.

We introduce four hardware tables filled with pre-computed values, and a custom instruction to implement the round transformation. Figure 4.9(a) shows the functional view of the custom instruction. The custom instruction takes the state as an input, and operates on it column-by-column. S_{ij} indicates the byte in the ith row and jth column of the state. The custom instruction is multi-cycled, and takes one column of the state as its input. Select signals of the multiplexers (x, y) are set to appropriate values, in order to select bytes of the correct column. Outputs of table lookups are XORed with each other, and the result is XORed with the round key (variable r in the figure). A similar instruction, with its own set of four hardware tables, is implemented for decryption. In addition, load and store instructions are defined to read in data and write out results.



(a)



(b)

Figure 4.9: Functional view of custom instructions (a) ENC_ROUND, and (b) MD5_ITERATION in AES and MD5, respectively

The semantics of the instructions added to speed up AES execution are as follows:

- **ENC_ROUND(S,r,i,j,k,l)**:
  $S_1 = r[i] \oplus T1[S_{11}] \oplus T2[S_{24}] \oplus T3[S_{33}] \oplus T4[S_{42}]$,
  $S_2 = r[j] \oplus T1[S_{12}] \oplus T2[S_{21}] \oplus T3[S_{34}] \oplus T4[S_{43}]$,
  $S_3 = r[k] \oplus T1[S_{13}] \oplus T2[S_{22}] \oplus T3[S_{31}] \oplus T4[S_{44}]$,
  $S_4 = r[l] \oplus T1[S_{14}] \oplus T2[S_{23}] \oplus T3[S_{32}] \oplus T4[S_{41}]$
  (where, S_1, S_2, S_3, S_4 are the first, second, third and fourth columns of the state, and $T1$, $T2$, $T3$ and $T4$ are hardware tables).

- **DEC_ROUND(S,r,i,j,k,l)**: It has a structure similar to $ENC\_ROUND()$. This instructions are similar to the fast table lookups proposed in [88, 89]. The suggestions for table lookup based optimizations for AES were originally mentioned in the specification of Rijndael algorithm which was later renamed as AES.

The performance of the software implementation of AES is 415 cycles/Byte, while the use of custom instructions improves the performance to 102 cycles/Byte. Since the size of the input block is 128 bits (16 bytes), the cycles consumed per round of AES in the optimized version is $(16 * 102)/10 = 163$ cycles. This includes the overhead for loading the state from memory into the custom registers initially, loading in the round key before every round execution, and finally, reading out the state. This translates to 9 cycles per table lookup which results in 144 cycles for the 16 lookups in a round operation, and the remaining 19 cycles for XOR operations. Thus, the addition of custom instructions provides a speed-up of 4.1X in AES encryption. The area overhead of this instruction can be reduced by defining a single table, *i.e.*, $T1$, instead of four. The original inputs to tables $T2$, $T3$, and $T4$ are rotated by fixed amounts and used to look up table $T1$ in order to get the same outputs. Thus, additional cycles are needed for the extra 12 rotate operations per each round. Assuming, a rotate operation takes four cycles (as per estimates on the Xtensa processor), number of cycles needed per round of AES would increase to around 224 cycles.

4.4.3.3 MD5

MD5 takes input data of arbitrary length, and outputs a fixed-length message digest of 128 bits [4]. It breaks up the input data into blocks of 512 bits each, and processes each block sequentially using a compression function. At each step, the compression function takes two inputs, a 512-bit input block and a 128-bit block (output by the application of compression on the previous 512-bit input block), and outputs a 128-bit block. The 128-bit block obtained after compressing the last 512-bit input block is the output message digest. The compression function comprises four rounds, each having a similar structure, but parameterized on different primitive sub-functions. Each round consists of 16 iterations of its corresponding functionality implemented using the primitive function. Each iteration is parameterized on a 32-bit constant value. The 128-bit input block is broken into four 32-bit words, A, B, C, and D, which are successively modified by each iteration. The primitive functions used in the four rounds are given by $(x \wedge y) \vee (\overline{x} \wedge z)$, $(x \wedge z) \vee (y \wedge \overline{z})$, $x \oplus y \oplus z$, and $y \oplus (x \vee \overline{z})$, respectively, where x, y, and z are 32-bit inputs to the primitive functions.

According to the performance profile of MD5 shown in Figure 4.3, *Final* and *Update* functions dominate its computation. Both of these functions implement the MD5 compression function described above. Thus, compression function is the dominant operation in MD5 and, it is targeted for custom instruction addition. We define a custom instruction MD5_ITERATION that implements any iteration from among the 4×16 iterations comprising the compression function. MD5_ITERATION implements the functionality shown in Figure 4.9(b). The semantics of this instruction are as follows:

- **MD5_ITERATION**$(i, A_i, B_i, C_i, D_i, X_r, T_i, s_i)$:
  $(A_{i+1}, B_{i+1}, C_{i+1}, D_{i+1}) = F_i(i, A_i, B_i, C_i, D_i, X_r, T_i, s_i)$, where F_i is the function performed by the circuit shown in Figure 4.9(b).

Each iteration takes as its input the current message digest state, as given by A_i, B_i, C_i and D_i, a 32-bit subset (X_r) of the 512-bit input block, and a constant value, T_i, corresponding to that iteration, chosen from a table of pre-computed constants. In each round, the appropriate pre-defined non-linear primitive function g is used. Each iteration outputs updated values A_{i+1}, B_{i+1}, C_{i+1}, D_{i+1}, which are passed on to the next iteration. A load instruction is defined to read in words of the 512-bit input block.

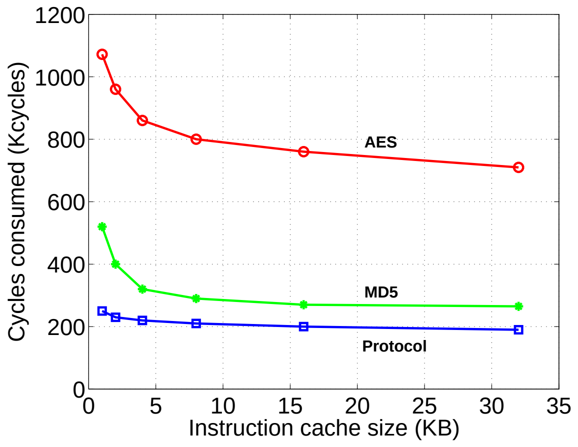
The performance of the software implementation of MD5 is 105 cycles/Byte, while the use of custom instructions improves the performance to 34 cycles/Byte (a speed-up of 3.1X). Thus, the optimized version requires 2176 cycles to process a 512-bit input ((512/8)*34). This includes the overhead of the reads and writes to the custom registers. Each of the four rounds takes 544 cycles, and each of the 16 iterations within a round takes 34 cycles.

4.4.4 Impact of configurable options and interplay with extensibility

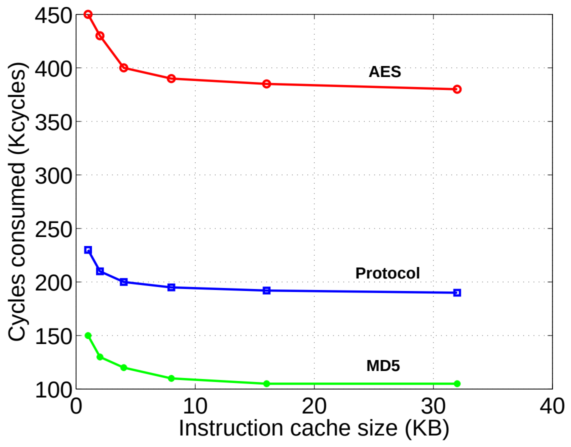
In this section, we study the influence of configurable parameters on protocol and cryptographic processing in IPSec, both in the presence and absence of the extensible options described in the previous subsection. We independently vary the value of each configurable option, and study its effect on the performance of IPSec processing for the same workload. We present results that are averaged over multiple workloads. The results are given in terms of number of cycles consumed *per packet* for protocol processing, encryption using AES in ESP, and hashing with MD5 in AH. We use the **base processor configuration** defined as *1KB direct-mapped instruction and data caches with 16 Byte lines, 32 registers, 4 write buffer slots, and 32-bit PIF width*, as the basis for evaluating the effect of varying the configurable parameters. In the following subsections, we describe the results of our investigations.

4.4.4.1 Cache size

Figures 4.10(a) and 4.10(b) show the effect of increasing the I-cache size on protocol and cryptographic processing (AES and MD5) in the absence and presence of the extensible options, respectively. Without the extensible options, the performance of AES and MD5 improves steadily with increasing I-cache sizes. The inclusion of custom instructions reduces the impact of the I-cache size on AES and MD5 processing. This is because each custom instruction replaces a significant number of original instructions (the AES and MD5 code size reduces by 40% and 30%, respectively). Inclusion of custom instructions does not alter the impact of I-cache on the performance of protocol processing. This is because the effect of the checksum custom instruction on protocol processing code size is relatively small. Table 4.2 gives the processor cycles for varying I-cache sizes.



(a)



(b)

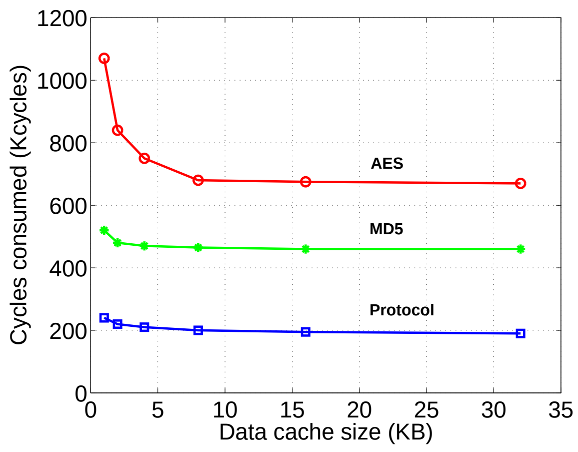
Figure 4.10: Effect of instruction cache size on IPSec performance in (a) the absence and (b) presence of extensible options

Table 4.2: Effect on I-cache size on performance

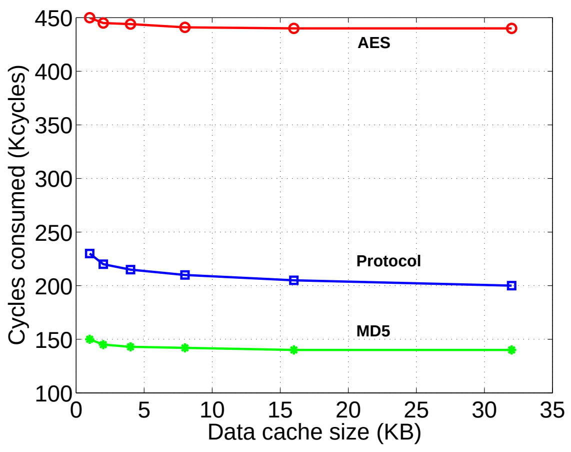
I-cache size						
	Unoptimized			Optimized		
Cache size (KB)	AES	MD5	Protocol	AES	MD5	Protocol
1	1072	520	250	450	150	230
2	960	400	230	430	130	210
4	860	320	220	400	120	200
8	800	290	210	390	110	195
16	760	270	200	385	105	195
32	710	265	190	380	105	190

The influence of D-cache size on the performance of IPSec cryptographic and protocol processing without and with the extensible options is illustrated in Figures 4.11(a) and 4.11(b), respectively. The AES implementation is based on table lookups, and requires 8KB of tables altogether (4KB for encryption, and 4KB for decryption). The computational kernel of MD5 is a loop which comprises 64 operations using 2KB of constant values. Thus, in Figure 4.11(a), we see that the gain in performance levels out at 8KB and 2KB for

AES and MD5, respectively. There is a modest improvement in protocol processing until a 4KB D-cache size, beyond which performance saturates. With the introduction of custom instructions (Figure 4.11(b)), we can see that the impact of D-cache size on AES and MD5 becomes negligible. The custom instructions for AES and MD5 use either hardware tables or define special state registers to hold constant values. Hence, the corresponding memory accesses are eliminated. Table 4.3 shows the effect of D-cache size on performance.



(a)



(b)

Figure 4.11: Effect of data cache size on IPSec performance in (a) the absence and (b) presence of extensible options

Table 4.3: Effect on D-cache size on performance

D-cache size						
	Unoptimized			Optimized		
Cache size (KB)	AES	MD5	Protocol	AES	MD5	Protocol
1	1070	520	240	450	150	230
2	840	480	220	445	145	220
4	750	470	210	444	143	215
8	680	465	200	441	142	210
16	675	460	195	440	140	205
32	670	460	190	440	140	200

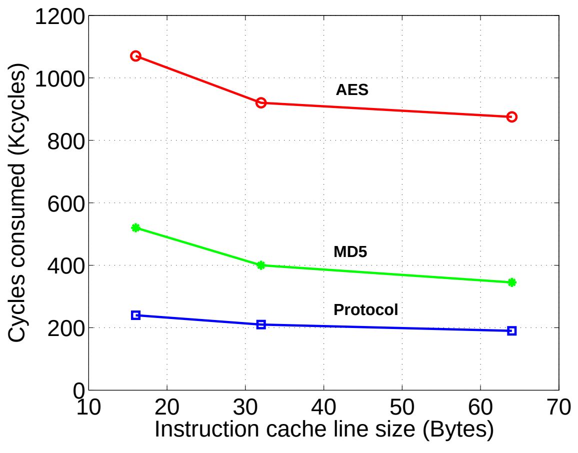
4.4.4.2 Cache associativity

Changing the associativity of the I-cache has no effect on the performance of protocol and MD5 processing (with and without the extensible options). In the case of AES, in the absence of custom instructions, its performance improves by 30% with an increase in associativity indicating the presence of conflicts among instructions. Inclusion of custom instructions greatly reduces the impact of associativity.

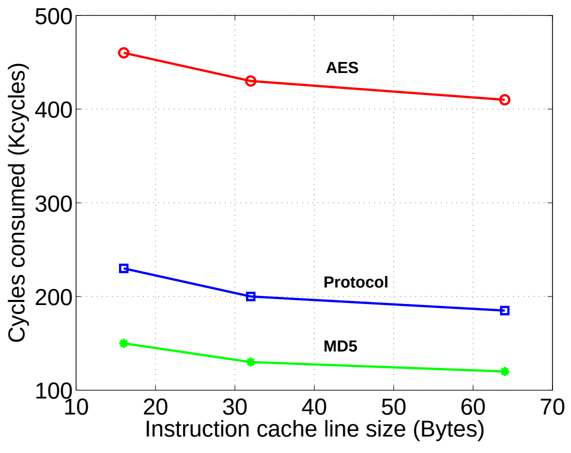
Increasing the D-cache associativity improves the performance of AES (without custom instructions) by 10%. However, as in the case of I-cache associativity, the introduction of custom instructions makes AES processing independent of D-cache associativity. An increase in D-cache associativity improves the performance of protocol processing by 10% (both with and without the checksum custom instruction). This indicates possible data conflicts in protocol processing code that are ameliorated by increasing associativity.

4.4.4.3 Cache line size

Increasing the I-cache line width (from 16B to 64B) improves the performance of protocol processing (20%) and MD5 (35%) more than AES (16%) in the absence of custom instructions (Figure 4.12(a)). Code for protocol processing and MD5 have fairly large basic blocks. Thus, increasing the line width exploits the inherent spatial locality and thereby reduces the number of compulsory misses. The custom instructions greatly reduce the impact of I-cache line size on AES and MD5 (Figure 4.12(b)). Protocol processing gets an appreciable performance gain from increased line width even in the presence of custom instructions. Table 4.4 shows the effect of I-cache line size on performance.



(a)



(b)

Figure 4.12: Effect of instruction cache line size on IPSec performance in (a) the absence and (b) presence of extensible options

Table 4.4: Effect on I-cache line size on performance

I-cache line size						
	Unoptimized			Optimized		
Line size (B)	AES	MD5	Protocol	AES	MD5	Protocol
16	1070	520	240	460	150	230
32	920	400	210	430	130	200
64	875	345	190	410	120	185

D-cache line width has no effect on the performance of MD5 and AES in the presence of custom instructions (Figure 4.13(b)). In the absence of custom instructions, the performance of AES improves with increasing line width (Figure 4.13(a)), thus hinting at spatial locality within the data (due to AES encryption being applied sequentially on 128-bit blocks making up the packet payload). The performance of protocol processing improves modestly with increasing D-cache line width. Protocol processing involves performing operations on a packet buffer. Therefore, increasing the line width can result in better utilization of the spatial locality within the data. Table 4.5 shows the effect of D-cache line size on performance.

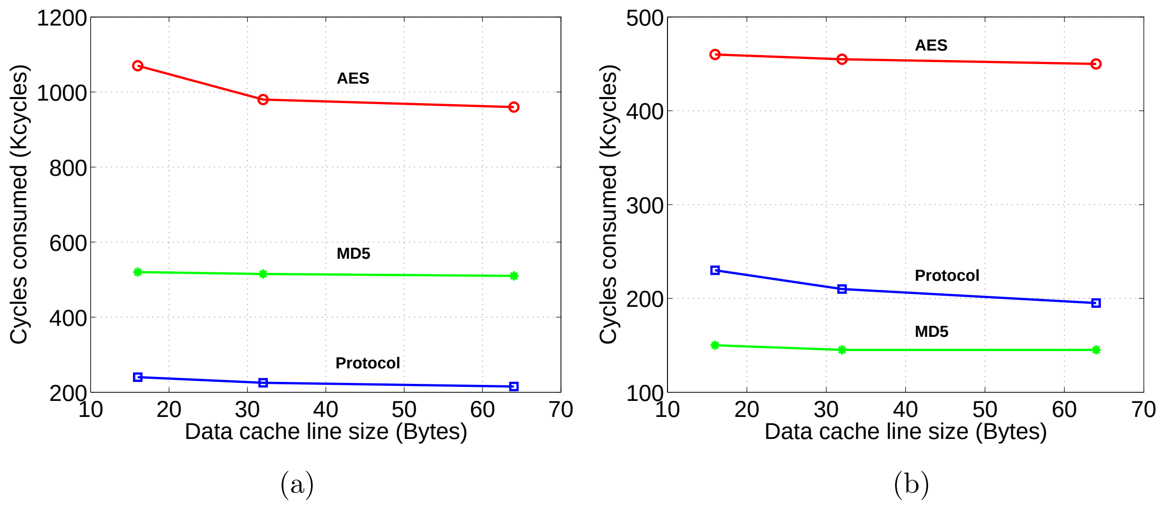


Figure 4.13: Effect of data cache line size on IPSec performance in (a) the absence and (b) presence of extensible options

4.4.4.4 Register file

The Xtensa processor uses windowed registers to improve processor performance. Such registers reduce the interrupt response time by eliminating the register saves and restores

Table 4.5: Effect on D-cache line size on performance

D-cache line size						
	Unoptimized			Optimized		
Line size (B)	AES	MD5	Protocol	AES	MD5	Protocol
16	1070	520	240	460	150	230
32	980	515	225	455	145	200
64	960	510	215	450	145	195

at procedure entry and exit points. Also, it allows more local variables to remain in the registers, thereby decreasing the time spent in stack frame maintenance. Increasing the register file size from 32 to 64 reduces the number of cycles spent in servicing window overflows and underflows by nearly 80%. However, overall, it results in only a modest decrease of 5% in the number of cycles consumed by protocol and cryptographic processing.

4.4.4.5 Write buffer depth

Increasing the slots in the write buffer from 4 to 32 reduces the protocol cycle count by 10%. It has no influence on MD5 and AES algorithm performance. Study of the protocol processing performance shows that nearly 30% of the cycles are spent waiting for write buffer slots to become available. This could be due to bursts of stores that are generated by the protocol processing code at a higher rate than the memory bandwidth, exhausting the write buffer slots. Thus, increasing the number of write buffer slots reduces this delay, and leads to an improvement in performance.

4.4.4.6 PIF width

The PIF width determines the size of blocks in which data are transacted between the processor and main memory on a bidirectional interface. Thus, increasing the PIF width from 32 to 128 bits improves the memory performance by allowing faster refilling of the cache lines being fetched from the memory. Furthermore, the performance of extensible instructions also improves by increasing the PIF width to 128 bits. The custom instructions in AES, MD5 and checksum operate on blocks of 128 bits, 128 bits and 64 bits, respectively. Thus, increasing the PIF width allows faster transfer of data from the memory. The performance of AES and MD5 improves by 10%, while the performance of protocol processing improves by nearly 30%.

4.4.5 Summary

In the previous two sub-sections, we investigated the effect of configurable and extensible parameters on IPSec-related protocol processing, MD5 (AH) and AES (ESP). As part of our investigation, the extensible parameters were implemented as custom instructions to replace the hotspots in IPSec-related protocol and cryptographic processing. With the extensible additions in place, we studied the configurable parameter values that would result in optimal performance. The configurable values identified are *4KB, 4-way set-associative I-cache with 64-byte lines*; *4KB, direct-mapped D-cache with 64-byte lines*; *32 write buffer slots*, *register file with 32 registers*, and *128-bit PIF*. The area overhead of this configuration over the base processor is 1.8X.

Table 4.6: Performance speedup of IPSec due to extensibility and configurability

Option	Protocol	MD5	AES
Configurable	2X	1.2X	1.3X
Extensible	1.1X	3.1X	4.1X

Table 4.6 gives a summary of the amount by which IPSec-related protocol and cryptographic processing benefit from extensibility and configurability. Protocol processing benefits much more from configurability than extensibility. The opposite holds true for cryptographic processing. This is intuitive, since MD5 and AES are dominated by computations which can be suitably targeted by introduction of custom instructions, whereas protocol processing is dominated by memory-related issues which are best addressed through configurability rather than extensibility.

Figure 4.14 shows the results of extensibility and configurability on the overall execution of the IPSec protocol in the AH and ESP modes. The cycles consumed originally by the AH and ESP modes are normalized to 100 (first and third bars in Figure 4.14, respectively). The AH and ESP cycles consumed are divided into those spent on cryptographic and protocol processing. The cycles consumed by the optimized AH and ESP modes are normalized to

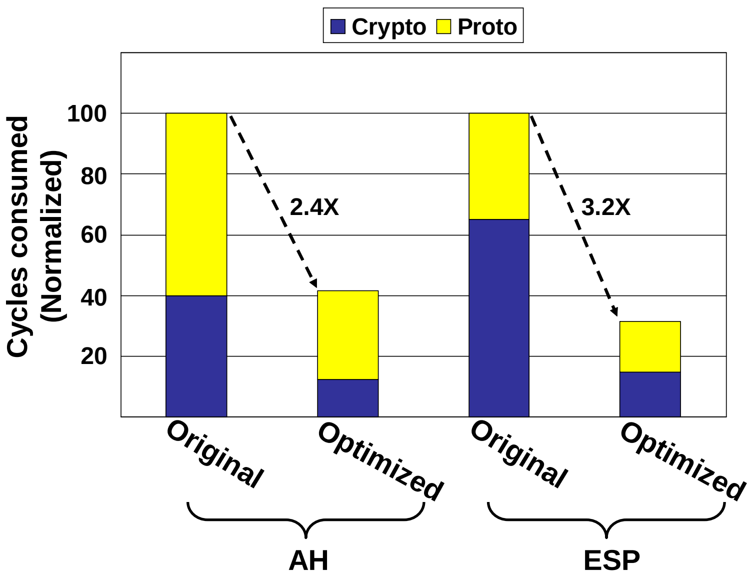


Figure 4.14: Speed-up of IPSec AH and ESP modes

cycles originally consumed by the AH and ESP modes, respectively, in order to show the overall speedup. The second and fourth bars in Figure 4.14 show the results for optimized AH and ESP, respectively. We can see that AH is sped up by 2.4X and ESP by 3.2X.

4.5 Chapter summary

In this chapter, we studied how the configurable and extensible parameters of a state-of-the-art embedded processor affect the performance of IPSec running on it. Using a detailed analysis framework, we identified values for the configurable parameters, and introduced custom instructions to target compute-intensive IPSec tasks. The custom processor thus obtained provides a maximum speed-up of 3.2X for IPSec processing.

Chapter 5

Satisfiability-based Side-channel Attack Framework for Embedded Cryptographic Software

Many electronic systems contain implementations of cryptographic algorithms in order to provide security. It is well known that cryptographic algorithms, irrespective of their theoretical strength, can be broken through weaknesses in their implementations. In particular, side-channel attacks, which exploit unintended information leakage from the implementation, have been established as a powerful way of attacking cryptographic systems. All side-channel attacks can be viewed as consisting of two phases — an observation phase, wherein information is gathered from the target system, and an analysis or deduction phase in which the collected information is used to infer the cryptographic key. Thus far, most side-channel attacks have focused on extracting information that directly reveals the key, or variables from which the key can be easily deduced.

We propose a new framework for performing side-channel attacks by formulating the analysis phase as a search problem that can be solved using modern Boolean analysis techniques such as satisfiability (SAT) solvers. This approach can substantially enhance the scope of side-channel attacks by allowing a potentially wide range of internal variables to be exploited (not just those that are directly related to the key). For example, software implementations take great care in protecting secret keys through the use of on-chip key generation and storage. However, they may inadvertently expose the values of intermediate variables in their computations. We demonstrate how to perform side-channel attacks on software implementations of cryptographic algorithms based on the use of a satisfiability solver for reasoning about the secret keys from the values of the exposed variables. Our attack technique is automated, and does not require mathematical expertise on the part of the attacker. We demonstrate the merit of the proposed technique by successfully applying it to three popular cryptographic algorithms, DES, 3DES, and AES. This chapter is based on results presented in [54, 55].

5.1 Introduction

Security has emerged as a critical concern in a wide range of electronic systems. Extensive experience with the use and deployment of security technologies has shown that, in practice, most security systems are broken by exploiting weaknesses in their implementations, making it important to consider security during the complete design process.

Cryptographic primitives, such as encryption and hashing algorithms, form the basis of most security mechanisms. A cryptographic system may be abstracted as a mathematical function that performs a given mapping of its input to its output. However, in reality, it should be viewed as a specific (hardware or software) implementation of the mathematical function. *Cryptanalysis* refers to the process of breaking a cryptographic system without a brute-force search (*e.g.*, for an encryption algorithm, deriving the n-bit key without 2^n operations). Traditionally, cryptanalysis has focused on just the mathematical function underlying the system, *e.g.*, by analyzing statistical properties of the outputs under the application of targeted inputs [44, 45]. However, many of these attacks are infeasible in practice due to the large amount of data required to implement them.

More recently, a powerful class of attacks, called *side-channel attacks*, has emerged, which exploits information from the implementation to substantially reduce the complexity of performing cryptanalysis [46–49, 90]. Side-channel attacks can be viewed as consisting of two phases — an observation phase, wherein information is gathered by monitoring a 'side-channel' in the target system, and an analysis or deduction phase in which the collected information is used to infer the cryptographic key. Information leakage through a side-channel is an inadvertent by-product of the implementation process. Examples of side-channel information used in successful attacks are operation timing [46], power dissipation [47, 48], electromagnetic radiation [90], and behavior in the presence of induced faults [49]. Surprisingly small amounts of leaked information are sufficient to break the secret key [52]. A wide range of design techniques have been proposed to counter side-channel attacks [91–93]. With the use of such techniques, the presence of side-channels can be minimized. However, it is very difficult to completely eliminate them [53].

While some of the early side-channel attacks targeted hardware implementations, software implementations are equally, if not more, vulnerable. Data exposure can occur in software implementations through memory bus exposure, core dump files, persistence of data in disk memory after swap, *etc.* [94]. This problem of data exposure exists even in secure software implementations [95]. Recent studies have revealed the possibility of data exposure from software computations even after the computation is over [96]. In some instances, even sensitive data, like passwords, were left in accessible system buffers. Software side-channels typically reveal data in bytes or (larger) words, making them especially attractive targets for attacks.

In this work, we propose a framework for side-channel attacks by formulating the analysis phase as a Boolean search problem and solving it using state-of-the-art SAT solvers. We demonstrate this approach in the context of software side-channel attacks. Our approach substantially enhances the scope of side-channel attacks by allowing a potentially wide range of internal variables to be exploited (not just those that are trivially related to the key). The exposure of secret keys or variables that are directly related to them leads to a trivial compromise of security. For example, in the DES algorithm, knowledge of the inputs to each S-box in a round will allow the attacker to trivially calculate the key. Therefore, secret keys and other "easy targets" are often protected from exposure, *e.g.*, through the use of protected on-chip key generation and storage [97]. However, seemingly harmless variable values, if exposed, can be sufficient to deduce the secret keys when powerful analysis techniques, such as the SAT solver used in this work, are employed.

The Boolean SAT problem is defined as follows. Given a Boolean formula made up of a conjunction of clauses, each of which is a disjunction of Boolean literals, determine whether values can be assigned to the literals such that all the clauses in the formula are satisfied, *i.e.*, evaluate to 1. Such a literal assignment is referred to as the *satisfying assignment*. The function of a SAT solver is to find a satisfying assignment for any given Boolean formula, if one exists, else give a proof that no such assignment is possible. While SAT has been shown to be NP-complete, efficient heuristics exist that can solve many real-life SAT formulations. Furthermore, the many applications of SAT have motivated advances in SAT solving techniques that have been incorporated into freely-available SAT software tools [98, 99]. Many practical search problems in a wide range of areas have been formulated as SAT problems. In the field of design automation, SAT has been successfully applied in hardware and software verification and circuit testing. Given the versatility and effectiveness of SAT solving techniques, it was intuitive to use a SAT solver as an automated reasoning engine in our proposed framework for enabling side-channel attacks.

5.1.1 Work contributions

The contributions of this work include:

- A general framework for enabling side-channel attacks by formulating the analysis of side-channel information as a search problem that can be solved using SAT solvers. This approach is fully automatic and obviates the need for mathematical expertise on the part of the attacker.

- Demonstration that a large subset of the internal variables in a cryptographic algorithm, not just the key or variables that are directly related to it, can be used to launch successful attacks.

- Application of the proposed framework to perform software side-channel attacks on the popular symmetric encryption algorithms, DES, 3DES and AES.
- Characterization of the minimal subsets of internal variables that are sufficient to break DES, 3DES and AES, given current state-of-the-art SAT solvers.

Previous work in the area of side-channel attacks has identified various side-channels, and proposed specific *ad hoc* techniques to exploit the information derived from each of them. However, due to the nature of the collected information, the analysis phase has mostly been quite simple. In the context of software attacks, existing work gives ample evidence of software data leakage. However, there is no general framework to transform these vulnerabilities into actual attacks on security software. Furthermore, when implementations take basic measures to protect the keys and other directly related variables from leakage, more powerful analysis techniques, such as the one proposed in our work, are necessary. From another perspective, a knowledge of the internal variables that can be used to launch side-channel attacks can translate into design guidelines that point to parts of the implementation that should be protected. To the best of our knowledge, this is the first attempt to apply Boolean analysis techniques to side-channel attacks.

The rest of the chapter is organized as follows. Section 5.2 discusses the various ways in which software side-channels are created, thereby enabling side-channel attacks. Section 5.3 gives an overview of our proposed technique, and illustrates the formulation of a cryptographic algorithm as a Boolean formula, using DES, 3DES and AES as examples. Section 5.4 presents results of our comprehensive experiments with a state-of-the-art SAT solver to identify the intermediate values in DES, 3DES and AES which enable successful inferring of the key. We conclude in Section 5.5 with our observations and directions for future work.

5.2 Motivation

In this section, we motivate the prevalence of leakage of intermediate values from software computations. We begin by enumerating the various techniques employed in practice to obtain the intermediate values. We conclude the section by giving details of the scheme employed in our work to obtain the intermediate values required by our satisfiability-based cryptanalysis framework to cryptanalyze cryptographic algorithms.

5.2.1 Leakage of software variable values

The architecture of a typical hardware-software system implementing a cryptographic computation is shown in Figure 5.1. For a given value of the secret key, it injectively maps a plaintext to a ciphertext. The architecture is divided into software and hardware sub-systems. The software sub-system comprises application programs layered on top of the system libraries and the operating system (OS). System libraries make commonly used software routines available to an application code through a function call interface while a system call interface enables an application to access hardware resources via the OS. The OS interacts with the hardware sub-system using machine instructions. The hardware component consists of the processor, input-output devices, and the hierarchical memory system made up of an on-chip cache, main memory and magnetic disk storage. The processor has a dedicated on-chip memory which stores the cryptographic key thereby preventing exposure of the key bits through operations (*e.g.,* swapping) which cause the key bits to be transmitted outside the processor's secure perimeter (usually onto the main memory bus) [94]. However, the complexities involved in implementing such hardware-software systems opens avenues for intentional or unintentional leakage of data values at the various interfaces present: application-library, application-OS, library-OS, and OS-hardware.

Below we present specific cases of such data leakage which were observed in practice (indicated in Figure 5.1):

- *Unintentional leakage*: This happens inadvertently during normal operation due to bugs, improper policies, misconfiguration, *etc.* Chow *et al.* [96] showed the existence of program data in system buffers in main memory long after the program terminated. This included sensitive data, like passwords. Swapping of program data to disks greatly increases the probability of exposure due to the data retention property of

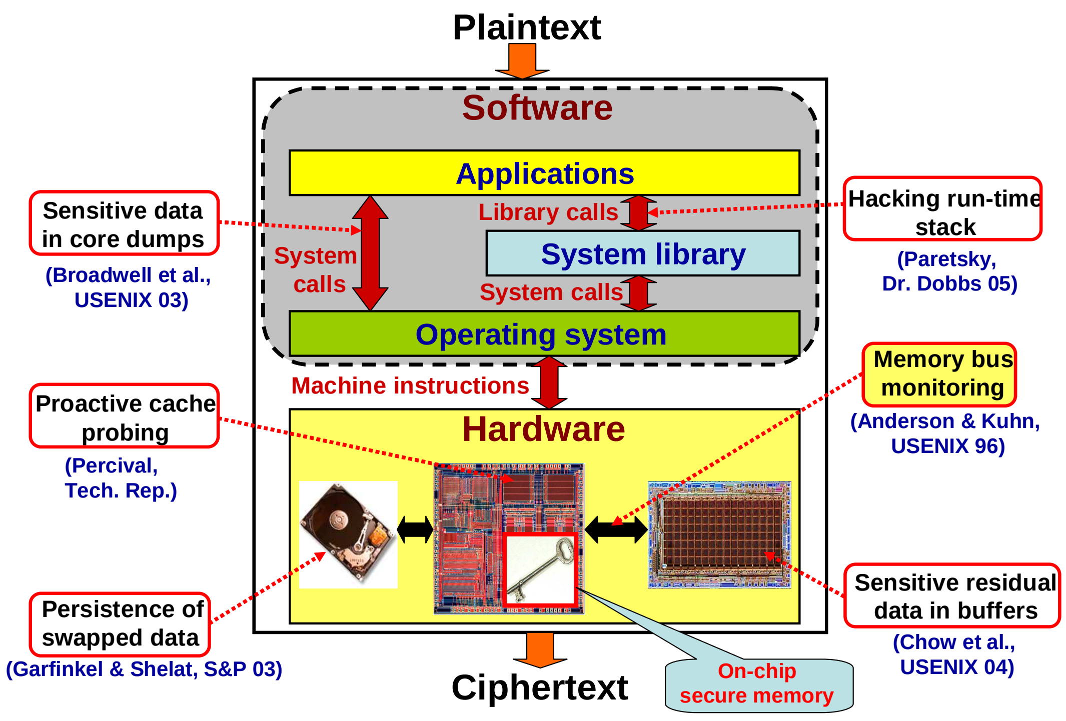


Figure 5.1: Leakage of software variable values in hardware-software systems

magnetic disks. Garfinkel and Shelat [100] showed concrete evidence of persistence of data on disks, and presented ways to extract the data. Core dumps occur when an application program crashes after performing an illegal operation, and are used for finding out the cause of failure. This diagnostic information can expose program data [94]. Broadwell *et al.* showed presence of sensitive information in the system crash reports generated by a widely used OS [101].

- *Intentional leakage*: This is precipitated by attacks which are crafted to exploit latent system vulnerabilities. A common ploy is to break a system by forcing it to implement operations other than the ones it was designed for. Examples of such malicious hardware or software attacks include hacking the run-time stack [102], proactively probing the cache using a Trojan process [103], monitoring the memory traffic on the system bus [104], *etc.* Also, there exist tools for dynamically examining the contents of program memory as the program is being executed [105].

Even software systems that have been implemented with the aim of maintaining security of the data being manipulated have been shown to have security breaches [106]. Thus, it is very hard to design hardware-software systems which completely prevent any kind of data exposure.

5.2.2 Proactive memory bus monitoring-based leakage

In this section, we illustrate the leakage technique we use to obtain the intermediate values which were used in our cryptanalysis framework. Our leakage technique is based on proactive manipulation of the data cache coupled with bus monitoring, in order to gather the intermediate values of the cryptographic computation. However, it should be stressed that our SAT-based framework is independent of the methods employed to leak the intermediate values of computation.

A cache is an on-chip buffer which stores recent memory accesses of a program with the aim of improving performance by exploiting spatial and temporal locality of the program. It consists of multiple storage blocks called *cache lines*, each capable of holding B bytes. Furthermore, the cache lines are partitioned into M equal-sized sets known as *cache sets*. For an N-way set-associative cache, each cache set has N cache lines. Typically, N takes values one (direct-mapped), two (two-way set-associative) or four (four-way set-associative). Thus, the total capacity of the cache is $M.N.B$ bytes. Each memory block is mapped to a specific cache set using the formula, $\lfloor A/B \rfloor \, mod \, M$, where A is the address of the memory block. Unless it is a software-controlled cache, an application program has no explicit control over the cache, *i.e.*, it cannot dictate how its data should be placed in the cache, and for how long. We simulated a standard software implementation of DES using a popular memory profiler [107] and observed the percentage of intermediate values in the DES computation that appeared on the memory bus (quantified by data cache misses) as a function of the data cache size. The results are shown in Figure 5.2. The x-axis plots the data cache size in KBytes, and the y-axis plots the miss ratio of all the intermediate values of DES computation. From the plot we can see that when the data cache size is increased from 1KB to 2KB, the percentage of intermediate variables appearing on the memory bus falls drastically, and is reduced to an insignificant percentage beyond 2KB.

Therefore, it is essential that we have a scheme that will proactively engineer cache misses of the intermediate variables of cryptographic computation which can in turn be recorded by monitoring traffic on the memory bus.

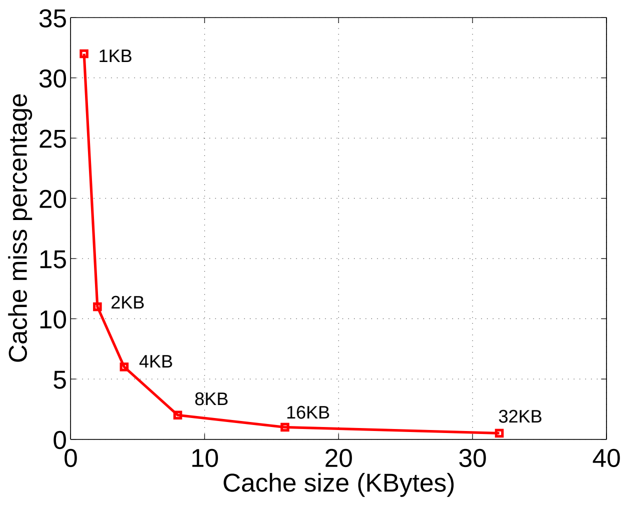


Figure 5.2: Probability of DES intermediate value exposure as a function of cache size

Modern computer systems employ a variety of sophisticated hardware and software mechanisms to enforce isolation of processes running simultaneously on the system. However, one low-level hardware resource, where execution of different processes intersects, is the cache. Though a process cannot read the contents of cache lines holding data belonging to another process, it can influence the execution of the other process. For example, it can increase the number of cache misses of the other process [103]. This is possible because the number of cache lines is much smaller than the number of lines in the main memory, thereby causing multiple memory lines to contend for a single cache line. Suppose $X = \lfloor Y/B \rfloor \, mod \, M$, where B and M are as defined above, and X and Y are cache line and memory line addresses, respectively. Then all the memory lines whose addresses are of the form $\{Y \,|\, Y = k.M + X.B,\ k \in N\}$ map to the cache line having address X. Thus, by reading or writing data into certain memory locations, a process could cause data belonging to another process to be evicted from the cache.

Symmetric cryptographic algorithms, which are targeted in the present work, iterate a set of operations (collectively termed *round operation*) a fixed number of times, say R, over the input plaintext, in order to produce the ciphertext. The proposed cryptanalysis framework requires intermediate values generated in a round operation. The algorithm, $COLLECT\_VALUES$, for gathering the intermediate values from the computation of the cryptographic algorithm is shown in Figure 5.3. Its inputs are the cryptographic algorithm P, the total execution time T of the algorithm, and the number of rounds R in algorithm P. The algorithm interrupts execution of the cryptographic algorithm at the end of each of round operation, and uses algorithm $FLUSH\_CACHE$ (described ahead), to evict the intermediate values of cryptographic computation from the data cache and record them by monitoring the memory bus.

```
COLLECT_VALUES (Cryptographic algorithm P, Execution time T, Rounds R)
1 : $timer = 0$
2 : $i = 1$
3 : **Start** $timer$
4 : **Run** P
5 : **while** ($timer \leq T$)
6 :   **if** ($timer == i.T/R$)
7 :     **interrupt** P
8 :     **stop** $timer$
9 :   **endif**
10 :   FLUSH_CACHE()
11 :   $++i$
12 :   **resume** P
13 :   **resume** $timer$
14 : **endwhile**
```

Figure 5.3: Algorithm for collecting intermediate values of cryptographic computation

Suppose cache line i holds a memory block belonging to process $P1$. Later, when another process $P2$ becomes active, and a memory block belonging to it is mapped to cache line i, a *cache miss* occurs. In order to service the miss, first the memory block (of $P1$) residing in cache line i is evicted to memory, and then, the new memory block (of $P2$) is transferred from memory to cache line i. The value belonging to $P1$ evicted from the cache can be recorded by the bus monitoring equipment. In our case, $P1$ is the cryptographic process, and $P2$ is $FLUSH\_CACHE$. Figure 5.4 shows the description of algorithm $FLUSH\_CACHE$. If cache size is X, the algorithm initializes a static array in the following manner (line 2 in

Figure 5.4):

- For a direct-mapped cache, it initializes one array of size X.
- For a Y-way set associative cache, it initializes Y arrays each of size X/Y such that the cumulative size of all the arrays is X.

Bus monitoring is begun to capture the memory traffic, and the elements in the cache are evicted by writing a dummy value ($0xFFFFFFFF$ in our case) into the initialized caches. For a set-associative cache, the dummy value is written into the same index in all the arrays so that all the elements in a cache set are evicted (lines $6-8$). This procedure evicts the intermediate values stored in the cache by the round operation prior to the interruption. These values evicted from the cache are recorded by the memory bus monitoring setup. Further analysis has to be done on the data collected in order to obtain the intermediate values required by the cryptanalysis framework. For getting the intermediate values corresponding to round i, a brute-force enumeration of the appropriate subset (data collected at time $i.T/R$) of the collected data is done. The brute-force approach is feasible due to two reasons: the number of values recorded for each round is not very large, and the SAT solver rejects wrong values by classifying them as unsatisfiable constraints (explained later).

```
FLUSH_CACHE (Cache size X, Cache associativity Y)
1 : for $i = 1$ to $Y$
2 :   initialize $A_i[X/Y]$
3 : endfor
4 : start memory bus monitoring
5 : count = 0xFFFFFFFF
6 : for $j = 1$ to $X$
7 :   for $i = 1$ to $Y$
8 :     $A_i[j] = count$
9 :   endfor
10 : endfor
11 : end memory bus monitoring
```

Figure 5.4: Algorithm for flushing intermediate values from data cache

In our work, we compiled open-source FIPS-43 [108] compliant software implementations of the DES, 3DES and AES encryption algorithms[1] on the Xtensa processor, a commercial

[1]Available at: `http://www.cr0.net:8040/code/crypto/`

32-bit embedded processor [76]. The software implementations of these algorithms were simulated using the Xtensa instruction set simulator (ISS), which models the processor, memory hierarchy, and system bus. The main memory trace was observed to extract values of program intermediate variables, which were then fed into the proposed SAT-based framework (described in the following section). We considered various cache configurations from 4KB upto 32KB. In all cases, we were able to obtain sufficient information (some internal variable values) to discover the key using the proposed framework.

5.3 SAT framework for enabling side-channel attacks

In this section, we present details of our proposed SAT-based cryptanalysis framework. We begin by giving an overview of the framework, and briefly describe its constituent steps. Next, we give a brief overview of the working of a SAT solver that serves as the underpinning of our work. Later, the method for representing a cryptographic algorithm as a Boolean formula is described. We use the popular cryptographic algorithms, DES, 3DES, and AES, for illustrating this formulation process, and to discuss the results of our experiments. However, it should be noted that our technique is general and can be applied to any cryptographic algorithm.

5.3.1 Overview of the proposed framework

We wish to represent the functionality of the cryptographic algorithm being targeted as an equivalent Boolean formula in conjunctive normal form (CNF), apply constraints corresponding to the observations, *i.e.*, plaintext, ciphertext, and internal (or intermediate) variables, produced by the secret key, to the formula, and finally, compute the secret key by using a SAT solver to solve the resulting formula. Consider an implementation of a cryptographic algorithm having a side-channel that leaks values of intermediate values (Figure 5.5). For $i \in \{1, 2, .., n\}$, plaintext P_i is mapped to ciphertext C_i for the secret key K. $\{V_i^{j+1}, V_i^{j+2}, \ldots, V_i^{j+k}\}$ (collectively denoted by $\{V_i^j\}$) represent the values of k intermediate variables leaked when the implementation transforms P_i to C_i.

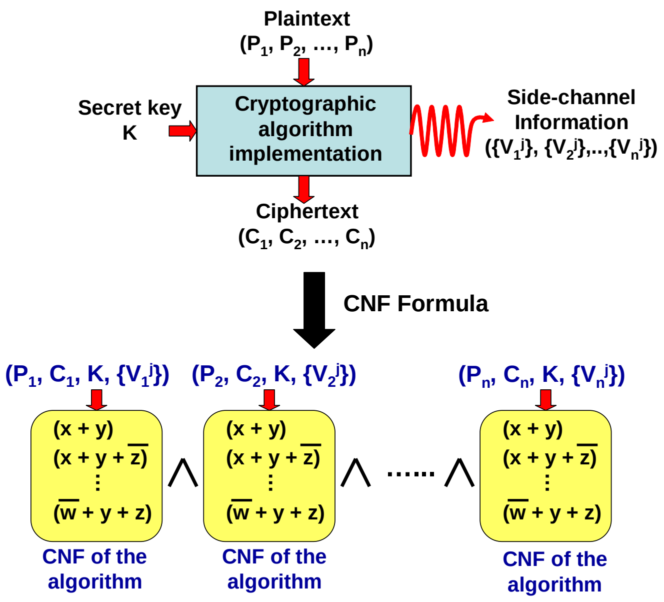


Figure 5.5: SAT formulation for side-channel cryptanalysis

Figure 5.6 illustrates the operational flow of the proposed SAT-based framework. The first step is to obtain the Boolean formulation of the algorithm (details are presented in Section 5.3.3). Let $\Psi(P, C, K, V^1, V^2, \ldots, V^m)$ represent the Boolean formula of the cryptographic algorithm where P, C, K and $\{V^1, V^2, \ldots, V^m\}$ represent literals corresponding to plaintext, ciphertext, secret key, and all the m internal variables, respectively. We illustrate the Boolean formulation of a cryptographic algorithm using Figure 5.5.

The Boolean formula is constrained based on known plaintext/ciphertext values, *i.e.*, by setting the plaintext and ciphertext literals in the formula to their observed values (*i.e.*, $(P_1, C_1), (P_2, C_2), \ldots, (P_n, C_n)$). This is done by concatenating multiple CNF formulae where each one is constrained on one known plaintext/ciphertext pair, *i.e.*, $\Psi(P_1, C_1, K, V^1, V^2, \ldots, V^m) \wedge \Psi(P_2, C_2, K, V^1, V^2, \ldots, V^m) \wedge \ldots \wedge \Psi(P_n, C_n, K, V^1, V^2, \ldots, V^m)$ $(n \geq 1)$. The next step exploits the side-channel information collected using techniques described in Section 5.2. Here, we further constrain the formula based on the intermediate variable values observed from the side-channel (*i.e.*, $\{V_1^j\}, \{V_2^j\}, \ldots, \{V_n^j\}$ where set $\{V_i^j\}$ represents the values of the intermediate variables of the algorithm observed for pair (P_i, C_i)). This is represented in the formula as $\Psi(P_1, C_1, K, \{V_1^j\}, \{V_1^j\}^c) \wedge \Psi(P_2, C_2, K, \{V_2^j\}, \{V_2^j\}^c,)$

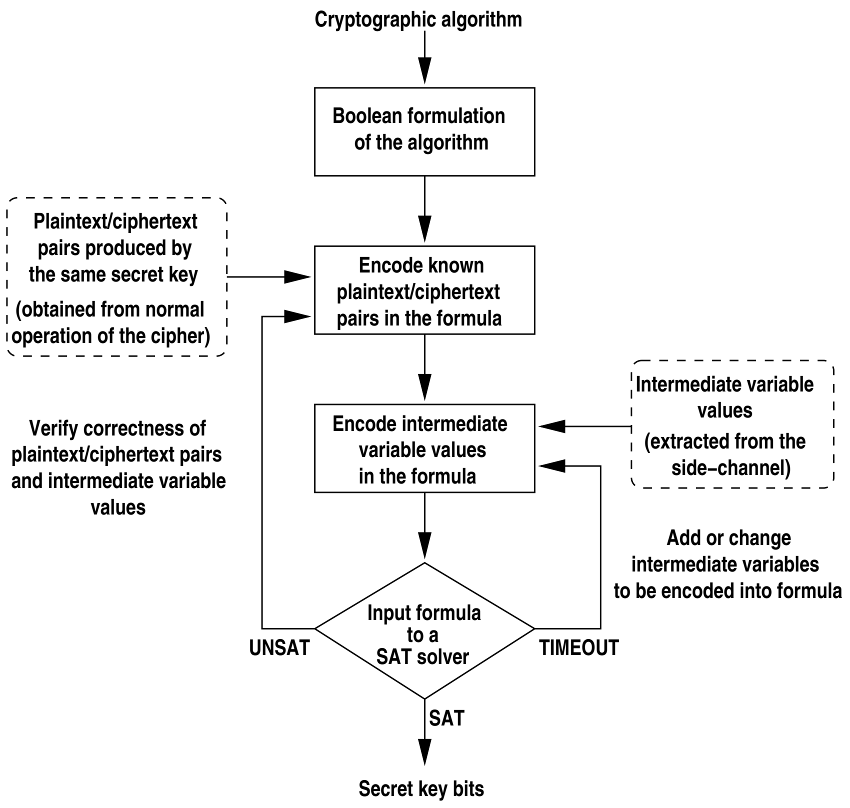

Cryptographic algorithm
Boolean formulation of the algorithm
Plaintext/ciphertext pairs produced by the same secret key (obtained from normal operation of the cipher)
Encode known plaintext/ciphertext pairs in the formula
Intermediate variable values (extracted from the side–channel)
Encode intermediate variable values in the formula
Verify correctness of plaintext/ciphertext pairs and intermediate variable values
Add or change intermediate variables to be encoded into formula
Input formula to a SAT solver
UNSAT
TIMEOUT
SAT
Secret key bits

Figure 5.6: High-level view of the proposed SAT-based framework

$\wedge \ldots \wedge\ \Psi(P_n, C_n, K, \{V_n^j\}, \{V_n^j\}^c)$ ($\{V_n^j\}^c$ represents the set of intermediate variables other than $\{V_n^j\}$ which remains unassigned). It is worth mentioning that all the constraints in the formula are with respect to the *same* secret key, K. Note that the encoding shown in the figure assumes that a block cipher is used in electronic code book mode. For other modes (chaining or feedback modes), the feedback in the algorithm is represented by constraining the values of appropriate variables (*e.g.*, initialization vector) in adjacent copies to be the same. The resulting Boolean formula is given as an input to the SAT solver. The SAT solver can terminate its search for a literal assignment with one of the following outcomes:

- **SAT**: A satisfying assignment is found. The key value, K, can be output by identifying the values assigned to literals corresponding to the key bits in the assignment.

- **UNSAT**: There is no satisfying assignment for the Boolean formula. Assuming the Boolean formulation is correct, this implies an error in the values of the plaintext-ciphertext or the intermediate variables encoded in the formula. Hence, we backtrack, re-encode the formula with correct values, and repeat the search for an assignment.

- **TIMEOUT**: The solver is unable to find either a satisfying assignment or prove no such assignment exists for the formula within a reasonable time or memory, and therefore aborts. The time and memory limits in our experiments were usually on the order of 2000 seconds and 2 GB, respectively. In this case, an iterative loop of modifying the side-channel information (either adding more variables to or replacing variables in the set of intermediate variables whose values were used as side-channel information) can be used until the solver gives a deterministic output (SAT or UNSAT) or an upper limit on the number of loop iterations is reached.

5.3.2 SAT solver algorithm

A SAT solver takes a CNF formula as its input, and either gives a Boolean assignment that satisfies the formula, or gives a proof that the input formula can never be satisfied or keeps searching until the timeout limit is reached. As mentioned earlier, a CNF formula is a logical AND of one or more clauses, where each clause is a logical OR of one or more literals. A literal is a variable or its complement. A variable which is not assigned any value is called a free variable. To satisfy a CNF formula, each clause must be individually satisfied. A clause in which all the literals evaluate to zero is known as a conflicting clause. Initially, a SAT solver algorithm does preprocessing on the CNF formula to decide if any decision can be made about the satisfiability of the formula. If a decision can be made, it outputs the result, else it enters a loop where in each iteration a free variable is picked and a value is assigned to it. The effect of assigning a value to this free variable is propagated in the formula through implications. For example, consider a clause $(y + \overline{z})$ where both y and z are free variables. Suppose, in some iteration, free variable y is assigned value zero. In order to prevent the above clause from becoming conflicting, z has to be set to zero. This propagation of values is termed *Boolean constraint propagation* (BCP). Sometimes, conflicting clauses arise and the conflict has to be resolved by undoing one or more previous variable assignments. Consider a pair of clauses $(y + \overline{z})(y + z)$. Suppose the free variable y is assigned zero. By BCP, z is set to zero to make the first clause satisfying. However, this leads to the second clause becoming conflicting since both y and z are set to zero. Resolution of this conflict involves undoing the assignments of z and y, and setting y to one. In few cases, conflicts arise that cannot be resolved. Consider a CNF formula, $(x+y)(x+\overline{y})(\overline{x}+y)(\overline{x}+\overline{y})$. Here, no variable assignment exists that can make all the clauses satisfiable. Therefore, the SAT heuristic iterates its loop until all free variables have been assigned some Boolean values (and the input formula becomes satisfiable) or it comes across an unresolvable conflict in which case it outputs an unsatisfiable result. During the iterations, the order in which free variables are chosen to be assigned values has been shown to greatly influence the complexity of finding a solution to the CNF instance. Currently, various heuristics are employed to decide the next free variable to assign a value to [98]. However, sometimes, due to the inherent complexity of the CNF formula, the SAT solver keeps iterating through its list of free variables until timeout or its memory limit is reached.

5.3.3 Boolean formulation of a cryptographic algorithm

In this paper, we limit our investigation to symmetric algorithms [3]. Encryption (and decryption) in symmetric algorithms consist of multiple iterations of a round transformation, each of which is parameterized on a different key (termed *round key*). Round key generation (also known as *key expansion*) refers to the process of generating round keys from the secret key. Thus, the operation of a symmetric algorithm is divided into two parts: round key generation and encryption/decryption process. Therefore, the Boolean formula of a cryptographic algorithm should include both key generation and encryption/decryption operations. We demonstrate this using DES, 3DES, and AES. Formulation of 3DES is a simple extension of that of DES.

5.3.3.1 CNF formulation of DES and 3DES

DES takes a 64-bit plaintext and a 64-bit secret key to produce a 64-bit ciphertext. Figure 5.7(a) shows the round key generation operation. The bits of the 64-bit secret key,

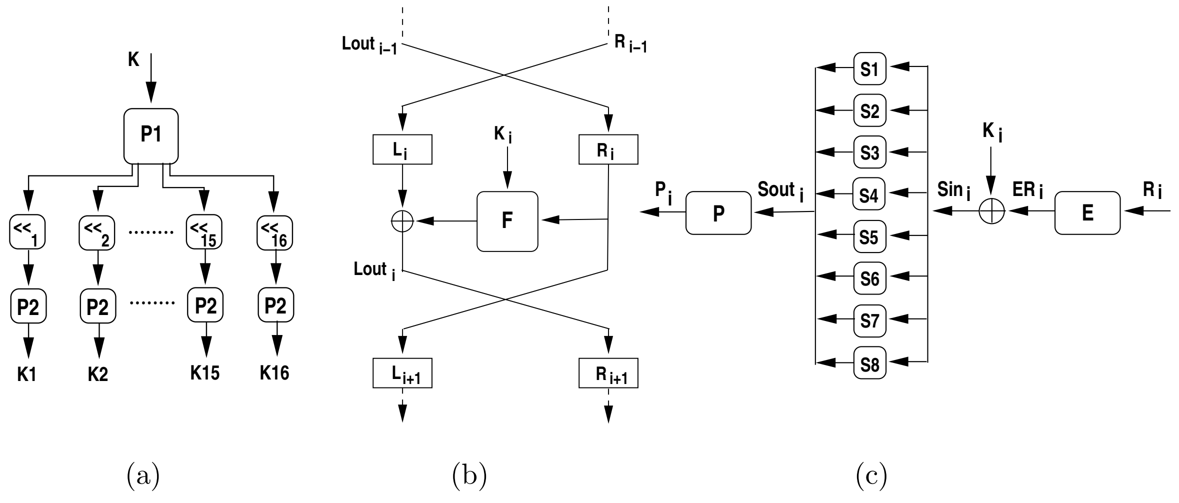


Figure 5.7: Functional view of DES encryption: (a) round key generation, (b) round transformation, and (c) F function used in the round transformation

K, are permuted using a permutation function, $P1$. $P1$ also removes the eight parity bits (located at bit positions 8, 16, 24, 32, 40, 48, 56 and 64), leaving a 56-bit output. The 56-bit value is rotated by a fixed offset, and passed through another permutation function, $P2$, to produce a round key. The rotate offset is different for each round (denoted by $<<_1$, $<<_2$, ..., $<<_{16}$ in Figure 5.7(a)). $P2$ chooses 48 bits at pre-determined bit indices from the rotated 56-bit value, and permutes them. Thus, 16 distinct 48-bit round keys are generated from the 64-bit secret key.

Encryption in DES is done by iterating the plaintext 16 times through a round transformation where a distinct round key is used for each round. The DES round transformation is a Feistel structure. Figure 5.7(b) shows the operations of a DES round transformation. Input to round i is split into two 32-bit halves: left half (L_i) and right half (R_i). R_i is transformed by the F function whose other input is the round key, K_i. The output of the F function is XORed with L_i to produce a 32-bit output, $Lout_i$. $Lout_i$ and R_i become the right (R_{i+1}) and left (L_{i+1}) halves of the next round, respectively. Figure 5.7(c) expands the F function (in round i) into its constituent operations. The 32-bit right half, R_i, is expanded into a 48-bit value, ER_i, by passing it through the expansion permutation, E. ER_i is XORed with the 48-bit round key, K_i, to produce Sin_i. The 48-bit value, Sin_i, is split into eight six-bit vectors which are input to eight distinct S-boxes ($S1$, $S2$,.., $S8$). An S-box performs a table-lookup with pre-computed values and takes a six-bit input to produce a four-bit output. The four-bit outputs of the eight S-boxes are combined to form a 32-bit value, $Sout_i$. The bits of $Sout_i$ are permuted by permutation, P, to produce P_i, the 32-bit output of the F function.

Obtaining the Boolean formulation for DES round key generation is straightforward. The bits of the 64-bit secret key are permuted and rotated to obtain the round keys. Therefore, for each round, we pre-compute the bit indices (of the secret key) which form the corresponding round key. For example, the key for round five is formed by putting the 19th bit of the secret key as the first bit of the round key, 60th bit as the second bit, 43th bit as the third bit, and so on. Thus, based on this pre-computed bit index mapping between the secret key and round keys, the appropriate secret key literals can be used in the round transformation. The Boolean formulation of DES encryption requires us to deal with three types of logic functions, *i.e.*, XOR, table lookup and permute. Given any logic function, $F(.)$, its corresponding Boolean formula can be derived using the following logical relation:

$$\begin{aligned} (Z = F(.)) &\equiv (Z \rightarrow F(.))(F(.) \rightarrow Z) && (5.1)\\ &\equiv (\overline{Z} + F(.))(\overline{F(.)} + Z) && (5.2) \end{aligned}$$

Assuming $F(.)$ and $\overline{F(.)}$ are in the product-of-sum form, the above expression can be expanded into a Boolean formula using the logic relation, $(a + bc) = (a + b)(a + c)$. The Boolean formulas of the logic functions in a DES round can be obtained as follows:

- **XOR**: The Boolean formula representing the XOR of two vectors can be obtained by the conjunction of the Boolean formula representing the XOR of individual bits. Let $z_i = x_i \oplus y_i$, where x_i and y_i are the ith input bits, and z_i the ith output bit. The Boolean formula, Φ_i, representing this operation can be derived as follows:

$$\begin{aligned} \Phi_i &= (\overline{z_i} + (x_i \oplus y_i))(z_i + \overline{(x_i \oplus y_i)})\\ &= (\overline{z_i} + (\overline{x_i} + \overline{y_i})(x_i + y_i))(z_i + (\overline{x_i} + y_i)(x_i + \overline{y_i}))\\ &= (\overline{z_i} + \overline{x_i} + \overline{y_i})(\overline{z_i} + x_i + y_i)(z_i + \overline{x_i} + y_i)(z_i + x_i + \overline{y_i}) \end{aligned}$$

- **Permutation**: The permutation functions, E and P (Figure 5.7(c)), rearrange their input bits at the output (E also duplicates some of the input bits). If the jth input bit x_j is assigned to the ith output bit z_i, then the corresponding Boolean formula is given by $(\overline{z_i} + x_j)(z_i + \overline{x_j})$. We get the Boolean formula for the permutation function by the conjunction of formulae for all the output bits.

- **S-box**: The S-boxes are the only non-linear functions in the DES algorithm. An S-box maps a 6-bit input to a 4-bit output. This enables us to enumerate the behavior of an S-box using a truth table, and use a logic minimizer tool to obtain logic expressions for each of the four outputs in terms of the six inputs. Using the logic expressions, Boolean formulae can be derived for each of the four output bits (using the logic relation described in Equation (5.1)). The formula for a single output bit comprises 34 clauses. Conjunction of the Boolean formulae of the four output bits gives the formula for the S-box with 136 clauses.

3DES is computed by using three iterations of the DES algorithm using different keys for each iteration. It consists of DES encryption with key k_1, followed by DES decryption with key k_2, and finally DES encryption with key k_3. Operations in DES decryption are similar to DES encryption except that the order of the round keys is reversed, *i.e.*, we use the 16th round key first, and go down to the first one. Thus, 3DES effectively uses a 192-bit key for encryption. The Boolean formula for 3DES is derived by conjoining Boolean formulae for DES encryption, DES decryption, and DES encryption with different key literals.

5.3.3.2 CNF formulation of AES

AES encrypts data in 128-bit blocks, by iterating the block multiple times through a round transformation. The number of rounds is 10, 12 and 14 for 128-bit, 192-bit and 256-bit keys, respectively. In the key expansion step, an N-bit key is expanded to a $128 * (M + 1)$-bit key, where M is the number of rounds (which is dependent on N, as mentioned above). For example, a 128-bit key is expanded into $128 * (10 + 1) = 1408$ bits. During key expansion, N bits of the secret key are copied into the first N bits of the expanded key, and every subsequent byte of the expanded key is obtained by performing rotation, substitution and XOR operation on the four previous bytes of the expanded key. Rotation and XOR operations can be converted to clauses, as explained in the previous section. Byte substitution is implemented as a 256-entry table having an 8-bit input and 8-bit output. The table is represented as a 256-entry truth table with eight inputs and eight outputs. A logic minimizer takes this table as input and produces minimal logical expressions for each of the eight outputs in terms of variables representing the eight input bits. The logical expressions are then converted to CNF formulae. This procedure is similar to that used for converting an S-box into a CNF formula.

For encryption (or decryption), the 128-bit input block is represented as a 4×4 matrix of bytes, called *state*. A round transformation comprises four operations:

- *ByteSub*: Substitute bytes in the *state* using table lookup.
- *ShiftRow*: Shift rows of the *state* by constant amounts.
- *Mixcolumn*: Columns of the *state*, which are represented as polynomials in GF(2^8), are multiplied by $3x^3 + x^2 + x + 3$ modulo $x^4 + 1$.
- *AddRoundKey*: XOR the bytes in the *state* with the round key bytes.

On a 32-bit processor, the four operations in the round transformation can be combined into four table lookups and four XORs as shown below

$$\begin{bmatrix} e_{1,j} \\ e_{2,j} \\ e_{3,j} \\ e_{4,j} \end{bmatrix} = T_1[a_{1,j}] \oplus T_2[a_{2,j}] \oplus T_3[a_{3,j}] \oplus T_4[a_{4,j}] \oplus \begin{bmatrix} k_{1,j} \\ k_{2,j} \\ k_{3,j} \\ k_{4,j} \end{bmatrix}$$

where $(e_{1,j}, e_{2,j}, e_{3,j}, e_{4,j})$ is the jth column of the output, $(a_{1,j}, a_{2,j}, a_{3,j}, a_{4,j})$ is the jth column of the state, $(k_{1,j}, k_{2,j}, k_{3,j}, k_{4,j})$ is the jth column of the round key, and $(T_1[], T_2[], T_3[], T_4[])$ are four pre-determined 256-entry tables, each of which takes a 8-bit input and outputs a 32-bit value. Figure 5.8 illustrates the round transformation realized using table lookups where S_{ij} represents the entry in the ith row and jth column of a 4×4 state matrix. Indices of S inputs to the tables in Figure 5.8 are explained by the $ShiftRow$ operation in the AES round transformation. The two operations in the round transformation are XOR and table lookup. These operations are converted into CNF clauses as explained previously.

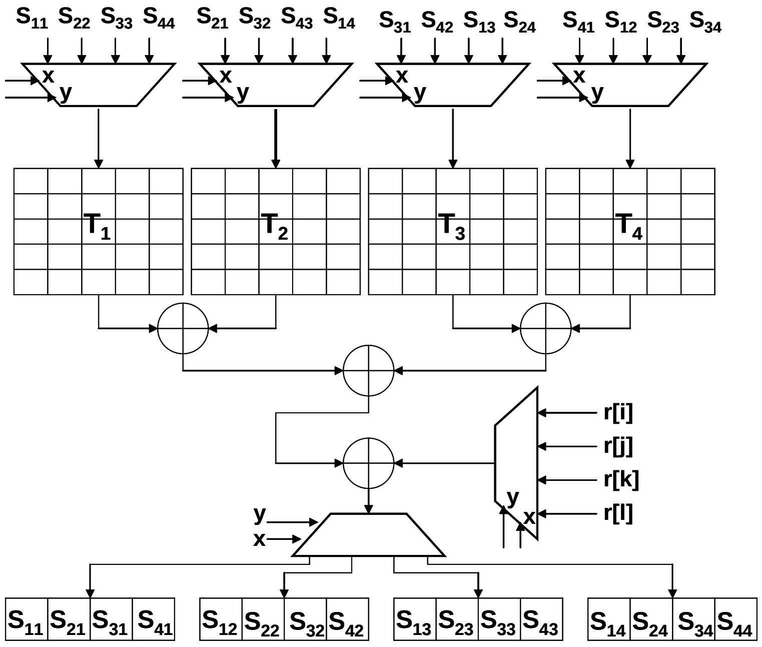


Figure 5.8: AES round transformation

5.3.3.3 Summary

Table 5.1 summarizes the number of variables and clauses present in the Boolean formulae for three popular cryptographic algorithms: DES, 3DES and AES (128-bit key). We see the number of clauses needed for AES is an order of magnitude greater than that required for DES, and five times more than the number of clauses needed for 3DES. This is primarily due to the higher complexity of AES round operations compared to DES. Each AES round requires about 50000 clauses as compared to 35000 clauses for the entire 16-round DES. Moreover, AES key expansion is much more complex than DES key expansion. In DES, the round keys are generated by permuting the 56-bit (64 bits minus the 8 parity bits) secret key bits by pre-determined amounts, and taking a 48-bit subset of them. Thus, the clauses needed for key generation are quite simple. In contrast, in AES the round keys are generated by performing algebraic operations on the 128-bit secret key. AES key expansion requires around 40000 clauses compared to almost nothing for the same step in DES.

Table 5.1: Results of the Boolean formulation

Algorithm	Variables	Clauses
DES	6904	35232
3DES	20328	104928
AES	10240	542432

5.4 Experimental results

In this section, we present the details of our experimental setup followed by results of our side-channel attack method for DES, 3DES and AES. We conclude the section with some observations based on the experimental results. We studied the efficacy of all the intermediate variables in DES regarding their ability to enable our technique to compute the secret key. Our exhaustive studies show that knowledge of certain sets of intermediate variables allows our technique to successfully determine the secret key. We refer to these sets of variables as *enabling sets*. We present some rules that invariably describe how the enabling sets for the DES algorithm can be formed. Thus, all the enabling sets can be enumerated by iterating through these rules. These rules also hold for 3DES where they are applied separately to its three DES segments. We also present results of applying our method to cryptanalyze AES.

5.4.1 Experimental setup

We performed all our experiments on a PC with a 1.6 GHz Pentium processor and a 512MB RAM running Debian Linux OS. Figure 5.9 shows the experimental setup which is divided into two parts that are described below:

- *Satisfiability-based cryptanalysis framework*: The input to this flow is the software implementation of the target cryptographic algorithm. We implemented our intermediate value leakage flow using the Xtensa ISS framework [76]. We instrumented

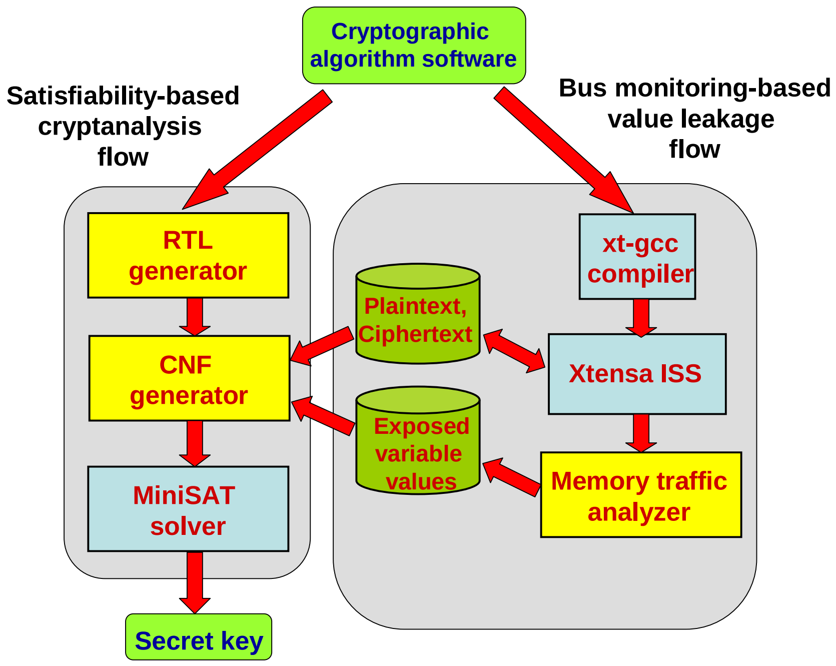


Figure 5.9: Experimental setup

the cryptographic software code such that its execution is interrupted at the end of every round, and the program which flushes the cache (described in Section 5.2.2) is executed. The instrumented code was compiled using the xt-gcc compiler, and then executed on the Xtensa ISS using known plaintext(s). At the end of the simulation, we have the ciphertext(s) corresponding to the input plaintext(s), and memory traffic containing values evicted at the end of each round of the cryptographic algorithm. This traffic is analyzed in order to extract the required intermediate values, *i.e.*, the enabling sets. As already mentioned, this process is implemented through brute-force enumeration with the help of the SAT solver which gives an UNSAT when wrong values are input for variables making up an enabling set (for a given plaintext-ciphertext pair).

- *Bus monitoring-based value leakage flow*: The first step of the flow is to obtain the structural RTL description of the cryptographic algorithm using an RTL generator. Our scripts automatically convert the structural description of the cryptographic algorithm into its equivalent CNF formula. Next, constraints in the form of plaintext, ciphertext, and the corresponding intermediate values (an enabling set) is introduced into the CNF formula. This constrained CNF formula is given as input to the SAT solver. We used the MiniSAT SAT solver from Chalmers University [99], since it has been benchmarked to be one of the best performing publicly available SAT solvers (`http://www.satlive.org`). However, similar results were also obtained using other state-of-the-art solvers such as zChaff [98].

5.4.2 Cryptanalysis of DES

We present the rules characterizing the enabling sets of DES with the help of Figure 5.10. We obtained the rules in the following manner. For a given plaintext/ciphertext pair, we ran simulations where values of different combinations of intermediate variables in the DES algorithm were input to the SAT solver in sequence and checked whether the solver could compute the value of the secret key for each combination. Next, the list of all intermediate variable combinations which enabled the SAT solver to compute the secret key was made, and a minimum set of rules which encapsulated all the successful combinations was derived. Subsequently, these rules were verified by running the SAT solver over 1000 randomly generated plaintext/ciphertext pairs. Figure 5.10 shows four consecutive rounds in the DES algorithm (indices i, $i+1$, $i+2$, $i+3$, $i+4$ indicate rounds). The variables in an enabling set are encapsulated within this four-round DES structure wherever this structure might occur within the 16 rounds of the DES algorithm, *i.e.*, $i \in \{1, 2, .., 13\}$. The rules are enumerated below:

1. **Forward L-L path**: A lower-round L value separated from a higher-round one by a single XOR operation, and the R value adjacent to the lower-round L form an enabling set. In Figure 5.10, $\{L_i, L_{i+2}, R_i\}$ forms an enabling set according to this rule.

2. **Forward R-L path**: A lower-round R value separated from a higher-round L value by a single XOR operation, and the L value adjacent to the lower-round R value form an enabling set. By this rule, $\{R_i, L_{i+3}, L_i\}$ is an enabling set.

3. **Reverse R-L path**: A higher-round R value separated from a lower-round L value by a single XOR operation, and the L value adjacent to the R value form an enabling set. $\{R_{i+4}, L_{i+3}, L_{i+4}\}$ becomes an enabling set according to this rule.

4. **Reverse L-L path**: A higher-round L value separated from a lower-round L value by a single XOR operation, and the R value adjacent to the higher-round L value form an enabling set. Based on this rule, $\{L_{i+4}, L_{i+2}, R_{i+4}\}$ becomes an enabling set.

5. **Two-XOR path**: A lower-round L value separated from a higher-round L value by two XOR operations, and the R values adjacent to these L values form an enabling set. Therefore, $\{L_i, R_i, L_{i+4}, R_{i+4}\}$ becomes an enabling set.

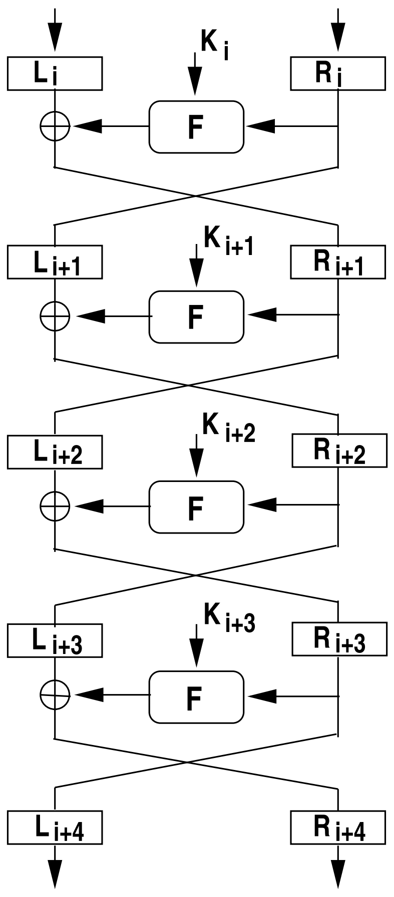


Figure 5.10: DES structure illustrating invariant rules

A minimum of three intermediate variables (rules 1-4) and a maximum of four (rule 5) are needed to compute the DES secret key. Special cases arise when index i is 1 or 13.

When i is 1, L_1 and R_1 (analogs of L_i and R_i in Figure 5.10) are the left and right half of the plaintext which is already provided. Thus, according to rules 1 and 2, we need the knowledge of *only one* intermediate variable, either L_3 or L_4, to compute the secret key. To apply rule 5, we require two intermediate variables, L_5 and R_5, to extract the secret key. Similarly, when i is 13, L_{17} and R_{17} (analogs of R_{i+4} and L_{i+4} in Figure 5.10 since there is no crossover of L and R values in the last round) are the two halves of the ciphertext which is known. According to rules 3 and 4, knowledge of either L_{16} or L_{15} is required to extract the secret key. To apply rule 5, values of both L_{13} and R_{13} are required. Thus, we see that in special cases, *i.e.*, $i \in \{1, 13\}$, the minimum number of intermediate variables required to compute the DES key reduces to one.

For time efficiency, multiple plaintexts and their corresponding ciphertexts produced by the same secret key can be encoded into the Boolean formula. In some cases, this extra information increases the power of the SAT solving process. The time taken to compute the secret key as a function of the number of plaintext/ciphertext pair values encoded into the Boolean formula for rule 3 (L_{16}) and rule 5 ($\{L_5, L_6\}$) enabling sets is shown in Figures 5.11 and 5.12, respectively. Along with the plaintext/ciphertext pair values, the corresponding values of variables $\{L_{16}\}$ and $\{L_5, L_6\}$ are also encoded. Figure 5.11 shows the time taken to compute the secret key using 2, 4, 8, 16 and 32 plaintext/ciphertext pairs when set $\{L_{16}\}$ is encoded. The SAT solver times out when values of only one plaintext/ciphertext pair is provided. Here, we can see that our technique can compute the secret key within two seconds when provided with two pairs of values, and the time taken increases as more pairs are provided. This observation is true across enabling sets found by rules 1, 2, 3 and 4. Similarly, Figure 5.12 shows the average trend of time taken to compute the secret key using 1, 2, 4, 8, 16 and 32 plaintext/ciphertext pairs when set $\{L_5, L_6\}$ is encoded. Here, the time taken decreases as the number of pairs encoded increases from one to four, and then increases. In general, this observation holds for enabling sets obtained by rule 5. Increasing the number of plaintext/ciphertext pairs progressively constraints the size of the search space which the SAT solver needs to go though. However, the size of the

CNF formula increases linearly with the number of plaintext/ciphertext pairs input. Thus, beyond a certain number of plaintext/ciphertext pairs, the increase in time required to read the formula and process it exceeds the reduction in search time. This could possibly explain the above behavior.

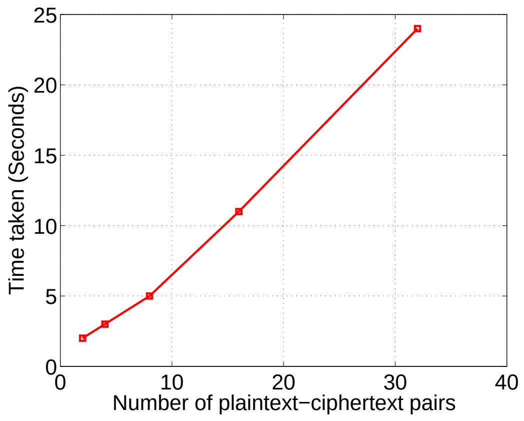


Figure 5.11: Time to compute the secret key with the value of variable L_{16}

There are various ways in which new enabling sets can be formed by replacing variables in an enabling set by equivalent ones. Consider the enabling set produced by rule 1, $\{L_i, L_{i+2}, R_i\}$. From Figure 5.10, we see that variable L_{i+2} is the same as variable R_{i+1}. Thus, $\{L_i, R_{i+1}, R_i\}$ is also an enabling set. Similarly, a rule 5 enabling set $\{L_i, R_i, L_{i+4}, R_{i+4}\}$ is equivalent to $\{L_i, L_{i+1}, L_{i+4}, L_{i+5}\}$. With respect to enabling the SAT solver to solve a Boolean formula, providing the value of variable R_{i+1} in Figure 5.10 is equivalent to providing the values of variables P_i, $Sout_i$ or Sin_i in the F function (Figure 5.7(b)) of round i. This can be easily explained. For example, consider the enabling set, $\{L_i, R_{i+1}, R_i\}$ (which is equivalent to the rule 1 enabling set, $\{L_i, L_{i+2}, R_i\}$). Values of L_i and R_{i+1} enable the SAT solver to find the output of the F function by a simple backward implication through the round XOR operation. This derived value can be further back-propagated through the F function until the output of the XOR operation (inside the F function) is computed. One input to this XOR can be easily derived from the forward propagation of R_i, and the other input is the round key, K_i. Thus, a simple implication will reveal the bits of the round key. By providing values of P_i, Sin_i or $Sout_i$ instead of R_{i+1}, we are directly providing the values of the output of the F function, and obviating the need for the first back implication through the round XOR. Therefore, variable sets $\{L_i, P_i, R_i\}$, $\{L_i, Sout_i, R_i\}$ and $\{L_i, Sin_i, R_i\}$ become enabling. In this manner, the collection of enabling sets can be increased.

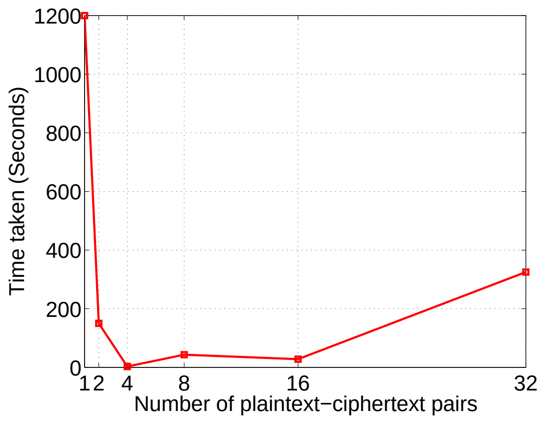


Figure 5.12: Time taken to compute the secret key with the values of variables L_5 and L_6

5.4.3 Cryptanalysis of 3DES

3DES is made up of 48 DES rounds, where rounds 1 to 16 implement DES encryption with key k_1, rounds 17 to 32 implement DES decryption with key k_2, and rounds 33 to 48 implement DES encryption with key k_3. Effectively, 3DES has a 192-bit secret key comprising three 64-bit keys, k_1, k_2 and k_3. The enabling sets for DES, which are described above, can be extended to 3DES. We get an enabling set for 3DES by combining the enabling sets of each of its three constituent DES segments. For the sake of clarity, we separate the results for 3DES into two classes below - 'rule 5 3DES enabling sets,' and 'non rule 5 3DES enabling sets':

- **Rule 5 3DES enabling sets**: These are obtained by combining rule 5 enabling sets of the three DES segments, *i.e.*, analogs of $\{L_i, R_i, L_{i+4}, R_{i+4}\}$ (Figure 5.10) in the three segments. For example, we can form a rule 5 3DES enabling set by putting together the following three DES enabling sets, $\{L_9, R_9, L_{13}, R_{13}\}$ (for the first DES segment), $\{L_{25}, R_{25}, L_{29}, R_{29}\}$ (for the second DES segment), and $\{L_{41}, R_{41}, L_{45}, R_{45}\}$

(for the third DES segment). An interesting example is the one obtained by combining $\{L_5, R_5\}$ (for the first segment), $\{L_{21}, R_{21}\}$ (for the second segment) and $\{L_{45}, R_{45}\}$ (for the third segment). Here, since L_1 and R_1 constitute the plaintext, it is sufficient to provide values of only L_5 and R_5 for finding the 64-bit secret key of the first segment. Similarly, since L_{49} and R_{49} constitute the ciphertext, values of L_{45} and R_{45} are enough to compute the 64-bit key of the third segment. After finding the key of the first segment, the SAT solver can compute the outputs of this segment, L_{17} and R_{17} (which are also the inputs to the second segment), through forward implications. Thus, it is sufficient to provide only the values of L_{21} and R_{21} to compute the 64-bit key of the second segment (since set $\{L_{17}, R_{17}, L_{21}, R_{21}\}$ forms a rule 5 enabling set). Likewise, providing the values of set $\{L_{29}, R_{29}\}$ instead of $\{L_{21}, R_{21}\}$ gives the same result. In this case, after the SAT solver computes the key of the third segment, it performs backward implications from the output of the third segment (which is the ciphertext) to find the inputs of this segment, L_{33} and R_{33} (which are also the outputs of the second segment). As in the previous case, $\{L_{29}, R_{29}, L_{33}, R_{33}\}$ forms a rule 5 enabling set. For rule 5 enabling sets, our technique could derive the 192-bit 3DES key in 750 seconds on average (using four plaintext/ciphertext pairs).

- **Non rule 5 3DES enabling sets**: These are formed by combining enabling sets obtained by rules 1, 2, 3 and 4 for each of the three DES segments of 3DES. An interesting example of this enabling set is obtained by combining rule 1 enabling sets for the first and second segments, and rule 3 enabling set for the third segment: $\{L_3\}$ (L_1 and R_1 are plaintext), $\{L_{17}, L_{19}, R_{17}\}$, and $\{L_{48}\}$ (L_{49} and R_{49} are ciphertext). Likewise, 3DES enabling sets can be obtained by various combinations of enabling sets for the three DES segments. For non rule 5 enabling sets, our technique could derive the 192-bit 3DES key in 1165 seconds on average (using four plaintext/ciphertext pairs).

5.4.4 Cryptanalysis of AES

Cryptanalysis of AES is much more complex than DES or 3DES. Figure 5.8 shows the functionality of an AES round transformation. Each round takes a 128-bit value as its input and processes it 32 bits at a time to produce a 128-bit output which is input to the next round. Thus, four iterations of the functionality shown in Figure 5.8 are required to get through all the 128 bits of the input. In our experiments with 128-bit key AES (which has 10 rounds), knowledge of the 128-bit input and 128-bit output of any round was the minimum required to obtain the key bits. The first and tenth rounds are special cases whose input and output are the plaintext and ciphertext, respectively. In these cases, either the 128-bit output of the first round, or the 128-bit input to the tenth round is required to obtain the 128-bit key. In all these cases, a single plaintext-ciphertext pair was sufficient, and the SAT solver output the secret key within five seconds.

5.4.5 Observations

Prior to our work, it was observed that hardness of the DES CNF increases abruptly beyond three rounds for SAT solvers, and becomes unsolvable for four rounds and higher (irrespective of the number of plaintext and ciphertext values encoded into the formula) [57, 109]. However, our experimental results demonstrate that a four-round DES can be solved by a SAT solver when only two plaintext and the corresponding ciphertext values are encoded into the CNF. The SAT solver aborts for five- (and higher) round DES. An interesting question to resolve is whether five-round DES can be solved by making suitable enhancements to the SAT solving algorithms or it represents a provably insurmountable barrier? Researchers in satisfiability have used a metric called clause-to-variable ratio, as a way of quantifying the hardness of solving 3-CNF instances [110] (3-CNF comprise only three-literal clauses). Based on this metric, they have empirically shown the existence of a *threshold phenomenon* – a ratio of about 4.3 separates under-constrained 3-CNF (which can be proven to be easily satisfiable) from over-constrained 3-CNF (which can be easily proven to be unsatisfiable). The 3-CNF instances lying in the neighborhood of the threshold value, *i.e.*, 4.3, are extremely hard to solve. Table 5.2 shows the number of variables, clauses and the clause to variable ratio for one- to five-round DES coded as 3-CNF. We see that the clause-to-variable ratio hovers around 1.37 in all five cases. At least based on the threshold metric, the five-round DES does not show an abrupt increase in hardness of solving. On the other hand, the clause-to-variable ratio of 3-CNF might not be a suitable metric for representing the hardness of DES type of formulations.

Table 5.2: 3-CNF DES formula

Round	Variables	Clauses	Ratio
1	4766	6560	1.376
2	8788	12096	1.376
3	12802	17632	1.377
4	16816	23168	1.377
5	20830	28704	1.378

5.5 Chapter summary

In this work, we have presented a novel framework for performing side-channel attacks on cryptographic software. We have argued and demonstrated the dangers of software side-channels in compromising secret keys. Also, we have developed an automated SAT-based framework for exploiting the vulnerabilities of the software side-channel. However, the SAT solver has some limitations, *e.g.*, it cannot break the key in cases where the exposed variables are separated by more than four DES rounds. It would be interesting to resolve the question of whether five-round DES presents an insurmountable barrier for SAT solvers. Based on our experience, we foresee scope for further research along two directions: a thorough analysis of the nature of software side-channels and their prevention, and improving SAT solving techniques with the aim of enhancing their cryptanalysis capabilities.

Chapter 6

Verifying Integrity of Data Stored in Untrusted Memory with Fewer Queries

Memory integrity verification, which enables detection of tampering of data stored in untrusted memory, is an essential requirement of secure processors that provide private and tamper-proof computation. Limited on-chip storage in a secure processor makes it necessary for it to store data (including program code) in an untrusted external memory where it is easily susceptible to adversarial tampering. Thus, to ensure validity of computation, it is extremely important to have techniques that can verify integrity of data stored in untrusted memory. Existing memory integrity verification techniques, like Merkle trees, impose a very high communication overhead, *i.e.*, a large number of queries from the processor to memory, in order to perform data integrity verification. Given that memory latency is very high compared to the execution speed of the processor, this imposes a significant running time penalty for applications executing on the processor. Our proposed technique, which uses properties of Toeplitz matrices, performs integrity verification with low communication overhead while incurring a modest increase in on-chip storage requirement. In this chapter, we present details of the proposed technique and provide corresponding proofs of security and correctness.

6.1 Introduction

Memory integrity verification refers to the process of detecting any unauthorized tampering of data stored in external memory. This verification is an important component of secure and trusted processing architectures. Most of the proposed architectures for secure and trusted computing comprise a tamper-proof processor enclosing on-chip memory for storing cryptographic keys, and highly sensitive code and data (*e.g.*, that of the trusted kernel), and special-purpose cryptographic hardware for carrying out efficient cryptographic computations [111–113]. However, code and data of trusted applications that cannot be stored in the limited trusted on-chip memory are sent to the external memory which is outside the secure perimeter imposed by the tamper-resistant processor casing. Thus, it is imperative to have a memory integrity verification scheme that can detect any unauthorized tampering of data between the time it is written into and read back from the external memory by the secure processor.

The external memory, containing program code and data, is assumed to be in total control of an adversary who can alter values in any memory location in any manner. Memory integrity verification is performed by a program running on the secure processor, and it ensures the integrity of code and data requested by any program in the course of its normal execution. Based on the values stored in the external memory, the integrity verification program computes some information, and stores it in a dedicated private memory on the processor. Later, the verification program uses this private information in order to detect any tampering of data at any location of the external memory. Whenever changes are made to data in the external memory, this private information has to be updated accordingly. Also, in order to verify the integrity of the data (or code) requested by an executing program, a verification program makes queries to the external memory for additional data. These additional data are needed only for verification purposes. The size of the on-chip private memory used by the verification program is referred to as its *space complexity*, and the number of additional queries made by the verification program is referred to as its *communication complexity.* The two main requirements of any effective memory integrity verification program are as follows:

- It should detect any form of adversarial corruption of values stored in an external memory.
- Since integrity verification is done on code and data requested by an executing program, it is important that verification should not impose a significant execution time overhead.

The execution time of a program on a processor comprises two components: processor execution time (time spent by the processor executing arithmetic and logic computations), and memory access time (time spent by the program waiting for loads and stores to be executed). However, rapid advances in processor architectures, combined with the inability of developments in memory technology to keep up with it, have resulted in program execution time being dominated by its memory access time (studies have shown that some programs spend more than half their execution time waiting for their memory accesses to complete [114]). It has been estimated that memory performance is getting worse relative to processor performance by 40% each year, and at present, high-end processors, like Pentium 4, have a memory latency of about 400 cycles in spite of employing various memory latency hiding techniques, like prefetching, speculation, *etc.* [115]. Therefore, in order that memory integrity checking does not impose a high execution time overhead, it is necessary to keep its communication complexity as small as possible. However, this is not true for the most widely used memory integrity checking technique, Merkle trees [116]. A Merkle tree is a tree data structure in which the leaf nodes hold blocks of memory, and every internal node holds the hash of the concatenation of contents of all its children. After a Merkle tree is built for a memory, the hash in the root node (called *root hash*) is stored securely on the processor while all the internal nodes are stored in the memory. Thus, the Merkle tree space complexity is only p bits (assuming the hash length is p bits) and is independent of the total number of blocks in memory. A Merkle tree constructed for data integrity verification of an untrusted external memory is illustrated in Figure 6.1.

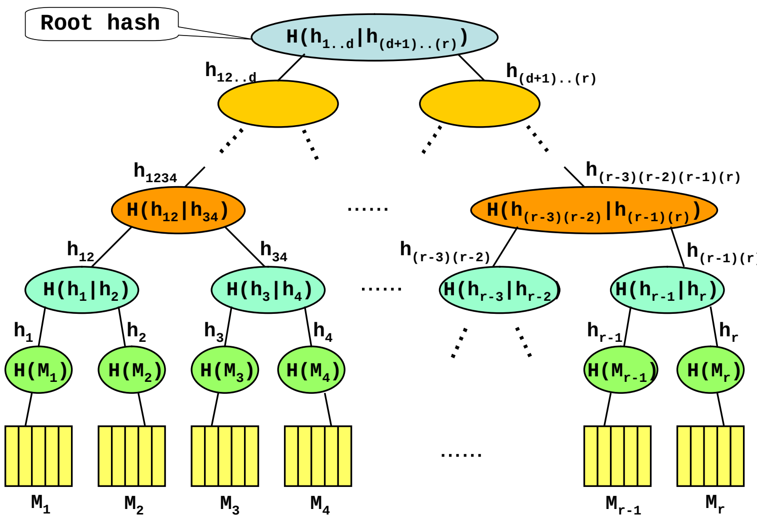


Figure 6.1: Integrity verification using Merkle tree

Suppose that an external memory of size n blocks is divided into r sets having m blocks each, *i.e.*, $r = (\frac{n}{m})$. Let the r sets be denoted as $\{M_1, M_2, \ldots, M_r\}$. They form the nodes of the Merkle tree (Figure 6.1). Suppose that a block is read in from leaf node M_3. Initially, contents of the leaf nodes, M_3 and M_4, are read from the memory, and their hashes, $h_3 = H(M_3)$ and $h_4 = H(M_4)$, are computed, respectively. Hash values h_3 and h_4 are concatenated, and the resulting value is hashed to generate $h_{34} = H(h_3|h_4)$ ($|$ denotes concatenation). The hash value in the neighboring node, h_{12}, is read from memory, concatenated with hash value h_{34}, and hashed to generate $h_{1234} = H(h_{12}|h_{34})$. This process is continued until the root hash value is computed. Finally, the computed root hash value is compared with the pre-computed root hash value stored in a secure memory on the processor. Thus, in order to perform integrity checking of a memory block, we need to fetch $2*m$ blocks (corresponding to the two leaf nodes) followed by $log_2\, r = log_2(\frac{n}{m})$ blocks containing the internal nodes along the path from the leaf to the root node of the Merkle tree. Thus, the communication complexity of Merkle tree integrity checking is $(2*m+log_2\, r)$ (assuming a memory block is fetched per memory access). For typical values of memory size n, the communication complexity of integrity checking can be quite high. Moreover, optimizations aimed at reducing the height of the Merkle tree by increasing the leaf node size have limited performance benefit, as will be demonstrated next.

Consider a memory M of size 1GB having 64B-sized blocks. Thus, the number of blocks in the memory, $n = (2^{30}/2^6) = 2^{24}$. Figure 6.2 plots the communication complexity of Merkle tree integrity verification of memory M as a function of the leaf node size m. The communication complexity is plotted on the y-axis, and the leaf node size m is varied as 1, 2, 4, 8, 16, 32 and 64 memory blocks on the x-axis. For each value of leaf node size, Figure 6.2 shows the number of memory queries needed to fetch the leaf node (curve labeled 'Leaf node queries'), the internal nodes holding the hash values (curve labeled 'Internal node queries'), and the sum of the two values (curve labeled 'Total queries'). We can see that increasing the leaf node size reduces the height of the Merkle tree and slows the growth in communication complexity upto around 8 blocks per node. However, beyond that point the communication complexity is predominantly influenced by the number of memory accesses required to read in the blocks comprising a leaf node, and grows linearly with the leaf node size. Thus, given the high memory latency of modern processors (around 400 cycles), the high communication complexity of a Merkle tree-based scheme translates into a significant execution time overhead for memory integrity verification. Note that another way to decrease the height of the Merkle tree is to increase the number of children per node. However, this does not change much in terms of memory accesses, since more memory accesses have to be made at the internal nodes in order to fetch the children.

We can perform space-communication complexity trade-offs, in order to reduce the communication complexity. The Merkle tree has different levels, starting from level zero, which has the root hash node, and going up until the last level which contains hash values of all the leaf nodes (nodes $\{h_1, h_2, \cdots, h_r\}$ in Figure 6.1). In general, the ith level of a Merkle tree has 2^i nodes (assuming a binary tree). When the root hash node alone is stored on the processor, $c = log_2 r$ internal hash nodes have to be fetched from memory for integrity verification. Assuming one memory block is fetched on every memory access, we would

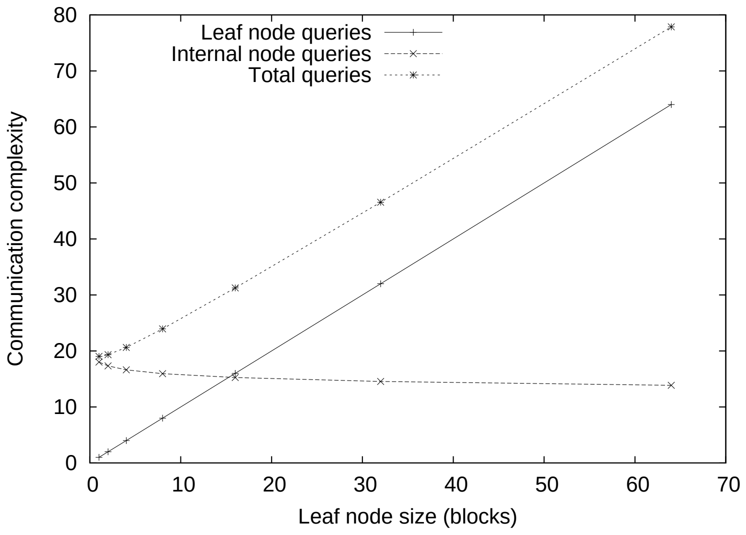


Figure 6.2: Merkle tree communication complexity

need c memory accesses, and the communication complexity is equal to $(2 * m + c)$. If all the 2^i nodes at the ith level of the Merkle tree are stored on the processor, then only $(c - i)$ internal hash nodes need to be fetched from memory, and the communication complexity reduces to $(2 * m + (c - i))$. Thus, an increase in space complexity can be traded for a decrease in communication complexity. In the extreme case, no internal nodes are required to be fetched from the memory when the hash values of all the leaf nodes are stored on the processor, *i.e.*, nodes $\{h_1, h_2, \cdots, h_r\}$ in Figure 6.1. In this case, only the blocks in a leaf node have to be fetched for verifying the integrity, and the communication complexity is equal to m. This drastic reduction in communication complexity comes at the cost of a maximum increase in space complexity. We refer to this case as the *store hash* scheme. However, even the store hash scheme has an unnecessary overhead with respect to communication complexity: even though a read or write is performed to one memory block in a leaf node, the remaining $(m - 1)$ blocks have to be read in order to perform integrity verification. The communication complexity of the store hash scheme can be further reduced by decreasing the leaf node size m at the cost of an increase in space complexity (which is given by $r * p = \frac{n}{m} * p$ where p is the length of the hash value). However, for large values of n, making the value of m very small can lead to a prohibitively expensive space complexity. We propose a scheme which enables us to reduce the communication complexity by reducing the leaf node size m while incurring much less space overhead than in the case of the store hash scheme. The proposed scheme achieves this benefit at the cost of additional computations with respect to computing and updating the information stored in its private memory. However, the execution time overhead due to this additional computations can be reduced by introducing custom functional units aimed at optimizing these computations [76].

6.1.1 Contribution of the work

In this chapter, we propose a new method for verifying the integrity of an untrusted memory with low communication complexity while incurring a modest space overhead. The main idea underlying the proposed scheme is as follows: given a group of s members, $\{A_1, A_2, \cdots, A_s\}$, we generate a value X which can be used to check the integrity of any one of the s members. We refer to X as the state value of the group. Value X can be trivially generated by obtaining the cryptographic hash values of all the s members of the group, and concatenating them (in fact, this is equivalent to the *store hash* scheme). However, our scheme generates a state value X which is lot more space-efficient than the above-mentioned trivial case. An example application of the proposed scheme would be to improve the performance of data integrity checking of secure databases. In databases, hash values are computed over large partitions of the database. Thus, in order to check the integrity of data accessed by database queries, we need to read in entire contents of the partition to which the queries are made (even though data other than that accessed by the queries are not immediately needed). This high communication complexity results in performance degradation. However, by using the proposed scheme, the database can be partitioned into groups of small sets of data, and state values of all the groups computed in a space-efficient manner. Thus, only the data in the set to which the query was made need to be read for verifying the integrity of the query. This reduction in communication complexity results in better performance. In the following paragraph, we provide a high-level sketch of the proposed memory integrity checking technique.

Consider a memory with n blocks. A block is the basic unit of storage in memory. The n memory blocks are partitioned into sets of size m blocks each. Thus, the number of sets in the memory is $r = (\frac{n}{m})$. Let the n blocks in memory be represented as $\{b_1, b_2, \ldots, b_n\}$ where b_i is the ith block. Then, the first set comprises blocks $\{b_1, b_2, \ldots, b_m\}$, the second set comprises blocks $\{b_{m+1}, b_{m+2}, \ldots, b_{2m}\}$, and so on, until the rth set which consists of blocks $\{b_{(r-1)m+1}, b_{(r-1)m+2}, \ldots, b_{rm}\}$. Next, the sets are divided into t non-overlapping groups containing s sets each where $t = (\frac{r}{s})$. Thus, there are two levels of hierarchy: first, we have sets which comprise blocks, and second, we have groups which comprise sets. A state value is computed for each group such that this value can be used to check the data integrity of all the sets in the group. Given a group i $(1 \leq i \leq t)$, the state value of the group is computed as follows:

- A message authentication code (MAC), h_j $(1 \leq j \leq s)$, is computed for each of the s sets in the group.
- A random value X_i is generated, and is assigned as the state value of group i.
- A transformation $T()$, parameterized on value X_i, is applied to each of the s MAC values to generate transformed MAC values, f_j $(1 \leq j \leq s)$, *i.e.*, $f_j = T(X_i, h_j)$ $(1 \leq j \leq s)$.
- Value X_i is securely stored on the processor while the s transformed MAC values, f_j $(1 \leq j \leq s)$, are stored in the external memory.

Subsequently, when a block is read from set j $(1 \leq j \leq s)$ belonging to group i $(1 \leq i \leq t)$, we do the following to check the data integrity of that block:

- Read the contents of set j, and the corresponding transformed MAC value f_j from the external memory into the processor.
- Compute the MAC value for set j, h_j, and use the state value of group i, X_i, which is stored on the processor, to compute the transformed MAC value $f'_j = T(X_i, h_j)$.

- Verify whether the computed transformed MAC value f'_j is equal to the transformed MAC, f_j, value read from memory. If yes, then, the block passes the integrity check. Else, the memory is declared to be tampered with.

The most important component of the proposed technique is transformation $T()$ which maps the MAC values of the sets to their transformed MAC values. In the proposed method, transformation $T()$ is based on Toeplitz matrices. The space complexity of the proposed scheme is $\sum_{i=1}^{t} |X_i| \leq t * p$ where p is length of the MAC value (since the length of the hash value of any set, X_i, is upper bounded by the length of the MAC value). The communication complexity of the proposed scheme is m (which is the size of a set).

6.1.2 Related work

Most of the previous work in memory integrity verification has relied on the use of a Merkle tree (or hash tree) [116] which was originally proposed as a way of authenticating data between untrusted entities using a minimum amount of memory. Blum *et al.* [117] were the first to propose the use of a hash tree for verifying the correctness of data stored in large untrusted memories and provided theoretical proofs of security. Subsequently, hash tree-based memory integrity verification has been used in various scenarios: for building secure databases using untrusted storage [118], managing the persistent state in digital rights management systems [119], verifying data structures, like stacks and queues, stored in untrusted memory of memory-constrained embedded systems such as smartcards [120], and certifying program execution on a trusted processor [121]. Though the space complexity of hash tree-based schemes is small ($O(1)$ since only the root hash is stored securely), the communication complexity is $O(log_m N)$ where m is number of children per node and N is the number of leaf nodes in the hash tree (also, the number of blocks in the memory if there is a block per leaf node). Wide usage of hash trees prompted research aimed at mitigating the performance drawback of hash-tree based memory integrity verification schemes. Gassend *et al.* [122] proposed cache-centric architectural enhancements directed at reducing the latency of tree-based integrity verification. Williams and Sirer [123] use analytical modeling to determine the size of the leaf node, *i.e.*, the number of memory blocks per leaf, that will result in an optimal performance. Both these works [122, 123] improve performance by fine-tuning implementation-specific parameters, and do not propose any conceptual changes. Any architectural recommendations, which involve increasing the size of on-chip storage, will also improve the performance of the proposed scheme. Clarke *et al.* [124] proposed an adaptive scheme for memory integrity verification which combined the benefits of log-based Merkle tree-based schemes. Due to the performance penalty of using logs, this scheme is ideally suited when memory integrity checks are not required to be done frequently. On the theoretical side, there are some works which propose hash functions that enable fast incremental hash re-computation on modified data which find application in memory integrity checking [125–128]. However, these works are broad in scope, and they do not specifically target reducing communication complexity of memory integrity checking. Our proposal reduces communication complexity compared to the existing techniques while incurring a reasonable space overhead. This is made possible by introducing some extra computations on the processor, and these operations are used for constructing the information stored in the private memory. However, these additional operations involve bit manipulations which can be significantly spedup by introducing custom functional units into the processor datapath [76]. A related scheme for reducing communication complexity of integrity checking consists of using RSA-based screening to construct a signature per basic block of code rather than one per instruction, thereby reducing the amount of signature data required to be read from memory during verification [129]. This method uses modular exponentiation operation which is computationally expensive.

The rest of the chapter is organized as follows. Section 6.2 provides formal definitions of the routines used by the proposed technique, and their corresponding security definitions. Section 6.3 gives a description of the routines employed in the proposed scheme. This section also includes a discussion of the space and communication complexities of the proposed scheme with respect to that of Merkle tree and store hash schemes. Section 6.4 gives the proofs of security and correctness of the proposed scheme. We show that the proposed scheme has very low collision probability. Section 6.5 concludes with some observations and directions for future work.

6.2 Definitions

In this section, we begin by giving a formal description of a memory integrity checker, and follow it up with a formal definition of two sub-routines, `CHECK()` and `UPDATE()`, used by our proposed technique.

6.2.1 Basic model and definitions

In our threat model, we assume the processor is secure and tamper-proof while the external memory is under the control of the adversary. This implies that the adversary cannot either observe or influence the computations performed on, and the values stored inside the processor. Any violation of the secure perimeter of the processor boundary can be detected, and will result in the erasing of all sensitive cryptographic data stored on the processor. However, the adversary can freely observe data stored at any location of the external memory, and arbitrarily change their values. We define a memory integrity checker as a program that detects any corruption of data stored in a large untrusted memory in the above defined threat model. The checker runs within a secure processor perimeter, and can be trusted. As part of its operation, the checker maintains a *state* of the external memory in a much smaller private on-chip memory, and uses this state information to detect any adversarial corruption of values stored in the untrusted memory. Processor reads and changes the contents of external memory using load and store instructions, respectively. The checker program updates its state on a store instruction, and subsequently the validity of data read from the same memory location during a load instruction is verified using the state information. The checker program handles load and store instructions issued by the processor in the following manner:

- STORE(x,y): The checker updates its *state* variable to reflect that value y was written into memory location x.

- LOAD(x): Using its *state* variable, the checker verifies if value z read from memory location x is the same as the value last written into it. If the checker determines with sufficiently high confidence that value z is uncorrupted, then it passes it to the processor with the *"accept"* signal. Else, it sends a *"reject"* signal to the processor.

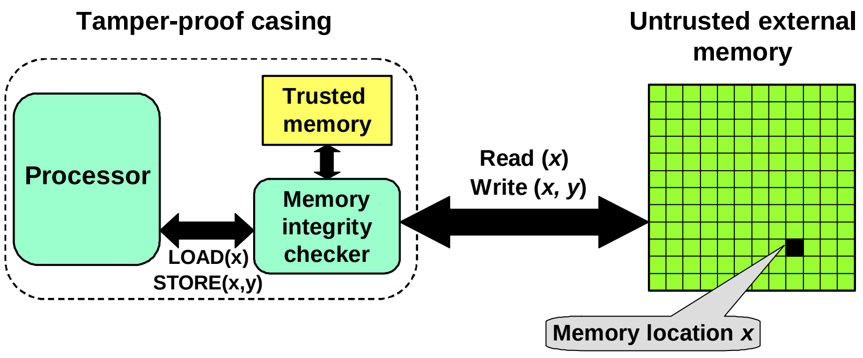


Figure 6.3: Memory integrity checker

Figure 6.3 illustrates the working of a memory integrity checking program. The size of the trusted memory used by the checker to store the state variable determines its space complexity. The functioning of a memory checker can be formally specified as follows:

Definition 1: *A memory integrity checker is a probabilistic program C that maintains a private value state which is updated on writes to an untrusted memory M. If y is a value written to location x in M, and z the value read subsequently from the same location, then the following conditions hold:*

- *Completeness:* $Pr[\forall x, y, z \textit{ where } y = z \textit{ s.t. } C(state, x, z) = accept] \geq (1 - \epsilon)$.
- *Soundness:* $Pr[\forall x, y, z \textit{ where } y \neq z \textit{ s.t. } C(state, x, z) = accept] < \epsilon$

In the definition above, 's.t.' stands for "such that" (and holds true for the rest of the chapter).

Completeness implies that valid inputs are accepted by C with overwhelming probability, and soundness implies that wrong inputs are accepted by C with negligible probability. A challenge is to compute the state in such a way that its size is reasonable while being able to detect corruption of values in memory with a high probability, *i.e.*, make ϵ as small as possible.

The proposed memory integrity checking technique comprises two routines: `UPDATE()` and `CHECK()`. The `UPDATE()` routine constructs and updates the state of the checker to reflect new additions and modifications to existing values in untrusted memory, and `CHECK()` utilizes the state of the checker to determine whether the values read in from the external memory are corrupted. Thus, the `UPDATE()` routine is executed on a store instruction, and the `CHECK()` routine is executed on a load instruction. Usually, an executing program issues higher number of loads than stores [115]. Therefore, in general, it is advisable to formulate the `CHECK()` routine to have as low latency as possible while the `UPDATE()` routine can tolerate greater latency. We now proceed to formally define both the routines.

Definition 2: *Let M be the memory comprising n memory blocks which are grouped into r sets having m blocks each (where $r = \frac{n}{m}$). Furthermore, we form non-overlapping groups having s sets each such that $t = (\frac{r}{s})$. The t groups are represented as $\{M^1, M^2, \ldots, M^t\}$ where each group M^i $(1 \leq i \leq t)$ comprises s sets represented as $\{M_1^i, M_2^i, \ldots, M_s^i\}$.* `UPDATE()` *takes group M^i as input, and outputs a private value $STATE_{M^i}$ and public values $(MAC_{M_1^i}, MAC_{M_2^i}, \ldots, MAC_{M_s^i})$,* i.e., *$(STATE_{M^i}, MAC_{M_1^i}, MAC_{M_2^i}, \ldots, MAC_{M_s^i})$ $\leftarrow$* `UPDATE(`M^i`)`. *$STATE_{M^i}$ is the state value of group M^i. It is stored in the on-chip memory. $(MAC_{M_1^i}, MAC_{M_2^i}, \ldots, MAC_{M_s^i})$ are MAC values of sets in group M^i. They are stored in the external memory. The* `CHECK`*() procedure uses both the state value and the MAC value to verify the integrity of a set belonging to group M^i. The completeness and soundness properties for the* `CHECK`*() procedure are defined as follows:*

- *Completeness: $Pr[\forall M^i$, $(STATE_{M^i}, MAC_{M_1^i}, MAC_{M_2^i}, \ldots, MAC_{M_s^i}) \leftarrow$*`UPDATE(`$M^i$`)`*, where $M_j^i \in M^i$ s.t.* `CHECK(`$STATE_{M^i}, MAC_{M_j^i}, j, M_j^i$`)`$= 1] \geq (1 - \epsilon)$
- *Soundness: $Pr[\forall M^i$, $(STATE_{M^i}, MAC_{M_1^i}, MAC_{M_2^i}, \ldots, MAC_{M_s^i}) \leftarrow$*`UPDATE(`$M^i$`)`*, where $M_j \notin M^i$ s.t.* `CHECK(`$STATE_{M^i}, MAC_{M_j}, j, M_j$`)`$= 1] < \epsilon$

A stronger form of soundness is *collision resistance* which says that for any group M^i, it should be very difficult for an adversary to modify its jth set M^i_j $(1 \leq j \leq s)$ and its MAC value, $MAC_{M^i_j}$ to M'^i_j and $MAC'_{M^i_j}$, respectively, such that they can successfully pass the `CHECK`() routine, *i.e.*, `CHECK(`$STATE_{M^i}$, $MAC'_{M^i_j}$,j,M'^i_j`)` $= 1$. It also includes the case where $MAC_{M^i_j}$ is same as the $MAC'_{M^i_j}$. Collision resistance is an important property which should be satisfied by any memory integrity verification protocol.

6.3 Proposed memory integrity verification technique

In this section, we present implementation details of the `UPDATE`() and `CHECK`() routines introduced in the previous section. For the rest of the chapter, we adopt the following notation. We consider memory M to consist of n memory blocks which are arranged into r sets having m blocks each such that $r = (\frac{n}{m})$. The r sets are divided into t non-overlapping groups containing s sets each where $t = (\frac{r}{s})$. Thus, $M = \{M^1, M^2, \ldots, M^t\}$ where M^i $(1 \leq i \leq t)$ is a group, and $M^i = \{M^i_1, M^i_2, \ldots, M^i_s\}$ where M^i_j $(1 \leq j \leq s)$ is a set belonging to group M^i (every set M^i_j contains m memory blocks). We can see that total number of blocks in memory M is equal to $n = r * m = (t * s) * m$. We begin this section by providing details about the `UPDATE`() routine (Section 6.3.1), followed by the implementation of the `CHECK`() routine (Section 6.3.2), and conclude with a discussion of the space and communication complexities of the proposed technique (Section 6.3.3).

6.3.1 `UPDATE()` routine

The `UPDATE`() routine is called on every store instruction issued by the processor, and it is responsible for building and updating the state information stored in the private on-chip memory of the memory checker program. The state information is formulated such that it can be used to detect any adversarial corruption of data stored in memory by legitimate store operations of executing programs. For any given memory $M = \{M^1, M^2, \ldots, M^t\}$, a distinct state value X_i $(1 \leq i \leq t)$ is generated for each of its t groups. These values are stored in the private memory of the checker. In addition, MAC values are generated for all the s sets in each of the t groups, and these MAC values are stored in the memory. Let $M^i = \{M_1^i, M_2^i, \dots, M_s^i\}$ be the ith group with its s sets. The state value of the group and MAC values of the constituent sets are generated by `UPDATE`() in the following manner:

1. Compute the MAC codes $h_j^i = \texttt{HMAC(}M_j^i\texttt{)}$, $1 \leq j \leq s$, for all the s sets of group M^i. The function `HMAC`() is a keyed MAC function which takes an arbitrary-sized input and outputs a p-bit MAC code. The key of the function is stored securely on the processor, and is not known to an adversary. We represent p-bit MAC code of set M_j^i as $h_j^i = \{h_{j1}^i, h_{j2}^i, \dots, h_{jp}^i\}$ $(1 \leq j \leq s)$ where h_{jk}^i $(1 \leq k \leq p)$ represents the kth bit of h_j^i.

2. Generate a p-bit random vector $X_i = \{X_{i1}, X_{i2}, \dots, X_{ip}\}$ where X_{ik} $(1 \leq k \leq p)$ represents the kth bit of X_i.

3. Construct the $(p \times p)$ bit matrix B_i by p successive right circular shifts of vector X_i. Matrix B_i is shown below:

$$\begin{bmatrix} X_{i1} & X_{i2} & \dots & X_{i(p-1)} & X_{ip} \\ X_{ip} & X_{i1} & \dots & X_{i(p-2)} & X_{i(p-1)} \\ & & \dots & & \\ X_{i3} & X_{i4} & \dots & X_{i1} & X_{i2} \\ X_{i2} & X_{i3} & \dots & X_{ip} & X_{i1} \end{bmatrix}$$

   The first row of matrix B_i is vector X_i, the second row is vector X_i right circular shifted by one bit, the third row is vector X_i right circular shifted by two bits, and so on. We can see that B_i is a Toeplitz matrix, *i.e.*, elements along the diagonals are equal.

4. Map the MAC codes, h_j^i $(1 \leq j \leq s)$, to transformed MAC codes, f_j^i, by multiplying them with matrix B_i, *i.e.*, $f_j^i = B_i * h_j^i$, $1 \leq j \leq s$. The multiplication, $B_i * h_j^i$, is

performed modulo 2, and is realized as follows:

$$\begin{bmatrix} X_{i1} & X_{i2} & \dots & X_{i(p-1)} & X_{ip} \\ X_{ip} & X_{i1} & \dots & X_{i(p-2)} & X_{i(p-1)} \\ & & \dots & & \\ X_{i3} & X_{i4} & \dots & X_{i1} & X_{i2} \\ X_{i2} & X_{i3} & \dots & X_{ip} & X_{i1} \end{bmatrix} * \begin{bmatrix} h^i_{j1} \\ h^i_{j2} \\ \dots \\ h^i_{j(p-1)} \\ h^i_{jp} \end{bmatrix}$$

$$= \begin{bmatrix} X_{i1} \cdot h^i_{j1} \oplus X_{i2} \cdot h^i_{j2} \oplus \dots \oplus X_{ip} \cdot h^i_{jp} \\ X_{ip} \cdot h^i_{j1} \oplus X_{i1} \cdot h^i_{j2} \oplus \dots \oplus X_{i(p-1)} \cdot h^i_{jp} \\ \dots \\ X_{i3} \cdot h^i_{j1} \oplus X_{i4} \cdot h^i_{j2} \oplus \dots \oplus X_{i2} \cdot h^i_{jp} \\ X_{i2} \cdot h^i_{j1} \oplus X_{i3} \cdot h^i_{j2} \oplus \dots \oplus X_{i1} \cdot h^i_{jp} \end{bmatrix}$$

The multiplication of MAC value h^i_j with matrix B_i realizes transformation $T()$ mentioned in Section 6.1.1. We represent the p-bit transformed MAC code as $f^i_j = \{f^i_{j1}, f^i_{j2}, \dots, f^i_{jp}\}$, $(1 \leq j \leq s)$, where f^i_{jk} $(1 \leq k \leq p)$ is the kth bit of f^i_j.

5. Bit vector X_i is the state value of group M^i, and is stored in the private memory of the memory checker. The s transformed MAC values f^i_j $(1 \leq j \leq s)$ corresponding to the sets in group M^i are stored in the external memory.

In the previous paragraph, we enumerated the steps for generating the state value and the transformed MAC values for a group M^i. Subsequently, when any set belonging to group M^i is modified (by a store instruction), the state value and the transformed MAC values corresponding to group M^i have to be updated. Let us assume that the kth set, M^i_k $(k \in \{1, 2, \dots, s\})$, in group M^i is updated, and let X_i be the present state value. The updated state value and the transformed MAC codes are determined as follows:

1. Compute the new MAC value, h'^i_k, of the updated kth set, M^i_k.

2. A new p-bit random value X'_i is generated. Let $X'_i = X_i \oplus \Delta X_i$, where ΔX_i is a p-bit vector which is the XOR difference of vectors X_i and X'_i. Let B'_i, B_i, and ΔB_i

be $(p \times p)$ matrices constructed by p right circular shifts of vectors X'_i, X_i, and ΔX_i, respectively.

3. Compute the new transformed MAC value, f'^i_k, for the kth set as $B'_i * h'^i_k$, and write it to memory.

4. The transformed MAC values of the unmodified sets in the group, M^i_j, $j \in \{1, 2, \ldots, s\} - \{k\}$, have to be updated according to the new state value X'_i. The $(s-1)$ new transformed MAC values f'^i_j, $j \in \{1, 2, \ldots, s\} - \{k\}$, are computed as follows:

$$\begin{aligned} f'^i_j &= B'_i * h^i_j \\ &= (B_i \oplus \Delta B_i) * h^i_j \\ &= (B_i * h^i_j) \oplus (\Delta B_i * h^i_j) \\ &= f^i_j \oplus (\Delta B_i * (B_i^{-1} * f^i_j)) \\ &= f^i_j \oplus ((\Delta B_i * B_i^{-1}) * f^i_j) \end{aligned}$$

   where B_i^{-1} is the inverse of matrix B_i, and f^i_j is the old transformed MAC value of set M^i_j $(j \in \{1, 2, \ldots, s\} - \{k\})$. The updated values are written to the memory.

The inverse of matrix B_i always exists since the matrix is constructed such that it has full rank (more discussion in Section 6.4). As shown above, the transformed MAC values of the unmodified sets in group M^i can be updated with respect to new state value X'_i by using only the old transformed MAC values. The old transformed MAC values f^i_j are read from memory, updated to f'^i_j (as shown above), and the new values are written back into memory. The main advantage of this updating procedure is that we do not need to read in the contents of the unmodified sets in order to update their transformed MAC values. In case any of the old transformed MAC values were tampered with, then, the new transformed MAC value will fail the `CHECK`() routine, and tampering detected. The main reason for refreshing the state value of a group, when a constituent set is modified, is to prevent replay attacks against the proposed scheme.

6.3.2 CHECK() routine

The CHECK() routine is executed when the processor issues a load instruction. This routine checks whether the value loaded from a particular location in the memory is the same as the value placed there by the most recent store instruction. It does this with the aid of the state information stored on the processor, and the MAC values stored in the memory. Consider that a memory block is read from the kth set of group M^i, M^i_k. We need to verify the integrity of data in set M^i_k as follows:

1. Read in the p-bit transformed MAC value f^i_k of set M^i_k from memory.
2. Compute the p-bit MAC value of set M^i_k, $h^i_k = \texttt{HMAC}(M^i_k)$.
3. Retrieve the state value X_i from the private on-chip memory, and construct the $(p \times p)$ bit-matrix B_i by p right circular shifts of vector X^i.
4. Calculate the p-bit transformed MAC value, $f'^i_k = B_i * h^i_k$.
5. Check whether the computed transformed MAC value, f'^i_k, is equal to the transformed MAC value read from memory, f^i_k. If yes, then the set M^i_k is considered to be legitimate, and the block read from memory is sent to the processor. Else, the memory M is said to be corrupted.

The difficulty of an adversary to successfully fool the CHECK() routine is determined by the hardness of finding two p-bit vectors y_1 and y_2 such that $B_i * y_1 = B_i * y_2$. In the next section, we present a detailed security analysis to prove that it is very difficult to find such vectors, y_1 and y_2, for any matrix B_i. Moreover, the adversary does not know matrix B_i which makes his job even more difficult.

6.3.3 Space and communication complexity

We saw in the previous section that a state value is computed for each group in memory, and these values are stored in a secure on-chip memory. Therefore, for a memory M comprising t groups, $\{M^1, M^2, \ldots, M^t\}$, the space complexity of the proposed scheme is given by $\sum_{i=1}^{t} |X_i| = t \cdot p = (\frac{1}{s}) \cdot (\frac{n}{m}) \cdot p$, where $|X_i|$ denotes the length of vector X_i (the memory parameters n, m, s and t are explained at the beginning of Section 6.3). The communication complexity of the proposed scheme is dependent on whether the memory operation is a read or a write, and is given as follows:

- Read operation: When a block is read from a memory set, the `CHECK`() routine requires all the m blocks in the set, and the corresponding transformed MAC value, in order to perform integrity verification. Therefore, the communication complexity of integrity verification on a read is $(m + 1)$ memory accesses (the additional query is for reading in the transformed MAC value).

- Write operation: When a block in a memory set is written to, the `UPDATE`() routine requires all the m blocks in the set, and the corresponding transformed MAC value. In addition, the routine also reads in the transformed MAC values of the remaining $(s-1)$ sets in the group for updating them. Therefore, the communication complexity of integrity verification on a write operation is $(m + s)$. *However, the write operation does not need to wait for the $(s - 1)$ transformed MAC values of the unmodified sets in order for it to finish executing.*

We compare the space and communication complexities of our proposed scheme with the two schemes mentioned in Section 6.1, *Merkle tree* and *store hash.* The Merkle tree scheme builds a tree-based hash structure whose leaf nodes are the r memory sets, and internal nodes hold hash values. Only the hash value in the root node is stored on the processor. This scheme has very low space complexity, and this advantage comes at the cost of high communication complexity. The store hash scheme stores the hash values corresponding to all the r memory sets on the processor which results in a low communication complexity. This advantage comes at the cost of a higher space complexity. Table 6.1 lists the communication and space complexities of the proposed scheme with those of Merkle tree and store hash. For read operations, the proposed scheme offers the low communication complexity advantage of the store hash scheme while imposing a much smaller space overhead (due to the ($\frac{1}{s}$) factor in space complexity). For write operations, the communication complexity of the proposed technique is slightly higher than that of the store hash scheme. Moreover, the factor s contributes a small additive overhead to communication complexity while giving a significant multiplicative reduction factor ($\frac{1}{s}$) to the space complexity of the proposed scheme.

Table 6.1: Comparison of space and communication complexity

Scheme	Communication complexity (Memory accesses)	Space complexity (Bits)
Merkle tree	$2m + \log_2(\frac{n}{m})$	p
Store hash	m	$(\frac{n}{m})p$
Proposed		
Read	$(m+1)$	$(\frac{1}{s})(\frac{n}{m})p$
Write	$(m+s)$	$(\frac{1}{s})(\frac{n}{m})p$

6.4 Security analysis

In this section, we analyze the security of the proposed memory integrity verification technique. First, we present a proof to show that the proposed technique has a low collision probability. This proof is based on a lemma which provides the necessary condition for the Toeplitz matrices used in our scheme to have full rank. Next, we will present an argument to show that the proposed scheme is complete and sound.

Lemma 1 (Vazirani [130]): *Let q be a prime with 2 a primitive root mod q, and $\overrightarrow{x}$ be a q-bit vector which is not all 0's and all 1's. Then, $(q \times q)$ matrix A over GF(2), whose rows are q cyclic shifts of vector $\overrightarrow{x}$, has rank $\geq (q-1)$.*

In other words, the above lemma says that any $(q-1 \times q)$ matrix, A^{q-1}, formed by choosing $(q-1)$ rows of matrix A has full rank. Hence, we choose a $(p+1)$-bit vector as the state value X such that $(p+1)$ is a prime with 2 as its primitive root (in the previous sections, we mentioned state value X to be a p-bit vector in order not to complicate things). From Lemma 1, we can see that the $(p \times p+1)$ matrix B formed by p right circular shifts of state value X has rank p. Since the MAC values are p-bit, we have to pad them with 1 before multiplying them with matrix B to obtain the p-bit transformed MAC values. For the remainder of the section, we use B to refer to a full rank $(p \times p+1)$ matrix formed by p right circular shifts of a $(p+1)$-bit state value. Now, we compute the probability of finding two p-bit vectors y_1 and y_2 such that $B*(1\circ y_1) = B*(1\circ y_2)$ where $\circ$ denotes concatenation. This probability is required to bound the collision probability of the proposed scheme. The following proof uses some techniques from [131].

Lemma 2: *The probability of finding two distinct p-bit bit vectors y_1 and y_2 such that $B*(1\circ y_1) = B*(1\circ y_2)$ is equal to 2^{-p}.*

Proof: Let $z = (1\circ y_1) \oplus (1\circ y_2)$. The $(p+1)$-bit vector $z = \{z_1, z_2, \ldots, z_{p+1}\}$ (where z_i is the ith bit of vector z) is not either all zeros (since y_1 and y_2 are distinct) or all ones (due to the leading 1). Now, we want to find the probability that $B*z = 0$, *i.e.*, determine the probability of picking the bits of matrix B such that the product $B*z$ evaluates to 0. We proceed as follows:

$$\begin{aligned}
B*z &= \begin{bmatrix} b_1 & b_2 & \ldots & b_p & b_{p+1} \\ b_{p+1} & b_1 & \ldots & b_{p-1} & b_p \\ & & \ldots & & \\ b_3 & b_4 & \ldots & b_1 & b_2 \\ b_2 & b_3 & \ldots & b_{p+1} & b_1 \end{bmatrix} * \begin{bmatrix} z_1 \\ z_2 \\ \ldots \\ z_p \\ z_{p+1} \end{bmatrix} \\
&= \begin{bmatrix} b_1 \cdot z_1 \oplus b_2 \cdot z_2 \oplus \ldots \oplus b_p \cdot z_p \oplus b_{p+1} \cdot z_{p+1} \\ b_{p+1} \cdot z_1 \oplus b_1 \cdot z_2 \oplus \ldots \oplus b_{p-1} \cdot z_p \oplus b_p \cdot z_{p+1} \\ \ldots \\ b_4 \cdot z_1 \oplus b_5 \cdot z_2 \oplus \ldots \oplus b_2 \cdot z_p \oplus b_3 \cdot z_{p+1} \\ b_3 \cdot z_1 \oplus b_4 \cdot z_2 \oplus \ldots \oplus b_1 \cdot z_p \oplus b_2 \cdot z_{p+1} \end{bmatrix}
\end{aligned}$$

$$= \begin{bmatrix} b_1 \cdot z_1 \oplus b_2 \cdot z_2 \oplus \ldots \oplus b_p \cdot z_p \oplus b_{p+1} \cdot z_{p+1} \\ b_1 \cdot z_2 \oplus b_2 \cdot z_3 \oplus \ldots \oplus b_p \cdot z_{p+1} \oplus b_{p+1} \cdot z_1 \\ \ldots \\ b_1 \cdot z_{p-1} \oplus b_2 \cdot z_p \oplus \ldots \oplus b_p \cdot z_{p-3} \oplus b_{p+1} \cdot z_{p-2} \\ b_1 \cdot z_p \oplus b_2 \cdot z_{p+1} \oplus \ldots \oplus b_p \cdot z_{p-2} \oplus b_{p+1} \cdot z_{p-3} \end{bmatrix} (\textit{rearranging row terms})$$

$$= \begin{bmatrix} z_1 & z_2 & \cdots & z_p & z_{p+1} \\ z_2 & z_3 & \cdots & z_{p+1} & z_1 \\ & & \cdots & & \\ z_{p-1} & z_p & \cdots & z_{p-3} & z_{p-2} \\ z_p & z_{p+1} & \cdots & z_{p-2} & z_{p-1} \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ \ldots \\ b_p \\ b_{p+1} \end{bmatrix}$$

$$= Z * b$$

where Z is a $(p \times p+1)$ matrix formed by p circular shift of $(p+1)$-bit vector z, and $b = \{b_1, b_2, \ldots, b_{p+1}\}$ is a $(p+1)$-bit vector. From Lemma 1, we know that matrix Z has rank p which implies that the homogeneous solution of $Z * b = 0$ has only one free variable, *i.e.*, p bits of vector b are uniquely determined. Therefore, the probability of $B * z = Z * b = 0$ is 2^{-p}.

QED

Using the above lemma, we compute the collision probability of the proposed memory integrity verification scheme. By collision, we define an event where an adversary is able to change a memory set M^i_j in a group to M'^i_j such that both of the sets map to the same transformed MAC value f^i_j, for some state value X_i (which is not known to the adversary).

Lemma 3: *The collision probability of the proposed memory integrity verification scheme is upper bounded by* $2^{-p/2}$.

Proof: Consider a set M^i_j belonging to group M^i. We want to compute the probability that an adversary can change set M^i_j to M'^i_j such that both of them map to the same p-bit transformed MAC value f^i_j for some state value X_i. Let h^i_j and h'^i_j be the p-bit MAC values corresponding to sets M_j^i and $M_j'^i$, respectively. B is a $(p \times p+1)$ matrix constructed by p circular shifts of $(p+1)$-bit vector X_i. A collision occurs in two cases. First, the MAC values of the two sets collide, or second, the MAC values are distinct, but, they map to the same transformed MAC value. In Lemma 2, we computed the probability of two distinct MAC values mapping to the same transformed MAC value. Thus, the collision probability of the proposed scheme is given by

$$\begin{aligned} Pr[Collision] &= Pr[h_j^i = h_j'^i] + Pr[h_j^i \neq h_j'^i] \cdot Pr[B*(1 \circ h_j^i) = B*(1 \circ h_j'^i)] \\ &= \frac{1}{2^{p/2}} + (1 - \frac{1}{2^{p/2}}) \cdot \frac{1}{2^p} \\ &= \frac{2^p + 2^{p/2} - 1}{2^p \cdot 2^{p/2}} \\ &\approx \frac{2^p + 2^{p/2}}{2^p \cdot 2^{p/2}} \\ &= \frac{2^{p/2}(2^{p/2}+1)}{2^p \cdot 2^{p/2}} \\ &\approx \frac{1}{2^{p/2}} \end{aligned}$$

QED

The proposed scheme is complete and sound. Completeness follows from the fact that, as long as the set is not modified, its MAC value is always mapped to the same transformed MAC value by the full rank matrix used for the transformation. And, from Lemma 3, we can see that probability of an adversary being able to modify a set such that it fools the CHECK() routine is negligible. Also, the operation of the memory checker leaks no information about state values to the adversary. The state values are randomly generated, and stored securely on the processor. Thus, the best an adversary can do is guess, and the probability of him making the correct one is 2^{-p}. Also, the transformed MAC values stored in the memory do not give much information to the adversary about the state values. This is so because the adversary has no knowledge of the MAC values since he does not know the key used in the HMAC() algorithm. Thus, this eliminates any possibility of solving a system of linear equations to solve for the state values. For the above reason, it is absolutely important to uses a `HMAC()` function and not a cryptographic hash function.

6.5 Chapter summary

In this chapter, we presented a novel memory integrity verification technique which has a low communication complexity while incurring a modest space complexity overhead. The proposed scheme gets this advantage by employing some additional operations for building the state values. However, most of these additional computations are bit manipulations which can be done very efficiently. In addition, we presented an analysis of the security of the proposed scheme showing that it has low collision probability.

Chapter 7

Conclusions and Future Work

In this dissertation, we introduced solutions pertaining to three aspects of secure embedded systems: enhancing performance, extending battery-life and improving robustness to software-based side-channel attacks. However, the design of secure and efficient embedded systems is a challenging task with many problems remaining to be solved. In this chapter, we enumerate various possible avenues for future research in this area.

The push in embedded systems is toward smaller and more resource-constrained devices, such as sensor nodes and RFIDs. Implementing security functionality in these small form factor embedded systems is an interesting challenge due to their operating constraints of low computational capability and finite battery-derived energy supply. In addition, these devices are quite susceptible to non-invasive attacks, like side-channel attacks, which are based on exploiting the correlation between the data manipulated during cryptographic computation by a device, and either the time taken or power dissipated or electromagnetic radiation emitted by the device during the computation. These observations motivate the need for novel embedded system-specific security mechanisms that are light-weight and robust against side-channel attacks. Devising such a security mechanism would involve research at three levels of security implementation: security protocols, cryptographic algorithms, and hardware-software systems which execute the security protocols and cryptographic algorithms. Topics of research at these levels are as follows.

- Formulate scalable security protocols whose security level and energy consumption can be varied by altering the number of protocol rounds and the complexity of operations in each round.

- Work at the cryptographic algorithm level includes

  - Develop cryptographic algorithms based on provably difficult problems which involve simpler operations suited for implementation on embedded systems. Examples of problems satisfying these criteria include learning parity with noise, sum of k mins, *etc.,* which have been advocated for use in RFID authentication mechanisms.
  - Modify existing algorithms to reduce their vulnerability to side-channel attacks. It would be interesting to look into areas, like coding theory, redundant number systems *etc.,* to achieve this objective.
  - In this dissertation, we analyzed the energy consumption characteristics of different block ciphers with respect to their constituent round operations. This information can be used to devise new energy-efficient ciphers suited for resource-constrained embedded systems.

- Improve existing hardware and software architectures to achieve the following:

  - Efficient processing of newer security protocols and cryptographic algorithms by leveraging latest developments in hardware/software co-design techniques, like custom instruction addition, co-processor design, FPGAs, *etc.*
  - Reduce susceptibility to side-channel attacks through the use of hardware techniques that reduce correlation between data values and side-channel information, like power, time, *etc.*
  - Improve resistance of embedded systems to malware, like viruses, worms and Trojan horses. Malware is starting to become a problem in embedded systems, as demonstrated by the Cabir virus which infected cell-phones.

References

[1] Y. L. Yin, “The RC5 encryption algorithm: Two years on,” *RSA Laboratories’ Cryptobytes*, vol. winter, pp. 14–15, 1997.

[2] *CERT E-crime watch.* www.cert.org/archive/pdf/2004eCrimeWatchSummary.pdf, 2005.

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

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

[5] *Epaynews survey.* http://www.epaynews.com/statistics/scardstats.html, 2005.

[6] S. Ravi, A. Raghunathan, and N. Potlapally, “Securing wireless data: System architecture challenges,” in *Proc. Int. Symp. System Synthesis*, pp. 195–200, Oct 2002.

[7] D. W. Carman, P. S. Kruus, and B. J. Matt, *Constraints and Approaches for Distributed Sensor Security.* Network Associates Labs Tech. Rep. 00-010, 2000.

[8] K. Lahiri, A. Raghunathan, and S. Dey, “Battery-driven system design: A new frontier in low power design,” in *Proc. Joint Asia and South Pacific Design Automation Conf. / Int. Conf. VLSI Design*, pp. 261–267, Jan. 2002.

[9] A. Hodjat and I. Verbauwhede, “The energy cost of secrets in ad-hoc networks,” in *Proc. IEEE CAS Wkshp. Wireless Communication and Networking*, Sept. 2002.

[10] Rockwell Scientific Inc. http://wins.rockwellscientific.com.

[11] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar, "SPINS: Security protocols for sensor networks," *Wireless Networks*, vol. 8, no. 5, pp. 521–534, 2002.

[12] Y. W. Law, S. Dulman, S. Etalle, and P. J. M. Havinga, *Assessing Security-Critical Energy-efficient Sensor Networks.* Univ. of Twente, The Netherlands, Tech. Rep. TR-CTIT-02-18, July 2002.

[13] M. Jakobsson and D. Pointcheval, "Mutual authentication for low-power mobile devices," in *Proc. Financial Cryptography*, pp. 178–195, Feb. 2001.

[14] D. S. Wong and A. H. Chan, "Mutual authentication and key exchange for low power wireless communications," in *Proc. IEEE MILCOM Conf.*, pp. 39–43, Oct. 2001.

[15] R. Karri and P. Mishra, "Minimizing energy consumption of secure wireless session with QoS constraints," in *Proc. Int. Conf. Communications*, pp. 2053–2057, May 2002.

[16] N. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha, "Analyzing the energy consumption of security protocols," in *Proc. Int. Symp. Low Power Design*, pp. 30–35, Aug. 2003.

[17] N. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha, "A study of the energy consumption characteristics of cryptographic algorithms and security protocols," *IEEE Trans. Mobile Comput.*, vol. 5, no. 2, pp. 128–143, 2006.

[18] D. Boneh and N. Daswani, "Experimenting with electronic commerce on the PalmPilot," in *Proc. Financial Cryptography*, pp. 1–16, Feb. 1999.

[19] W. Freeman and E. Miller, "An experimental analysis of cryptographic overhead in performance-critical systems," in *Proc. Int. Symp. Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 348–357, Oct. 1999.

[20] S. K. Miller, "Facing the challenges of wireless security," *IEEE Computer*, pp. 46–48, July 2001.

[21] G. Apostolopoulos, V. Peris, P. Pradhan, and D. Saha, "Securing electronic commerce: Reducing the SSL overhead," *IEEE Network*, pp. 8–16, July 2000.

[22] D. S. Wong, H. H. Fuentes, and A. H. Chan, "The performance measurement of cryptographic primitives on Palm devices," in *Proc. Annual Computer Security Applications Conf.*, pp. 92–101, Dec. 2001.

[23] S. Ravi, A. Raghunathan, N. Potlapally, and M. Shankaradass, "System design methodologies for wireless security processing platform," in *Proc. Design Automation Conf.*, pp. 777–782, June 2002.

[24] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. on Embedded Computing Systems*, vol. 3, pp. 461–491, Aug. 2004.

[25] Z. Shi and R. Lee, "Bit permutation instructions for accelerating software cryptography," in *Proc. IEEE Int. Conf. Application-specific Systems, Architectures and Processors*, pp. 138–148, July 2000.

[26] J. Burke, J. McDonald, and T. Austin, "Architectural support for fast symmetric-key cryptography," in *Proc. Int. Conf. Arch. Support for Prog. Lang. & Operating Systems*, pp. 178–189, Nov. 2000.

[27] R. B. Lee, Z. Shi, and X. Yang, "Efficient permutation instructions for fast software cryptography," *IEEE Micro*, vol. 21, no. 6, pp. 56–69, 2001.

[28] A. Hodjat, P. Schaumont, and I. Verbauwhede, "Architectural design features of a programmable high throughput AES coprocessor," in *Proc. Int. Conf. Information Technology: Coding and Computing*, pp. 498–502, Apr. 2004.

[29] *SmartMIPS*. http://www.mips.com.

[30] *ARM SecurCore.* http://www.arm.com.

[31] *OMAP 1610 Platform.* Texas Instruments Inc. (http://www.ti.com).

[32] *MP211 Application Processor.* NEC Electronics (http://www.necel.com).

[33] M. Burnside and A. Keromytis, "Accelerating application-level security protocols," in *Proc. Int. Conf. Networks*, pp. 313–318, Oct. 2003.

[34] S. Miltchev, S. Ioannidis, and A. Keromytis, "A study of the relative costs of network security protocols," in *Proc. USENIX Annual Technical Conf.*, pp. 41–48, June 2002.

[35] *Hifn 7851 security processors.* http://www.hifn.com/products/7851.html, 2004.

[36] *NITROX security processors.* http://www.cavium.com/processor_security.html, 2004.

[37] *Intel IXP2850 network processor.* http://www.intel.com/design/network/ products/npfamily/ixp2850.htm, 2004.

[38] N. Shah, *Understanding network processors.* http://www-cad.eecs.berkeley.edu/ ~niraj/web/research/network_processors.htm, 2001.

[39] R. Friend, "Making the gigabit ipsec vpn architecture secure," *IEEE Computer*, pp. 54–60, June 2004.

[40] N. Potlapally, S. Ravi, A. Raghunathan, R. B. Lee, and N. K. Jha, "Impact of configurability and extensibility on IPSec protocol execution on embedded processors," in *Proc. Int. Conf. VLSI Design*, pp. 299–304, Jan. 2006.

[41] N. Potlapally, S. Ravi, A. Raghunathan, R. B. Lee, and N. K. Jha, "Configuration and extension of embedded processors to optimize ipsec protocol execution," *IEEE Trans. VLSI Systems*, vol. 15, no. 5, pp. 605–609, 2007.

[42] *Lightweight IPSec Implementation.* http://www.hta-bi.bfh.ch/Projects/ipsec/, 2004.

[43] S. Ravi, A. Raghunathan, and S. T. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *Proc. Int. Conf. VLSI Design*, pp. 605–611, Jan. 2005.

[44] E. Biham and A. Shamir, "Differential cryptanalysis of the full 16-round DES," in *Proc. Crypto '92*, pp. 487–496, Aug. 1992.

[45] M. Matsui, "Linear cryptanalysis method for DES cipher," in *Proc. Crypto '94*, pp. 386–397, Aug. 1994.

[46] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Proc. Crypto '96*, pp. 104–113, Aug. 1996.

[47] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proc. Crypto '99*, pp. 388–397, Aug. 1999.

[48] T. Messerges, E. A. Dabbish, and R. H. Sloan, "Examining smart-card security under the threat of power analysis attacks," *IEEE Trans. Computers*, vol. 51, pp. 541–552, May 2002.

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

[50] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Proc. Crypto '97*, pp. 513–525, Aug. 1997.

[51] S. Skorobogatov and R. Anderson, "Optical fault induction attack," in *Proc. Cryptographic Hardware and Embedded Systems Wkshp.*, pp. 2–12, Aug. 2002.

[52] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *J. Computer Security*, vol. 8, no. 2, pp. 141–158, 2000.

[53] J. S. Coron, D. Naccache, and P. Kocher, "Statistics and information leakage," *ACM Trans. Embedded Comput. Systems*, vol. 3, pp. 492–508, Aug. 2004.

[54] N. Potlapally, S. Ravi, A. Raghunathan, N. K. Jha, and R. B. Lee, "Satisfiability-based framework for enabling side-channel attacks on cryptographic software," in *Proc. IEEE Design Automation and Test in Europe (DATE) Designers Forum*, pp. 18–23, Apr. 2006.

[55] N. Potlapally, S. Ravi, A. Raghunathan, N. K. Jha, and R. B. Lee, “Aiding side-channel attacks on cryptographic software with satisfiability-based analysis,” *IEEE Trans. VLSI Systems*, vol. 15, no. 4, pp. 465–470, 2007.

[56] I. S. Bichl, “Cryptanalysis of the data encryption standard by the method of formal coding,” in *Proc. Eurocrypt ’82*, pp. 235–255, May 1982.

[57] F. Massacci and L. Marraro, “Logical cryptanalysis as a SAT problem,” *J. Automated Reasoning*, vol. 24, pp. 165–203, Feb. 2000.

[58] U. S. Department of Commerce, *The Emerging Digital Economy II.* http://www.esa.doc.gov/508/esa/TheEmergingDigitalEconomyII.htm, 1999.

[59] W. W. W. Consortium, *The World Wide Web Security FAQ.* http://www.w3.org/Security/faq/www-security-faq.html, 1998.

[60] *LAN MAN Standards Committee of the IEEE Computer Society.* Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification: IEEE standard 802.11, 1990.

[61] *IPSec Working Group.* http://www.ietf.org/html.charters/ipsec-charter.html.

[62] *SSL 3.0 Specification.* http://wp.netscape.com/eng/ssl3/.

[63] *Wireless Application Protocol 2.0 - Technical White Paper.* http://www.wapforum.org/, Jan. 2002.

[64] *Compaq iPAQ Pocket PC.* http://h20022.www2.hp.com, 2002.

[65] J. Goodman, A. Chandrakasan, and A. Dancy, “Design and implementation of a scalable encryption processor with embedded variable dc/dc converter,” in *Proc. Design Automation Conf.*, pp. 855–860, June 1999.

[66] N. Potlapally, S. Ravi, A. Raghunathan, and G. Lakshminarayana, “Optimizing public-key encryption for wireless clients,” in *Proc. IEEE Int. Conf. Communications*, pp. 1050–1056, May 2002.

[67] *OpenSSL Project.* http://www.openssl.org.

[68] *Familiar Project.* http://familiar.handhelds.org.

[69] *National Instruments Corp.* http://www.ni.com.

[70] A. Menezes, P. V. Oorschot, and S. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.

[71] V. Gupta, S. Gupta, S. Chang, and D. Stebila, "Performance analysis of elliptic curve cryptography for SSL," in *Proc. ACM Wkshp. Wireless Security*, pp. 87–94, Sept. 2002.

[72] J. Daemen and V. Rijmen, "Rijndael, the advanced encryption standard," *Dr. Dobb's Journal*, pp. 137–139, Mar. 2001.

[73] *SSLdump Project.* http://www.rtfm.com/ssldump/.

[74] A. K. Lenstra and E. R. Verheul, "Selecting cryptographic key sizes," *J. Cryptology: The J. of the Int. Association for Cryptologic Research*, vol. 14, no. 4, pp. 255–293, 2001.

[75] *Counterpane Internet Security: Crypto-gram Newsletter.* http://www.counterpane.com/crypto-gram.html, 2004.

[76] *Xtensa application specific microprocessor solutions - Overview handbook.* Tensilica Inc. (http://www.tensilica.com), 2001.

[77] *ARCtangent^{TM}processor.* Arc International (http://www.arc.com).

[78] PICO Flex, Synfora Inc. (http://www.synfora.com).

[79] *ARM processors.* http://www.arm.com.

[80] *MIPS32^{TM}M4K^{TM}core.* MIPS Technologies (http://www.mips.com).

[81] *LISAtek solution.* http://www.coware.com.

[82] T. Blackwell, “Speeding up protocols for small messages,” in *Proc. Special Interest Group on Data Comm.*, pp. 85–95, 1996.

[83] W. Wulf and S. McKee, “Hitting the memory wall: Implications of the obvious,” *ACM SIGARCH Comp. Arch. News*, pp. 20–25, Mar. 1995.

[84] *Lightweight TCP/IP Stack.* http://savannah.nongnu.org/projects/lwip/, 2003.

[85] R. Braden, D. Borman and C. Partridge, *RFC1071 - Computing the Internet Checksum.* Network Working Group.

[86] R. B. Lee, “Subword parallelism with max-2,” *IEEE Micro*, vol. 16, no. 4, pp. 51–59, 1996.

[87] J. Daeman and V. Rijmen, “Rijndael: The advanced encryption standard,” *Dr. Dobb's Journal*, vol. 26, no. 9, pp. 137–139, 2001.

[88] M. Fiskiran and R. B. Lee, “Fast parallel table lookups to accelerate symmetric-key cryptography,”in *Proc. Intl. Symp. Information Technology*, pp. 526–531, Apr. 2005.

[89] M. Fiskiran and R. B. Lee, “On-Chip Lookup Tables for Fast Symmetric-Key Encryption,” in *Proc. Intl. Conf. Application-Specific Systems, Arch. and Processors,*, pp. 356–363, July 2005.

[90] W. van Eck, “Electromagnetic radiation from video display units: An eavesdropping risk,” *Computers and Security*, vol. 4, pp. 269–288, 1985.

[91] L. Benini, A. Macii, E. Macii, E. Omerbegovic, M. Poncino, and F. Pro, “A novel architecture for power maskable arithmetic units,” in *Proc. Great Lakes Symp. VLSI*, pp. 136–140, Apr. 2003.

[92] H. Saputra, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, R. Brooks, S. Kim, and W. Zhang, “Masking the energy behavior of DES encryption,” in *Proc. Design Automation & Test in Europe Conf.*, pp. 10084–10089, Mar. 2003.

[93] K. Tiri and I. Verbauwhede, "Securing encryption algorithms against DPA at the logic level: Next generation smart card technology," in *Proc. Cryptographic Hardware and Embedded Systems*, pp. 125–136, 2003.

[94] J. Viega, *Protecting Sensitive Data in Memory.* http://www-106.ibm.com/developerworks/security/library/, 2001.

[95] J. Whittaker and H. H. Thompson, "Security bugs exposed: A systematic approach to uncovering software vulnerabilities," *Software Testing and Quality Engineering*, pp. 28–32, Mar. 2003.

[96] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *Proc. USENIX Security Symp.*, pp. 321–336, Aug. 2004.

[97] Safenet Inc., *Better Hardware Security Modules Through Better Design.* http://www.safenetinc.com/library/8/HSM_Design_principles.pdf, 2002.

[98] L. Zhang and S. Malik, "The quest for efficient Boolean satisfiability solvers," in *Proc. Int. Conf. Computer-Aided Verif.*, pp. 17–36, July 2002.

[99] N. Een and N. Sorenson, "An extensible SAT solver," in *Proc. Int. Conf. Theory & Appl. Satisfiability Testing*, May 2003.

[100] S. Garfinkel and A. Shelat, "Remembrance of data passed," *IEEE Security and Privacy*, vol. 1, pp. 17–27, Feb. 2003.

[101] P. Broadwell, M. Harren, and N. Sastry, "Scrash: A system for generating secure crash information," in *Proc. USENIX Security Symp.*, Aug. 2003.

[102] V. Paretsky, *The Role of Hardware in Exposing Security Breaches.* http://www.ddj.com/print, 2005.

[103] C. Percival, *Cache Missing for Fun and Profit.* http://www.daemonology.net/papers/htt.pdf, 2005.

[104] R. J. Anderson and M. G. Kuhn, "Tamper resistance - A cautionary note," in *Proc. USENIX Wkshp. Electronic Commerce*, pp. 1–11, Nov. 1996.

[105] D. Farmer and W. Venema, *The Coroner's Toolkit.* http://www.porcupine.org/forensics/tct.html, 2005.

[106] J. Whittaker, "Why secure applications are difficult to write," *IEEE Security and Privacy*, vol. 1, pp. 81–83, Mar. 2003.

[107] *Valgrind Memory Profiler.* http://www.valgrind.org.

[108] National Institute of Standards and Technology, *Federal Information Processing Standard 46-3.* http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf, 1999.

[109] C. H. Li, "Integrating equivalency reasoning into davis-putnam procedure," in *Proc. 17th National Conference on Artificial Intelligence*, AAAI / MIT Press, 2000.

[110] B. Selman, D. G. Mitchell, and H. J. Levesque, "Generating hard satisfiability problems," *Artificial Intelligence*, vol. 81, no. 1-2, pp. 17–29, 1996.

[111] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. Intl. Conf. Arch. Support Prog. Lang. and Operating Sys. (ASPLOS)*, pp. 169–177, Nov. 2000.

[112] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. USENIX Operating Sys. Design and Impl. Symp.*, pp. 135–150, Oct. 2000.

[113] R. B. Lee, P. Kwan, J. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *Proc. Intl. Symp. Comp. Arch.*, pp. 2–13, May 2005.

[114] A. Lebeck and D. Wood, "Cache profiling and SPEC benchmarks: A case study," *IEEE Computer*, vol. 27, no. 10, pp. 15–26, 1994.

[115] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.

[116] R. Merkle, "A certified digital signature," in *Proc. Crypto '89*, pp. 218–238, Aug. 1989.

[117] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," in *Proc. Symp. Foundations of Comp. Science*, pp. 90–99, Oct. 1991.

[118] U. Maheshwari, R. Vingralek, and W. Shapiro, "How to build a trusted database system on untrusted storage," in *Proc. USENIX Operating Sys. Design and Impl. Symp.*, pp. 135–150, Oct. 2000.

[119] W. Shapiro and R. Vingralek, "How to build a trusted database system on untrusted storage," in *Proc. Digital Rights Management Wkshp.*, pp. 176–191, 2001.

[120] P. T. Devanbu and S. G. Stubbleline, "Stack and queue integrity on hostile platforms," *Software Engineering*, vol. 28, pp. 100–108, Jan. 2002.

[121] B. Chen and R. Morris, "Certifying program execution with secure processors," in *Proc. USENIX HotOS Wkshp.*, May 2003.

[122] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and Merkle trees for efficient memory authentication," in *Proc. Intl. Symp. High Perf. Comp. Arch.*, pp. 295–306, Feb. 2003.

[123] D. Williams and E. G. Sirer, "Optimal parameter selection for efficient memory integrity verification using merkle hash trees," in *Proc. Intl. Symp. Network Comp. and Appl.*, pp. 383–388, July 2004.

[124] D. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas, "Towards Constant Bandwidth Overhead Integrity Checking of Untrusted Data," in *Proc. IEEE SYmp. Security and Privacy*, pp. 139–153, May 2005.

[125] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," in *Proc. Crypto '94*, pp. 216–233, Aug. 1994.

[126] M. Fischlin, "Incremental cryptography and memory checkers," in *Proc. Eurocrypt '97*, pp. 393–408, May 1997.

[127] M. Bellare and D. Micciancio, "A new paradigm for collision-free hashing: Incrementality at reduced cost," in *Proc. EuroCrypt '97*, pp. 163–192, May 1997.

[128] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh, "Incremental multiset hash functions and their applications to memory integrity checking," in *Proc. AsiaCrypt '03*, pp. 188–207, Nov. 2003.

[129] B. Chevallier-Mames, D. Naccache, P. Pallier, and D. Pointcheval, "How to disembed a program?," in *Proc. Eurocrypt '04*, pp. 441–454, 2004.

[130] U. Vazirani, "Efficiency considerations in using slightly random sources," in *Proc. Symp. on Theory of Computing*, pp. 160–168, May 1987.

[131] R. Impagliazzo, *Pseudo-random generators for cryptography and for randomized algorithms.* PhD thesis, University of California, Berkeley, 1992.