Design Techniques for Side-channel Resistant Embedded Software

Ambuj Sudhir Sinha

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Patrick R. Schaumont, Chair
Michael S. Hsiao
Sandeep K. Shukla

August 5, 2011
Blacksburg, Virginia

Keywords: Side-channel Countermeasures, Secure Embedded Systems, Virtual Secure Circuit, Bitslice Cryptography, Side Channel Attacks

Copyright 2011, Ambuj Sudhir Sinha
Design Techniques for Side-channel Resistant Embedded Software

Ambuj Sudhir Sinha

ABSTRACT

Side Channel Attacks (SCA) are a class of passive attacks on cryptosystems that exploit implementation characteristics of the system. Currently, a lot of research is focussed towards developing countermeasures to side channel attacks. In this thesis, we address two challenges that are an inherent part of the efficient implementation of SCA countermeasures. While designing a system, design choices made for enhancing the efficiency or performance of the system can also affect the side channel security of the system. The first challenge is that the effect of different design choices on the side channel resistance of a system is currently not well understood. It is important to understand these effects in order to develop systems that are both secure and efficient. A second problem with incorporating SCA countermeasures is the increased design complexity. It is often difficult and time consuming to integrate an SCA countermeasure in a larger system.

In this thesis, we explore that above mentioned problems from the point of view of developing embedded software that is resistant to power based side channel attacks. Our first work is an evaluation of different software AES implementations, from the perspective of side channel resistance, that shows the effect of design choices on the security and performance of the implementation. Next we present work that identifies the problems that arise while designing software for a particular type of SCA resistant architecture - the Virtual Secure Circuit. We provide a solution in terms of a methodology that can be used for developing software for such a system - and also demonstrate that this methodology can be conveniently automated - leading to swifter and easier software development for side channel resistant designs.

This research was sponsored by the National Science Foundation, Award 0644070
Acknowledgments

First, I want to sincerely thank Dr. Patrick Schaumont, under whose guidance I have completed my graduate studies. It has been a privilege working with him, and I have grown under him not only in terms of technical knowledge, but also personally. I am extremely grateful for his patience with me over some tough times, and his faith in me as a student. His work ethics have always been an inspiration for me and will continue to be so.

My journey so far wouldn’t have been possible without my family. Their support is one of the strongest sources of strength in my life, and I simply wouldn’t be where I am today without them. Thank you Mom for always being there and thanks Papa for being the pillar of strength that you are. My sister, Sonal, deserves a special note of thanks for being my best friend all through my life.

Over the course of my graduate studies I have made many friends who have helped make life here at Blacksburg merrier. Thank you everyone - Karthik, Tyagi, Aditya, Yagnesh, Sridharan, Siddhesh, Srikrishna, Arijee, Abhranil, Kaushik, Jatin, Arijita, Danielle, Suvarna, KrishnaRaj, Sachin - and especially Nandini - you guys are awesome!

Also, thanks a lot to all my lab mates for always being extremely helpful. Special thanks to Zhimin Chen - for his guidance and everlasting patience while working with me answering all my queries. I also want to thank Abhranil Maiti, who has been an extremely reliable lab-senior and friend, and has been ever-helpful in all matters ranging from Latex formatting to emergency rides to the Airport.
Contents

1 Introduction ................................................. 1

1.1 Related Articles ............................................. 3

2 Preliminaries ..................................................... 4

2.1 Side Channel Attacks ........................................... 4
  2.1.1 MTD: Quantifying side-channel leakage ................. 6

2.2 Bitslice Computing .............................................. 8

2.3 The Advanced Encryption Standard Algorithm ................. 11
  2.3.1 Three Different Implementations of the AES algorithm .. 11
  2.3.2 AES S-Box and the AES T-box .......................... 12
  2.3.3 Bitsliced SBOX ........................................... 13

3 An Evaluation of AES Implementations in Embedded Software 15

3.1 Introduction .................................................... 15

3.2 Platforms and SCA Setup ...................................... 17
  3.2.1 Platforms ................................................ 17
3.2.2 SCA setup
3.3 Performance Analysis
3.4 Side-channel Analysis
  3.4.1 Power Model Selection for DPA
  3.4.2 SCA Results
  3.4.3 Location Analysis
3.5 Summary of Contributions

4 Programming the VSC
  4.1 Dual Rail Precharge (DRP)
  4.2 The Concept Of VSC
  4.3 Implementation
    4.3.1 Balanced Instructions
    4.3.2 Pre-charge Instructions
  4.4 VSC Programming
    4.4.1 n-way Bitslice Programming
    4.4.2 VSC Programming as an Extension of n-way Bitslicing

5 Automatic bitslicing/VSC conversion
  5.1 The AutoBS_lib Library
  5.2 Programming using the AutoBS_lib Library
  5.3 VSC-AES using AutoBS_lib
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An example of Side Channel Attack attacks a crypto-device and uncovers the key enclosed.</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of 256 correlation coefficient traces. Around time 100us, the black trace which corresponds to the correct key byte emerges from all the other 255 traces.</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Figure showing 2 ways of executing an AND instruction on a 4-bit processor. Figure(a) shows AND instruction in the conventional format. In figure(b) Bitsliced AND is shown. Input bits are first separated into different registers, then individually AND-ed. The different shades of bits indicate parallel instances of the same program.</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Dataflow in AES. A single T-tbox operation includes the shaded operations.</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Bitslicing of AES. (a) State-level bitslicing allocates each bit of an AES state to a different processor register, mapping 32 states in 128 registers. (b) Byte-level bitslicing allocates each bit of a byte to a different processor register, mapping 2 states in 8 registers.</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Embedded platforms: (a) PowerPC hardcore embedded in Virtex 2 Pro FPGA; (b) Leon3 softcore implemented in Spartan 3E-1600 FPGA.</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Side-channel attack system setup.</td>
<td>20</td>
</tr>
</tbody>
</table>
3.3 Differential Power Analysis: Power Model and Hypothesis Test .......................... 24

3.4 Correlation Coefficient plots for Side-channel Attack (number of measurements = 5120) on three different AES implementations on PPC platform. . . 27

3.5 Correlation Coefficient plots for Side-channel Attack (number of measurements = 5120) on three different AES implementations on Leon3 platform. . 28

3.6 (a) Power trace of S-box AES on PPC platform; (b) Correlation coefficient plots for Side-channel Attack on S-box AES on PPC platform. . . . . . . . . 30

3.7 (a) Power trace of Bit-sliced AES on PPC platform; (b) Correlation coefficient plots for Side-channel Attack on Bit-sliced AES on PPC platform. . . . . . . . . 31

3.8 (a) Power trace of T-box AES on PPC platform; (b) Correlation coefficient plots for Side-channel Attack on T-box AES on PPC platform. . . . . . . . . 32

4.1 (a) A CMOS standard NAND has data-dependent power dissipation; (b) A DRP NAND gate has a data-independent power dissipation. ....................... 36

4.2 (a) Concept of the proposed solution: balanced processor and VSC programming; (b) Power dissipation from the new balanced processor system does not reveal the processed data. ........................................ 37

4.3 (a) Regular INV can be used as a balanced INV; (b) Regular MOV can be used as a balanced MOV; (c) Balanced AND instruction uses half AND operators and half OR operators. ......................................................... 40

4.4 Performing a bit-wise left shift by 1 in the VSC domain using the regular shift instruction. First, a left shift by 2 is performed, so that the data in the direct and complementary part both get shifted into their respective datapaths. Then, the shifted in 0s are complemented in the complementary data path. . 41
4.5 2-way bitslicing of an 8-bit program on an 8-bit processor. Different colors indicate different instances of the same program running in parallel. (a) Showing how an AND instruction is executed in 2-way bitslicing. (b) Showing how a left shift instruction is executed in 2-way bitslicing.

4.6 Processing arrays A: \((a_0, a_1, a_3, a_4)\) and B: \((b_0, b_1, b_2, b_3)\) with AND operations to generate C: \((c_0, c_1, c_2, c_3)\). (a) regular programming; (b) bitslice programming; (c) VSC programming.

5.1 Design-flow: Generating VSC code using \texttt{AutoBS.lib}

5.2 Examples of generated VSC-C code using \texttt{AutoBS-lib}
List of Tables

3.1 Differences between two platforms ........................................... 19
3.2 Performance Summary ................................................................. 22
3.3 Attack Results Summary - Showing number of bytes discovered ........... 25
5.1 Memory footprint comparison between Normal-style AES and BS_VSC-style
   AES .......................................................... 59
Chapter 1

Introduction

Although cryptographic algorithms are designed to resist at least thousands of years of cryptanalysis, implementing them with either software or hardware usually leaks additional information which may enable the attackers to break the cryptographic systems within days. An attack that makes use of such side channel information (execution time, power dissipation, or/and magnetic radiation) is called a Side Channel Attack (SCA) [1]. Side channel attacks pose a very real and serious threat to the security of embedded systems, because of the short amount of time required to break the system, and the relative ease of mounting the attack. As such, developing countermeasures to prevent side channel attacks has gained a lot of interest from the research community, and several solutions have been developed.

However, implementing these countermeasures as part of a system often comes with its own set of challenges. Embedded systems generally need to satisfy constrained conditions of operation such as low area and power consumption, small software footprint, while having to maintain high performance. While designing such a system, a designer is faced with several choices. For example, a designer may need to choose between the technology used, the processor, the amount and type of memory, the type of software implementation etc. However, the many different combination of choices also affect the security of the system from the point of view of a side channel attack. For example, a particular type of software
implementation could be more robust against a side channel attack than another; or a particular type of technology used could leak more information. The effect of design choices on side channel resistance of the system are not well understood yet. In order to design systems that are both efficient and secure, it is important for the designer to understand the effects of design choices on the side channel behavior of the system.

Another challenge facing design of side channel resistant systems is the increased design complexity. Countermeasures to side channel attacks often impose certain constraints and requirements on the software or hardware that is being designed. Meeting these requirements is generally a tedious and complex task. Because of these constraints, it is difficult to integrate the countermeasures and test them as part of the system. Currently, there are no tools or formal methodologies available that help speed up this process. This is another area of side channel design that has not received much attention.

The contribution of this thesis are as follows. We explore that above mentioned problems from the point of view of developing embedded software that is resistant to power based side channel attacks. Our first work is a comprehensive analysis of different software implementations of the AES algorithm, a popular symmetric-key cipher, from the perspective of side channel resistance. We also show how design choices (both hardware and software) affect the security and performance of the different implementations. We show that such a comprehensive analysis provides a deeper understanding of the design space of embedded system security.

Next, we present work that identifies the problems that arise while designing software for a particular type of SCA resistant architecture - the Virtual Secure Circuit (VSC). We first discuss what are the requirements to develop correctly functioning software for the VSC processor. We then address the problems that arise when developing this software. We finally describe a formal technique that can be used to develop software for the VSC correctly. We also show that the methodology can be automated and a tool can be developed for easier development, integration and testing of software on this platform.
The following chapters of the thesis are structured as follows. Chapter 2 discusses related work in this area and also discusses some preliminary knowledge on embedded security. In Chapter 3, we present our work that shows the effect of design choices on side channel resistance for different implementations of AES in Embedded software. Chapter 4 establishes the programming concepts behind VSC programming, and identifies the problems and solutions in this regard. Chapter 5 describes our methodology and tool flow for automated VSC programming. Finally, Chapter 6 summarizes the contributions of our work and identifies potential future targets.

1.1 Related Articles

Our work is described in the following papers:


Chapter 2

Preliminaries

In this chapter we give a brief overview of side channel attacks, bitslice programming, and the AES algorithm.

2.1 Side Channel Attacks

SCA can break a cryptographic device much faster than cryptanalysis. The reason is that, by exploiting the side-channel leakage of a device, SCA can reveal the secret key bit by bit or piece by piece. This breaks the exponential complexity of a brute force key search. In this section, we describe the concept of SCA.

Figure 2.1 shows a cryptographic device. The device implements a cryptographic algorithm, represented by \( f \). \( f \) takes the plaintext (\( PT \)) and the key (\( K \)) as inputs, and generates the encryption result (\( ER \)) (\( ER = f(PT, K) \)). The internal secret key \( K \) is not directly observable through the ports of the device. The objective of SCA is to reveal the value of \( K \). Each bit of \( ER \) is related to every bit of \( PT \) and every bit of \( K \). Suppose \( K \) has 128 bits, then a traditional \textit{brute-force} attack needs to consider \( 2^{128} \) possible key values. It is this huge search space that ensures the security of the cryptographic algorithm. However,
Figure 2.1: An example of Side Channel Attack attacks a crypto-device and uncovers the key enclosed.

SCA does not try to break the entire $K$ at once. Instead, SCA divides $K$ into pieces, which are broken one by one. For example, a typical AES algorithm uses an 128-bit key. Using an SCA that reveals one key byte at a time, the search space for the entire key is reduced from $2^{128}$ to $16 \times 2^8 = 2^{12}$.

Obviously, this is a drastic reduction, and SCA achieves this as follows. Although every bit of the output $ER$ is guaranteed to be related to every bit of $K$, this is not true for the intermediate values. We can always find intermediate values that are only related to a small part of $K$, e.g. one byte of $K$. For example, assume that we can find an intermediate value $IV$ which depends on a single key byte $K[7:0]$ and the plaintext $PT$. We can write $IV = g(PT, K[7:0])$. Then $K[7:0]$ can be discovered with only $2^8$ guesses.
To test which guess is correct, we therefore need to observe $IV$. This variable is inside of the implementation, but it is indirectly observable through its power dissipation. Indeed, the power dissipated by $IV$ is a part of the power dissipated by the entire device, which can be measured by the attackers. Hence, by measurement of the chip power dissipation, there is a way to test the guesses the attacker makes on a single key byte. Through proper correlation techniques, the chip overall power dissipation can be used in place of the power dissipation from $IV$; the power dissipated by unrelated components can be treated as noise.

In recent years, very powerful SCA techniques have emerged. Modern secure embedded systems therefore need to apply adequate countermeasures to prevent SCA.

### 2.1.1 MTD: Quantifying side-channel leakage

Unlike the current or voltage, the side-channel leakage is not directly measurable. Therefore, researchers usually use indirect approaches to quantify the side-channel leakage, for example to use the attacking effort needed for successful attacks to reflect the amount of side-channel leakage. One of the commonly used approaches is called *Measurements to Disclosure* (MTD). The basic idea is that the more measurements that are required to successfully attack a cryptographic design, the more secure that design is. From a statistical point of view, the more measurements means the more samples. Since the side-channel analysis is basically a statistical process, more samples usually lead to more accurate result.

We use the same approach to quantify resistance of a system to SCA. In particular, we mount a *Correlation Power Attack* (CPA) [2] on the system. With the same amount of measurements, the less key bytes are discovered from a design, the more secure that design is. In the rest of this section, we discuss our method to quantify the side-channel leakage based on MTD and CPA.

Suppose we have a set of random plaintext blocks ($pt[1 \ldots n]$). Each of them is used for one measurement. Correspondingly, we obtain $n$ power traces, each of which contain $m$
Figure 2.2: An example of 256 correlation coefficient traces. Around time 100us, the black trace which corresponds to the correct key byte emerges from all the other 255 traces.

The largest gap means the black trace corresponds to the correct key.

Ambuj Sudhir Sinha

Chapter 2. Preliminaries

sampling points \((tr[1\ldots n][1\ldots m])\). In our experiment, the CPA takes \(pt\) and \(tr\) as inputs, and discovers AES’s key byte by byte by focusing on the first round of AES. The details are presented in Algorithm 1. The basic process is to take \(pt[1\ldots n]\) and a guess value of a key byte \((key\_guess)\) to calculate an intermediate value of the AES software \(iv[1\ldots n]\) \((iv[i] = f(key\_guess, pt[i])\)). Sampling a complete power trace guarantees that the operations on the intermediate value occur during the sampling process. We use the Hamming weight of \(iv\) \((iv\_hw[1\ldots n])\) to approximate the power dissipated by \(iv\). Then we calculate the correlation coefficient of \(iv\_hw\) and the actual measured power traces at each sampling point. As a result, we obtain a correlation coefficient trace \(corr[1\ldots m]\) \((corr[i]\) is the correlation coefficient of \(iv\_hw[1\ldots n]\) and \(tr[1\ldots n][i]\)). By now, we have obtained one coefficient trace \(corr[1\ldots m]\) which is corresponding to one guess value of the key byte. Since there are 256 possible values for one byte of key, we can, therefore, obtain 256 coefficient traces. By grouping these coefficient traces together, we are able to identify one coefficient trace from all the other 255 traces. In particular, at some points (when the operations on the intermediate value occur), the abstract value of the coefficient trace corresponding to the correct key
Algorithm 1 Correlation power attack on one key byte.

**Require:** $pt[1 \ldots n]$ contains the random plaintext for encryption; $tr[1 \ldots n][1 \ldots m]$ contains the sampled power traces; $f$ maps the inputs to the intermediate value $iv$; $g$ calculates the correlation coefficient.

**Ensure:** $key_gap[j]$ is the CPA-attacked key byte.

```plaintext
/* Obtain correlation coefficient traces corr */
for $key_guess = 0$ to $255$ do
    for $i = 1$ to $n$ do
        $iv[i] = f(key_guess, pt[i])$
        $iv.hw[i] = HammingWeight(iv[i])$
    end for
    for $i = 1$ to $m$ do
        $corr[key_guess][i] = g(iv.hw[1 \ldots n], tr[i][1 \ldots n])$
    end for
end for
/* Find the correct key byte */
for $i = 1$ to $m$ do
    find $|corr[key_1][i]| = max(|corr[0 \ldots 255][i]|)$
    find $|corr[key_2][i]| = second_max(|corr[0 \ldots 255][i]|)$
    $gap[i] = corr[key_1][i] - corr[key_2][i]$
    $key_gap[i] = key_1$
end for
find $gap[j] = max(gap[1 \ldots m])$
return $key_gap[j]$ as the correct key byte
```

guess is much larger than all the other traces. Figure 2.2 gives an example of the correlation coefficient traces. We can see that around time 100$\mu$s, one trace emerged from all the other traces. And it turned out that the key guess corresponding to this trace is the correct key, which means the CPA on this key byte is successful.

### 2.2 Bitslice Computing

Bitslicing is a programming technique that is gaining popularity as a method of designing high performance implementations of symmetric-key crypto-algorithms. The concept of bitslicing for symmetric key ciphers was first introduced by Biham et al. for the DES algorithm [3]. In bitslicing, an algorithm is broken down into bit-level computations (AND, OR, NOT,
XOR) and each of these bit-level operations is implemented as a machine instruction.

We can understand bitslicing with the help of the following example. Consider a 3-bit program that runs on an 4-bit processor. Let the program consist of a statement \( c = a \& b; \), where \( a, b \) and \( c \) are 3 bit variables. The contents of \( a \) and \( b \) are ANDed and stored in \( c \). Bitslicing works as follows. First, the contents of \( a \) are separated into individual bits. Each bit is stored separately in a different variable. Say, for example, bit 0 of \( a \) is stored in a variable \( a_0 \), bit 1 is stored in \( a_1 \), and so on. A similar operation is performed on the contents of \( b \), and the bits of \( b \) are stored in variables \( b_0, b_1 \) and so on. For a variable \( x \), this vector of variables containing the individual bits of the variable \( x \) is denoted as \( BS(x) \). Within this vector, we denote a variable containing the \( u \)th bit of \( x \) as \( BS(x, u) \).

Next, the individual bits are ANDed together by doing AND operations between the corresponding variables of \( BS(a) \) with \( BS(b) \), that is ANDing all \( BS(a, u) \) with \( BS(b, u) \). The results are stored in another variable vector \( BS(c) \). This is equivalent to the original program, \( c = a \& b; \). Therefore, instead of 1 AND instruction, 3 AND instructions are needed to generate one element of C array. However, we can also see that performing 1-bit operations on an 4-bit platform leads to under-utilization of the processor datapath. An AND instruction in a 4-bit processor contains 4 AND gates, and each AND instruction executes 4 1-bit ANDs in parallel. Hence, for each AND instruction, we have 4 parallel 1-bit programs. By making use of this inherent parallelism in the hardware, we can run 4 instances of the same bitsliced program at the same time. This scenario is shown in figure 2.3. Bitslice programming guarantees that the processors datapath is under full usage, which makes it more efficient.

The conversion of other logical operators to the bitsliced style of programming can be explained in the same manner as the AND instruction above. For example, OR, XOR and NOT in the bitsliced domain behave exactly like the AND instruction. The conversion of bit-wise shift operators to bitsliced domain can be explained as follows. In the normal style of programming, a bitwise shift on a variable corresponds to a change in the position of bits.
Program: c = a & b;

\[
\begin{align*}
\text{c} & \quad = \quad a & \quad \& \quad b \\
\text{c0} & \quad = \quad a0 & \quad \& \quad b0 \\
\text{c1} & \quad = \quad a1 & \quad \& \quad b1 \\
\text{c2} & \quad = \quad a2 & \quad \& \quad b2
\end{align*}
\]

(a)

(b)

Figure 2.3: Figure showing 2 ways of executing an AND instruction on a 4-bit processor. Figure(a) shows AND instruction in the conventional format. In figure(b) Bitsliced AND is shown. Input bits are first separated into different registers, then individually AND-ed. The different shades of bits indicate parallel instances of the same program.

within the variable. In the bitsliced domain, this translates to a re-assignment between the variables of a bitsliced vector. For example, say a variable x has been bitsliced into a bitslice vector BS(x). If we want to perform a left-shift by 1 bit on x, this translates to the value of BS(x,0) being copied into BS(x,1), the value of BS(x,1) being copied into BS(x,2), and so on. Hence, bitwise shifts translate into a set of copy operations in bitslice programming.

In principle, any complex function can be broken down into these logical operations. Hence, we can argue that once these basic operations are defined, we can implement any algorithm using bitslice programming.

Based on the above example, we define bitslice programming as follows. It is the conversion of a program written for an m-bit processor to a program for a 1-bit processor, then deploying m parallel instances of the 1-bit program on the m-bit processor (the m-bit processor behaves as m 1-bit virtual processors).

The suitability of an algorithm for bitslicing depends on the hardware-level complexity of
the algorithm. Essentially, bitslicing treats an algorithm as netlist of combinational logic, and simulates this netlist using processor instructions.

2.3 The Advanced Encryption Standard Algorithm

The Advanced Encryption Standard is well known, and detailed descriptions of the algorithm have been published [4]. Here, we briefly review the major steps in the algorithm. AES operates on a block of 128 bits at one time. It takes a block of 128 bits of plaintext and iterates that data through several rounds to produce ciphertext. Each round includes 4 transformations: SubBytes, ShiftRows, MixColumns, AddRoundKey. We refer to the literature for a description of these operations. AES organizes a block of 128 bit data as a 4x4 matrix of 16 bytes. Figure 2.4 illustrates the dataflow within a single AES round. The AES-128 algorithm, which is investigated in this thesis, includes 10 such rounds.

2.3.1 Three Different Implementations of the AES algorithm

In the following sections, we describe the three different implementations of AES that we have evaluated. We emphasize the dataflow of the algorithm, and we highlight specific optimizations for software.

The AES implementations under study differ primarily in the way they perform cryptographic substitutions. Such substitutions are part of the substitute-and-permute network (SPN) kernel within AES. A software implementation can implement these substitutions in different ways. The most straightforward way is to use a lookup table, called an S-box, which is a byte-wide 256-element table. By optimizing the AES algorithm for execution on 32-bit architectures, the S-box can be merged with some of the permute-steps of the SPN network, resulting in a so-called T-box [4]. The T-box is a word-wide 256-element table. Besides the S-box and T-box lookup-table based formulation of AES, we also study a bitsliced
implementation AES. In bitslicing, the individual bits of a word are computed separately, using bit-level operations [3]. In this case, the lookup tables need to be decomposed into elementary operations; this works for AES S-box because the input and output of the S-box are mathematically related. Indeed, the substitution can be expressed using finite field arithmetic operations [5].

### 2.3.2 AES S-Box and the AES T-box

Each of the $s$-operations in Figure 2.4 is a 256-entry substitution table. This substitution is mathematically defined: an S-box operation on an input byte $a$ is found as a finite-field inversion (over $GF(2^8)$) followed by an affine transformation [4]. In each AES-128 round, there are 16 S-box substitutions. Such a formulation of AES leads to byte-wide computations over the entire round, which is inefficient on a 32-bit processor.
The S-box is therefore frequently implemented as a T-box, which includes each of the shaded operations in Figure 2.4. A T-box transforms one byte into four bytes, each of which represent part of a MixColumn result. In each AES round, four T-box operations can be combined to obtain a single row of the AES state matrix. Due to the detailed implementation of MixColumn, this approach requires four different T-box lookup tables. However, this approach provides a significantly better utilization of the 32-bit datapath of a processor.

2.3.3 Bitsliced SBOX

Bitslicing of AES has been discussed extensively in literature [3, 5–9]. There are two motivations for it: bitslicing improves performances, and it provides timing based side-channel resistance [10]. The latter property is achieved when lookup tables (memory-load operations) are replaced with computations (logical operations). In that case, data-dependent cache-behavior is avoided, resulting in AES constant-time execution. For our work however, we are investigating power-based side-channel resistance, which is not prevented by bitslicing.

Figure 2.5 illustrates two approaches to bitslicing of the AES state. In state-level bitslicing, each bit of the AES state is mapped into a different register [6]. 32 AES states can then be represented in 128 processor registers. This approach suffers from three drawbacks. First, it requires a large amount of processor registers, and may result in spilling, which reduces efficiency. Second, when less then 32 AES blocks need to be encrypted or decrypted, this method results in suboptimal processor utilization [11]. Finally, conversion of data blocks into bitsliced format and back requires additional overhead. However, the method results in an encryption speed of 9.2 cycles per byte on a Core2 processor [8].

Byte-level bitslicing, illustrated in Figure 2.5b, takes a more compact approach. The corresponding bits of each byte in two AES states (32 bytes in total) are mapped into the bits of a register [7]. Eight 32-bit processor registers thus can capture two entire AES states. In contrast to the previous method, this method needs a specific bit-level organization in order
Figure 2.5: Bitslicing of AES. (a) State-level bitslicing allocates each bit of an AES state to a different processor register, mapping 32 states in 128 registers. (b) Byte-level bitslicing allocates each bit of a byte to a different processor register, mapping 2 states in 8 registers.

to obtain an efficient implementation of the AES permutation operations (such as ShiftRows and MixColumns). Nevertheless, on a Core2 processor, a byte-level bitsliced AES was shown to execute at 7.59 cycles/byte [7]. For our experiments in Chapter 3, we implemented a byte-level bitsliced version of AES.
Chapter 3

An Evaluation of AES
Implementations in Embedded Software

In this chapter we present work that establishes the relation between different design parameters and the side channel resistance of the system. For this work we have used the Advanced Encryption Standard (AES) algorithm as the driver application for our experiments.

3.1 Introduction

The Advanced Encryption Standard, which was selected by the National Institute of Standards and Technology in 2001, is the dominant symmetric-key algorithm in use today. The algorithm has been intensively studied, and a large amount of results with respect to performance and security have been published. However, often it is unclear how these different optimizations behave when compared to one another. For example, optimizing for performance may have an adverse effect on security, and vice versa. In this chapter, we compare
three different implementation techniques for the software implementation of AES, that we have introduced earlier, and we compare them under two orthogonal concerns: performance and (power-based) side-channel resistance. We show that such a comprehensive analysis provides a deeper understanding of the design space of embedded system security.

Our experiments are targeted to two 32-bit processor architectures, the LEON-3 (soft-core) processor [12], and the PowerPC 405 processor [13]. These 32-bit architectures are representative for typical embedded systems.

Given these two processor platforms, and three different software implementations of AES, we analyze these designs from two perspectives: performance and power-based side-channel leakage. Performance is evaluated by measuring the execution time of the AES kernel. Power-based side-channel leakage is evaluated using differential power analysis (DPA) [1]. These experiments are performed on a side-channel analysis setup consisting of FPGA prototyping boards, a sampling oscilloscope, and post-processing software.

The main contributions of this work are as follows. First, we show that bit-slicing on 32-bit platforms does not outperform a T-box implementation of AES. While it is assumed that bit-slicing requires a wide processor wordlength to be effective [6], this was never demonstrated. In contrast, on high-end processors with wide execution units (64-bit or higher), bit-slicing is known to provide superior AES performance [7]. Second, we also demonstrate that memory-based table-lookups have smaller power-based side-channel leakage than the equivalent bit-sliced implementation. We attribute this to the fact that a single-step memory access performs the same amount of work that requires many different instructions for the bit-sliced implementation. Third, we describe several practical techniques to analyze the side-channel resistance of software implementations. We explain how to discriminate the correct secret key from the side-channel measurements. In addition, we present a technique to identify the most sensitive part of a software implementation.

In our evaluation, we did not consider side-channel resistant designs, such as masking or hiding [14]. Initially, we wanted to compare the algorithms by themselves, since that gave
already an interesting design space. The exploration of side-channel resistant designs is a useful extension of this work, but it is outside its current scope.

The remainder of this chapter is organized as follows. In Section 3.2, we describe the platforms that we have used for our experiments, and we present performance results for our three AES implementations in Section 3.3. Section 3.4 presents an in-depth discussion of the power-based side-channel analysis of these designs. We summarize the contributions of this work in Section 3.5.

3.2 Platforms and SCA Setup

In this section, we describe two embedded platforms on which the 3 different AES implementations are built. After that, we discuss the side-channel analysis setup on these two platforms.

3.2.1 Platforms

We chose two different platforms. The first platform is the SASEBO-G board [15], which contains a Virtex 2 pro FPGA (XC2VP30-5FG676C) with two embedded ASIC 32-bit PowerPC 405 RISC processor blocks inside. AES software is implemented on one PowerPC processor. In the rest of this thesis, we use PPC to represent this platform. Figure 3.1a illustrates the first platform.

The second platform is the Digilent Spartan 3E-1600 Development Board [16] with a Spartan 3E-1600 FPGA (S3E1600). The AES software runs on a 32-bit Leon3 SPARC V8 processor [12] built with the FPGA resources. We use LEON3 to represent this platform. Figure 3.1b gives an overview of the second platform.

The above two platforms are different in several major features. The PPC platform supports both the software’s instructions and data to be stored in on-chip memories, as shown in
Figure 3.1: Embedded platforms: (a) PowerPC hardcore embedded in Virtex 2 Pro FPGA; (b) Leon3 softcore implemented in Spartan 3E-1600 FPGA.

Figure 3.1a. The PowerPC processor and the on-chip memory are clocked at the same speed of 24 Mhz. Both instructions and data can be read or written in one clock cycle. In this case, there is no need to implement a cache. From an architectural perspective, the PPC platform enables the software to run at a fastest speed. In addition, the execution time of each AES implementation can be a constant.

The LEON3 platform uses the FPGA resources to implement the Leon3 processor, clocked at 50 Mhz. The instructions and data of the software are stored in an external DDR2 SRAM memory. Due to the lower speed of the external memory, both instruction cache and data cache are implemented within the FPGA. Figure 3.1b depicts the architecture of the second platform. Due to the cache effects, execution time of each AES implementation is data-dependent. Table 3.1 summarizes the differences of these two platforms.

The AES implementations follow the same design flow in both platforms. All 3 AES implementations are designed as 3 functions and are integrated to the testing program (Algorithm 2), shown in Section 3.2.2. The S-box and T-box based implementations are both
Table 3.1: Differences between two platforms

<table>
<thead>
<tr>
<th>PPC</th>
<th>LEON3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC processor</td>
<td>FPGA</td>
</tr>
<tr>
<td>NO cache</td>
<td>YES</td>
</tr>
<tr>
<td>on-chip code location</td>
<td>off-chip</td>
</tr>
<tr>
<td>on-chip data location</td>
<td>off-chip</td>
</tr>
<tr>
<td>constant exec. time</td>
<td>data-dependent</td>
</tr>
</tbody>
</table>

designed in C, while the bit-sliced implementation is written in Assembly. The pure-C designs can be simply compiled with ‘powerpc-eabi-gcc’ and ‘sparc-elf-gcc’. When compiling the Bit-sliced AES, we link the hand-written assembly to the object file and obtain the executable. At any of the above stages, the compilers’ optimization level is set to 2.

3.2.2 SCA setup

The side-channel analysis setup is the same for both platforms. Hence, we only use PPC platform as an example for explanation. The setup contains the PPC platform, an oscilloscope (Agilent DSO5032A) and a PC, shown in Figure 3.2.

The three parts of the setup are connected in a circular fashion. A RS232 cable connects the PPC platform and the PC. Between the oscilloscope and the PC is a USB cable, through which the PC is able to send commands to and get sampling waveform from the oscilloscope. The oscilloscope uses a current probe (Tektronix CT-2) to monitor the current flowing into the PPC platform. The sampling rate and precision are set to be the same for all 3 different AES implementations. We use the current to represent the power consumption of the embedded system. Side-channel analysis requires a number of measurements with different inputs (plaintexts for AES). In our experiment, the result of one measurement is the cur-
Figure 3.2: Side-channel attack system setup.
rent trace of the PPC platform and the corresponding random plaintext block for encryption. Each measurement consists of the following 4 steps. A side-channel analysis that requires $n$ measurements needs to repeat these 4 steps for $n$ times.

- Step 1: The PC sends a random plaintext block (16 bytes) to the embedded platform through the RS232 cable.

- Step 2: The PowerPC processor in the platform receives the plaintext and encrypts it with the AES software. The program running on the PowerPC processor is shown in Algorithm 2. $Cipher$ is one of the 3 AES implementations.

- Step 3: After sending out one block of plaintext, the PC sends command to the oscilloscope to sample the current trace when PowerPC is running the encryption.

- Step 4: After sampling is done, one current trace is sent back to PC for side-channel analysis.

**Algorithm 2** Program running on PowerPC

\[
\begin{align*}
exp\_key & \Leftarrow \text{KeyExpansion}(key) \\
\text{if } Cipher \text{ is bit-sliced AES then} & \\
\quad exp\_key & \Leftarrow \text{BitsliceConversion}(exp\_key) \\
\text{end if} \\
\text{loop} & \\
\quad \text{if UART is not empty then} & \\
\quad \quad \text{wait until 16 bytes received} & \\
\quad \quad in[15 : 0] & \Leftarrow \text{UART\_FIFO} & \\
\quad \text{end if} & \\
\quad out[15 : 0] & \Leftarrow Cipher(in, exp\_key) & \\
\text{end loop} & \\
\end{align*}
\]

### 3.3 Performance Analysis

In this section we present a performance based comparison between the three different AES implementations.
Table 3.2: Performance Summary

<table>
<thead>
<tr>
<th></th>
<th>PPC (24MHz)</th>
<th>Leon3 (50MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-box</td>
<td>T-box</td>
</tr>
<tr>
<td>latency (µs)</td>
<td>310</td>
<td>60</td>
</tr>
<tr>
<td>throughput (kbits/s)</td>
<td>412</td>
<td>2133</td>
</tr>
<tr>
<td>footprint (kB)</td>
<td>1.9</td>
<td>5.2</td>
</tr>
</tbody>
</table>

Table 3.2 compares the three implementations based on latency, throughput, and memory footprint. In terms of speed, T-box AES is by far the best implementation on both the platforms. It has a slightly large memory footprint because of the large lookup table that it uses (4 kB).

The performance of bit-sliced AES is second best on the PowerPC. On a 32-bit processor, the bit-sliced implementation processes two AES blocks at once. Hence, the effective execution time for encryption is half the latency. Bit-sliced AES incurs a large code footprint on both the platforms. This is because the different functions like SubBytes and ShiftRows, which are simple loads and stores from memory in a standard implementation, are expanded into a much larger set of instructions in the bit-sliced domain. It can also be seen that the performance of the bit-sliced implementation is comparatively worse on the Leon3 than on the PowerPC. This can possibly be attributed to the large code footprint of bit-sliced AES. Because of large code size, instructions are not present in the instruction cache, and have to be fetched frequently from main memory. This is not a problem on the PowerPC, because all the code resides on the on-chip memory.

The standard S-box implementation has the lowest throughput of the three implementations. However, it has a smaller code footprint as compared to the other two implementations.

It can be seen from these results that, in terms of performance, T-box AES is one of the best choices for implementation on embedded cores. Though bit-sliced AES has made speed records on other platforms [7], it may not be the best option to use on 32-bit embedded processors. As argued by Matsui [6], bit-sliced implementations can offer an advantage when the following conditions are met: the target processor has a large number of registers,
and the register length of the target processor is long. These conditions may not always be available on embedded platforms. In addition, bit-slicing benefits greatly from more complex instructions present on high end general purpose cores (e.g., SSE3 and SSE4 ISE [17]). These instructions are generally not present on embedded platforms.

### 3.4 Side-channel Analysis

In this section, we describe our experiments to evaluate and analyze the side-channel leakage of the three AES implementations under consideration. We are using differential power analysis (DPA) for our side-channel analysis. In this technique, power measurements are compared with a hypothetical power model. We discuss two important aspects in performing a practical DPA. The first is the selection of the power model, and the second is the definition of the attack success metric (measurements to disclosure, MTD). Next, we present the DPA results for the three AES implementation. Finally, we perform a refined analysis, in order to precisely locate the source of side-channel leakage for the chosen power model.

#### 3.4.1 Power Model Selection for DPA

The objective of DPA is to compare measurements, obtained from a prototype, with power estimates, generated using a power model. The power model is chosen so that it has a dependency on a part of the secret key. For example, in the first round of AES encryption, the output of the substitution step is given as follows.

\[
out[i] = \text{subbytes}(in[i] \oplus key[i]);
\]  

(3.1)

where \(in[i]\) is the \(i\)-th byte of the 128-bit plaintext, \(key[i]\) is the \(i\)-th byte of the 128-bit key, and \(out[i]\) is the \(i\)-th byte of the 128-bit AES state. In this formula, \(in\) is known, while \(key\) and \(out\) are unknown. However, \(out\) is indirectly observable through the power consumption.
of the algorithm. Hence, we can create a hypothesis test for a key byte, using a power model for \( \text{out} \). By making a guess for the \( i \)-th key byte, we can infer the value of \( \text{out[i]} \). The power model used in the DPA is an estimate for the power consumption of the hypothesized \( \text{out[i]} \). The actual hypothesis test, explained in the next section, will compare the estimated power consumption with the measured power consumption to identify the most likely key byte hypothesis. In this section, we discuss the selection of the power model.

In CMOS technology, where power is consumed as a result of state transitions, it is common to choose the Hamming Distance (the toggle count) as a power model [14]. For example, when performing power analysis on a processor, we could use the transitions of a processor register. However, in a software implementation of AES, we do not know what other data will share the same register location as \( \text{out[i]} \). As a result, the Hamming Distance model is of limited use. Instead, we use the Hamming Weight (the bit count) as the power model.

A second difficulty in selecting the power model is that a software implementation of AES
Table 3.3: Attack Results Summary - Showing number of bytes discovered

<table>
<thead>
<tr>
<th>Measurements</th>
<th>PPC</th>
<th>Leon3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bitsliced</td>
<td>S-box</td>
</tr>
<tr>
<td>2048</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>5120</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>10240</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>25600</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>40960</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>51200</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

typically may take many instructions. As a result, the power trace of an AES encryption may cover several hundred clock cycles. We cannot predict, in general, at which clock cycle $out[i]$ will be computed. To address this issue, we simply extend our power model over all the clock cycles of a power trace, by repeatedly using the Hamming Weight of $out[i]$ for every clock cycle that we wish to analyze.

Finally, when attacking an unknown AES implementation, the specific variation of the algorithm that is executing is unknown to an attacker. In this work, we compare three common implementations of AES (S-box based, T-box based, and bit-sliced). The S-box based and bit-sliced implementation produce the intermediate value $out[i]$ as specified above, and therefore the Hamming Weight of $out[i]$ is a valid power model. The T-box based implementation, on the other hand, does not have such an intermediate value, because it collapses the S-box substitution with several other AES operations. For a T-box, a more appropriate power model is one that is based on a T-box lookup table. This raises the obvious question how an attacker can select the power model. In practice, this is a minor issue. An attacker can simply try out multiple power models, and select the one for which the strongest hypothesis conclusions can be made (Section 5.2). Even 8 possible implementations for AES would not add more than 3 bit of uncertainty, and require 8 times more hypothesis tests. Therefore, in our tests, we used the most appropriate power consumption model for each implementation.
3.4.2 SCA Results

Table 3.3 presents a comparison between three AES implementations in terms of resistance to side channel attack.

An attack was mounted on all the three implementations of AES on the two platforms (PPC and Leon3). The attacks were based on the techniques described previously.

It can be clearly observed from the table that the bit-sliced implementation is the weakest in terms of side channel resistance. Using the S-box implementation as a reference, the bit-sliced version requires 5-8 times fewer power traces to find all bytes of the key. The T-box implementation is strongest, requiring maximum number of traces to break the key. This is expected, because the T-Box implementation combines SubBytes, ShiftRows and part of MixColumns into one highly compressed lookup table. This means that sensitive data, that can be attacked using a power analysis attack, is present in the system for a very short time as compared to the other two implementations. Figures 3.4 and 3.5 display correlation coefficient plots for the three implementations on both the platforms. It can be seen that the correlation peak for bit-sliced AES is higher than those for S-box and T-box based implementations.

A significant fact can be observed from the data in table 3.3. Among the two platforms, implementations on the Leon3 are much more easier to attack as compared to those on the PPC. A standard S-box implementation required 40960 power traces to break completely on the PPC, while this number for the Leon3 was 6400 (an increase by a factor of 6.4). Similar trend was observable for the other implementations as well. The same observation can be made by comparing the correlation coefficient plots for the different AES implementations on the two platforms. For the same number of measurements, the correlation peak for implementations on the Leon3 are larger than their counterparts for PPC. This shows that FPGA based soft-cores leak much more sensitive information as compared to ASIC cores.
Figure 3.4: Correlation Coefficient plots for Side-channel Attack (number of measurements = 5120) on three different AES implementations on PPC platform.
Figure 3.5: Correlation Coefficient plots for Side-channel Attack (number of measurements = 5120) on three different AES implementations on Leon3 platform.
The Effect of Cache

It can be seen from table 3.3 that the T-Box implementation cannot be broken on the Leon3. The correlation coefficient plot for T-box AES on Leon3 (figure 3.5) also does not show any clear peak which can be identified as the correct key. The reason for this is as follows. While mounting an attack on the T-Box implementation on the Leon3, our first observation was that the waveform was highly unstable and noisy, even when the beginning of every encryption was aligned. We tried to attack this particular waveform but were unsuccessful. We found that the cache in the Leon3 was causing such behavior. Because of the large lookup tables used by the T-Box, the effect of cache becomes noticeable, and the execution time for different encryptions differs depending on the plaintext. We verified this by disabling the cache on the Leon3 and doing the attack again. With the cache turned off, a very stable waveform was observed with constant execution time, and the T-box implementation could be easily broken.

This particular effect is absent in the PPC because it does not have cache and uses on-chip memory instead.

3.4.3 Location Analysis

Finally, we describe a technique that helps to identify which part of the algorithm in software is most sensitive to a power analysis attack.

We first identify which part of the algorithm corresponds to which part of the power trace. This can be done by inserting a series of ‘nop’ instructions at specific points in the program. The ‘nop’ instructions have a power profile which is easily distinguishable on the power trace. Thus, by inserting ‘nops’ at appropriate places in the program, we can separate different parts of the algorithm from one another in the power trace. Next, we perform an attack on the algorithm with these ‘nop’ instructions present in the program. The sensitive points in the algorithm can now be identified by seeing where in the power trace a correlation
Figure 3.6: (a) Power trace of S-box AES on PPC platform; (b) Correlation coefficient plots for Side-channel Attack on S-box AES on PPC platform.
Figure 3.7: (a) Power trace of Bit-sliced AES on PPC platform; (b) Correlation coefficient plots for Side-channel Attack on Bit-sliced AES on PPC platform.
Figure 3.8: (a) Power trace of T-box AES on PPC platform; (b) Correlation coefficient plots for Side-channel Attack on T-box AES on PPC platform.
peak occurs.

For the S-box implementation, the correlation peak was found to be occurring around the ShiftRows function of the first round. Figure 3.6 shows the correlation plot along with the power trace for the S-box implementation on the PPC. The ShiftRows function has been isolated from other parts of the algorithm by inserting a set of ‘nops’ before and after it. The ‘nops’ are easily distinguishable in the power trace as large drops in the power. The peak between the two drops corresponds to the ShiftRows function. The correlation peak can be clearly seen to be occurring over this part of the waveform. Figure 3.7 shows a similar plot for bit-sliced AES. Here too, the correlation peak occurs around the ShiftRows function. The sensitivity of ShiftRows is expected, as it operates directly on the sensitive data. The T-box implementation shows sensitivity around the first round, as shown in Figure 3.8. It is difficult to localize the correlation peak from the T-box implementation any further, as each round is compressed into a memory lookup and a set of XOR operations.

The plots presented in Figures 3.6, 3.7 and 3.8 are for attacks using small number of measurements (768 for bit-sliced AES and 5120 for S-box and T-box AES). Hence, the sensitive locations that we have pinpointed for the different implementations correspond to those parts of the algorithm that break the earliest under an attack. For an attack using a large number of measurements, correlation peaks will often be spread out over a large area of the power trace, and not be localized to just one function or part of the algorithm.

3.5 Summary of Contributions

In this chapter, we compared three different implementations of AES from the viewpoint of performance and power-based side-channel leakage. We concluded that large variations can be found in terms of performance as well as in terms of security. We have also shown that there are many factors that affect performance and side-channel resistance.
Chapter 4

Programming the VSC

The Virtual Secure Circuit is an architecture developed specifically to protect embedded software from side channel attacks. It achieves this by mimicking a hardware-based SCA countermeasure called the Dual-Rail Precharge (DRP) [18] technique by making small modifications to the instruction set of the processor and doing a special type of programming using these modified instructions. VSC emulates behavior of a hardware circuit using software, hence the name Virtual Secure Circuit. VSC improves the side channel resistance of embedded software by over 20 times. Work on the VSC has been described in [19].

However, as we will see later, writing a program for the VSC platform keeping these special conditions in mind is a difficult task. In this chapter, we present a solution to this programming problem based on the concepts of bitslice programming discussed earlier. The following sections are structured as follows. First, some background information on DRP and the VSC architecture is provided. Next, the instruction set modifications are discussed, and it is shown how to program the VSC using these instructions. Finally, a methodology for programming the VSC is presented based on bitslice programming.
4.1 Dual Rail Precharge (DRP)

VSC inherits its security properties from the DRP technique, which is an effective countermeasure against SCA for hardware platforms. Its basis is to reduce the side-channel information from the power dissipated by the intermediate variable $IV$ (Section 2.1). Hardware circuits implemented as DRP have a constant power dissipation. We use an example to explain the details.

Figure 4.1 explains the operation of the DRP technique using a DRP NAND gate. In this example, we approximate static and dynamic power dissipation of a logic gate through the Hamming Weight and Hamming Distance of its output respectively. In the case of a single NAND gate (Figure 4.1a), the static and dynamic power dissipation depend on the input values of the gate. For example, if the static power is 0, both inputs must be 1. This side-channel leakage is the basis for SCA.

Figure 4.1b shows the same test case on a DRP NAND gate. In this case, the circuit encodes each logic value with a complementary pair ($A_p, \overline{A_p}$). Furthermore, each pair is pre-charged to (0,0) in each clock cycle before evaluation. As a result, each clock cycle, every DRP signal pair shows exactly one transition from 0 to 1 and another one from 1 to 0. The resulting static and dynamic power dissipation are now independent of the input values of the DRP gate.

Despite the elegance of this concept, DRP circuits in hardware do have some disadvantages. First, DRP circuits are at least two times larger than equivalent standard CMOS circuits, and they have a much larger power dissipation. Second, the constant-power argument, based on Hamming Weight or Hamming Distance, does not hold when low-level electrical effects are taken into account. Small asymmetries between the direct and complementary paths of a signal pair still may lead to residual side-channel leakage. Nevertheless, careful design is able to reduce the imbalance to a very low level. This requires much more power measurements for a successful SCA [20].
Figure 4.1: (a) A CMOS standard NAND has data-dependent power dissipation; (b) A DRP NAND gate has a data-independent power dissipation.
Figure 4.2: (a) Concept of the proposed solution: balanced processor and VSC programming; (b) Power dissipation from the new balanced processor system does not reveal the processed data.

So far, the DRP technique has been broadly used in hardware as secure circuits, for example in SABL [21], in WDDL [22], and in MDPL [23]. However, software DRP technique has not been developed. The major reason for this is that DRP technique requires the executions of the direct and complementary datapaths in parallel. In regular processors, this cannot be realized.

4.2 The Concept Of VSC

The concept is illustrated in Figure 4.2. Similar to the DRP circuits, the processor has two parts: one performs direct operations and the other performs complementary operations. We call the processor a balanced processor. Every direct operation in the first part has a complementary counterpart in the second part. A direct operation takes input \(\text{in}\) and generates output \(\text{out}\). The complementary operation takes input \(\overline{\text{in}}\) and generates output \(\overline{\text{out}}\). \(\overline{\text{in}} = \text{NOT}(\text{in})\) and \(\overline{\text{out}} = \text{NOT}(\text{out})\). \text{NOT} means the bit-wise inversion operation.
The balanced processor has a set of instructions, referred to as balanced instructions. Each balanced instruction chooses a pair of complementary operations from two parts of the processor and executes them at the same time. The cryptographic algorithm is programmed with this set of balanced instructions. Therefore, the cryptographic algorithm has both a direct execution path and a complementary path. To run the cryptographic algorithm, both direct and complementary plaintext (balanced input) are supplied to these two paths. Finally, the algorithm generates both the direct and the complementary encryption results (balanced outputs), shown in Figure 4.2a.

Besides the balanced instructions, the processor also has another set of instructions that perform the pre-charge operations in the same way as the DRP circuits. Before executing balanced instructions, a set of pre-charge instructions first clear the execution datapath. Following that, the balanced instruction finishes the calculation.

With the above concept, we obtain a software version of DRP circuit. For a balanced instruction, similar to the DRP gate shown in Figure 4.1, the power dissipation from the direct operation always has a complementary counterpart from the complementary operation. The sum of these two is a constant, shown in Figure 4.2b. Suppose the output of the balanced instructions contains the direct and complementary forms of the intermediate value $IV$ mentioned in Section 2.1, attackers are not able to go from the power of the balanced instructions to the value of $IV$. Therefore, $IV$ is protected. If every intermediate value in the cryptographic algorithm is processed by the balanced instructions, then the entire algorithm is secured. This kind of software program is the VSC.

4.3 Implementation

The previous section describes the concept of VSC. The next step is to map the concept to the processor implementation. This includes the implementation of the balanced instructions, the pre-charge instructions and the secure programming style that leads to DRP behavior.
4.3.1 Balanced Instructions

The instruction set architecture of a processor includes different instructions. The logic function of the software is realized with logic instructions, such as AND, OR, and NOT. In theory, any algorithm can be realized with these three instructions. For higher efficiency, processors also implement the arithmetic instructions that are frequently used, such as ADD, SUBTRACT, and MULTIPLY. In addition to that, due to the processor architecture, shift instructions and data instructions, such as MOVE, LOAD, and STORE, also appear in the instruction set architecture. Finally, the processor also provides the control instructions, such as JUMP and conditional JUMP.

As mentioned above, each balanced instruction executes a pair of complementary operations. To make the balanced datapath compatible with the regular datapath, the width of either the direct or the complementary datapaths is designed to be half of the regular datapath’s width. Thus, the entire datapath of a balanced instruction is as wide as the regular instructions.

In order to achieve security against EM attacks, the direct and complementary datapaths are interleaved with each other (Figure 4.3).

After fixing the width of the balanced instructions, the next step is to fix the functions. Each balanced instruction must satisfy the condition that if complementary inputs are fed to the corresponding direct and complementary paths of the instruction, the output should also be complementary in its direct and complementary data-paths. Among the logic operations, NOT is the complementary instruction for itself. This means that if half of the input is direct value and the other half is complementary value, the output of NOT is still a pair of complementary values. Therefore, the regular NOT instruction can also be used as the balanced NOT instruction. The AND operation has OR operation as complementary counterpart and vice versa. Therefore, the balanced AND instruction should be half AND for the direct input and half OR for the complementary input. Similarly, the balanced OR instruction should be half OR and half AND.
Figure 4.3: (a) Regular INV can be used as a balanced INV; (b) Regular MOV can be used as a balanced MOV; (c) Balanced AND instruction uses half AND operators and half OR operators.
Figure 4.4: Performing a bit-wise left shift by 1 in the VSC domain using the regular shift instruction. First, a left shift by 2 is performed, so that the data in the direct and complementary part both get shifted into their respective datapaths. Then, the shifted in 0s are complemented in the complementary data path.

Data instructions are similar. They both move the programs’ intermediate values from one storage location to another. Since ‘move’ operations do not change the values of the operands, their complementary counterparts are themselves. Similar to NOT instruction, data instructions are shared by both balanced and regular instructions.

Shift instructions are similar to data instructions, that is they move values from one storage location to another. Hence, regular shift instructions can also be used as their balanced counterparts. However, because of the way shift instructions are implemented, certain conditions need to be kept in mind in order to ensure that the outputs of these instructions are balanced. This can be explained with the help of the following example. Say a left-shift by 1 bit is performed on a variable in the direct data-path. This would imply that a 0 would be shifted into the right-most bit of the variable. However, for the complementary data-path, a 1 should be shifted into the right-most bit, which does not happen automatically. After performing a shift operation, the user must ensure that the shifted-in 0’s in the direct path...
should be complemented by corresponding 1s in the complementary datapath. This is shown in figure 4.4. Secondly, shift instructions can push the data from the complementary datapath into the direct datapath and vice versa, which may be undesirable. The programmer must take care this doesn’t happen.

Currently, there is no easy and efficient way to design balanced arithmetic and control instructions. But fortunately, a number of cryptographic algorithms can avoid using these instructions. For example, the AES algorithm has a fixed control flow. So arguments of the control instructions are not functions of the key or the plain text input. It does not influence the security even if the attackers know these intermediate values. Another possible question is that, without balanced arithmetic instructions, whether it is practical to realize the software only with balanced logic instructions. Normally, using logic instructions to replace arithmetic instructions is very inefficient. However, for a number of cryptographic algorithms, only logic instructions are needed. Therefore, even though we do not have balanced arithmetic instructions, these cryptography can still be implemented without decreasing the efficiency too much.

4.3.2 Pre-charge Instructions

As mentioned earlier, besides complementary datapath, DRP technique also requires pre-charge operations. The proposed solution also faces the problem on how to implement pre-charge instructions for the same purpose. A pre-charge operation needs to pre-charge not only the result storage but also the computational datapath. For a balanced processor, a pre-charge instruction needs to pre-charge the processor’s datapath and the destination registers or memory locations.

In the DRP circuits, pre-charge is usually done by pre-charging the inputs. The proposed solution uses a similar way to realize the pre-charge operations. It turns out that by setting the input operands of different balanced instructions to 0, the corresponding outputs are either all 0 or all 1 (for NOT). Since all 0 and all 1 are both acceptable pre-charge results,
the pre-charge operation of a balanced instruction can be done by just executing the same balanced instruction with all 0 inputs.

One detail about VSC is with the XOR instruction. Since the complementary counterpart of XOR, XNOR, cannot be pre-charged by simply applying 0 on the inputs, we expand XOR to AND, OR, and NOT instructions.

In summary, there is no need to add additional dedicated pre-charge instructions. Every balanced instruction can finish the pre-charge operation for itself by taking all 0 as the inputs.

4.4 VSC Programming

VSC programming has two basic constraints. First, only the balanced instructions can be used. Second, the direct or the complementary datapath can only take half of the datapath of the processor. Although there could be many ways to handle the programming based on the above constraints, we propose a programming methodology based on an extended version of bitslice programming that we call n-way bitslice programming.

4.4.1 n-way Bitslice Programming

Bitslice programming was discussed in detail in Chapter 2. For our purpose, we extend the definition of bitslice programming to a more general definition called n-way bitslicing. n-way bitslicing is the conversion of a program written for an m-bit processor to a program for a k-bit processor, then deploying m/k parallel instances (where m/k = n) of the k-bit program on the m-bit processor. In other words, n-way bitslicing is a programming technique to run n parallel instances of a program on an m bit processor using the concepts of bitslicing. Note that n is smaller than m, and m needs to be a multiple of n. This is so that an integral number of instances of the smaller program can fit in the m-bit datapath, leading to full
Program: \( c = a \& b; \)

\[
\begin{array}{c}
\text{c7 c6 c5 c4 c3 c2 c1 c0} \\
\hline
\text{a7 a6 a5 a4 a3 a2 a1 a0} \\
\hline
\text{b7 b6 b5 b4 b3 b2 b1 b0}
\end{array}
\]

(a)

Operation: \( a \ll 1 : \)

\[
\begin{array}{c}
\text{a7 a6 a5 a4 a3 a2 a1 a0} \\
\hline
\text{a6 a5 a4 a3 a2 a1 a0 0}
\end{array}
\]

(b)

Figure 4.5: 2-way bitslicing of an 8-bit program on an 8-bit processor. Different colors indicate different instances of the same program running in parallel. (a) Showing how an AND instruction is executed in 2-way bitslicing. (b) Showing how a left shift instruction is executed in 2-way bitslicing.

In n-way bitslicing, the bits of a variable are separated in chunks of k bits as opposed to 1 bit. Hence, classic bitslicing, as explained earlier, is simply m-way bitslicing (when implemented on an m-bit processor). For n-way bitslice programming, we also modify our definitions of BS(x) and BS(x,u) as they were defined for classical bitslicing. Here, BS(x) refers to the vector of variables that contain the separated chunks of bits of variable x. BS(x,u) refers to the variable within this vector that contains the u-th chunk of bits. Figure 4.5 explains how 2-way bitslicing is with the help of an 8-bit program 8-bit program running on an 8-bit processor utilization.
processor. It can be seen that the 8-bit variables are divided into chunks of 4 bits, and each chunk is stored in a different variable. Operations are then performed on these chunks.

Operations can be defined in the same way for n-way bitslicing as they were for classical bitslicing. AND, OR, XOR and NOT operations can be converted to n-way bitslice programming in the same fashion as they were for classical bitslicing (figure 4.5(a)). However, bit-wise shift operators are no longer simple reassignments. This case is shown in figure 4.5(b). While shifting, the programmer has to ensure that the data being shifted does not cross over horizontally into the other k-bit processor’s datapath, but rather shifts vertically across variables of the same bitslice vector.

For this thesis, we call the normal (non-bitsliced) method of writing a program as Normal-style Programming. We term writing a program in the bitsliced way as BS-style programming. We use BS-style programming to refer to both classical bitslicing as well as n-way bitslicing.

Based on the above discussion, we can enumerate the steps to convert a program written in Normal-style to a program in BS-style.

- **Step 1.** Break down the program written in Normal-style into the basic logical operations described above. Re-write the program in Normal style using these operations only.

- **Step 2.** Convert each logical operation into its n-way BS-style equivalent. This will require re-arrangement of program data at bit-level as described in the examples earlier.

Note that a program written in n-way BS-style programming will process n-sets of data sets at the same time, while a program written in Normal-style programming will process only 1 set of data at a time. Hence, while converting from Normal-style to BS-style, the programmer should ensure that n-sets of data are provided to the new program.
4.4.2 VSC Programming as an Extension of n-way Bitslicing

Bitslice programming is very suitable for VSC programming. Firstly, BS-style programming uses only logical operations. VSC programming uses the same logical operations in the balanced format. This makes it easy to convert a program written in BS-style to VSC.

BS-style programming also solves the problem of using half the processor’s datapath for direct and the other half for complementary operations. As we have seen, BS-style programming treats the underlying processor as n parallel processors. For VSC programming, n/2 parallel processors can designate the direct datapath, while the remaining n/2 can be designated as the complementary datapath.

The easiest way to understand VSC programming using bitslicing is with classical bitslicing or m-way bitslicing. m parallel instances of a program are deployed on an m-bit processor. Next, the instructions are converted to balanced instructions, and programs that fall in the complementary datapath are treated as the complementary program. This can be seen in figure 4.6.

Similarly, n-way BS-style programming can also be used to write programs for the VSC platform. Note that in n-way bitslicing, the chunks of bits of one program were contiguous within a variable, whereas for the VSC programs, the direct and complementary programs are interleaved. However, this is only an implementation issue. The same concepts of n-way bitslicing explained earlier can be extended to a format in which the parallel programs are interleaved with each other, with only minor modifications to the implementation of some instructions.

Once a program has been written using only the balanced instructions, the precharge instructions can be added before each balanced instruction. We call this way of writing VSC code using the concepts of bitslicing as BS_VSC-style programming.

N-way bitslicing is more suitable for developing VSC software than classical bitslicing. This is because on an m-bit processor, implementing VSC with classical bitslicing would require m/2
Figure 4.6: Processing arrays A: \((a_0, a_1, a_3, a_4)\) and B: \((b_0, b_1, b_2, b_3)\) with AND operations to generate C: \((c_0, c_1, c_2, c_3)\). (a) regular programming; (b) bitslice programming; (c) VSC programming.
parallel instances of the direct program running, otherwise the program would be extremely inefficient. Such data parallelism may not always be available to the programmer. With n-way bitslicing, the programmer has the freedom to choose the number of parallel instances. Hence, n-way bitslicing provides a more adaptable method for programming the VSC.

In summary, in order to write programs for the VSC platform, the following steps can be followed:

- **Step 1.** Convert a program written in Normal-style to n-way BS-style. Make sure that each program instance either falls into the direct part or the complementary part of the balanced processor.

- **Step 2.** Convert the regular instructions to balanced instructions. Add the corresponding pre-charge operation before each balanced instruction.

Note that it is the responsibility of the programmer to ensure that the direct and complementary datapaths receive their correct data respectively.
Chapter 5

Automatic bitslicing/VSC conversion

In the preceding chapters, we have discussed how bitslice programming is done, and how it is extended to do VSC programming. However, there are several challenges while trying to develop software for the VSC platform:

- VSC programming is based on bitslice programming. Bitslice programming requires the data of the program to be arranged in a manner that is counter-intuitive to more common programming practices. This increases the programming complexity and the time required to develop the program.

- Bitslice programming or VSC programming uses only logical operations. Breaking down a complex program down to purely logical operations is a cumbersome task, and adds further complexity to the design step.

- For VSC programming, it is the responsibility of the programmer to ensure that the complementary nature between the data of the complementary and direct datapaths of the program is maintained at all times. Care must be taken to avoid any operations that cause a loss of this symmetry.
The above problems make it difficult to develop and debug software. Ideally, while developing code for the VSC, one would like a solution or a tool that (a)abstracts the details of a bitsliced implementation from the user, and (b)abstracts the details of a VSC implementation from the user. In other words, it should be possible for a programmer to specify a program in Normal-style programming, and be able to run implement it in a BS_VSC-style programming without worrying about the low level operations and constraints.

In section 4.4 we have already discussed systematic methods and steps to convert Normal-style code to BS-style code and BS_VSC-style code. It can also be seen that these steps can be applied to any general program. Based on these rules, we can develop an automation tool or library that takes in a program specified in Normal-style and generates code from it that adheres to all the constraints of VSC software.

In this chapter we discuss how VSC code can be generated from normal code using a C++ library designed by us. We call this library AutoBS_lib. In the following sections we describe a programming paradigm using this library that allows a programmer to easily write code for the VSC platform.

5.1 The AutoBS_lib Library

The AutoBS_lib is a library designed in C++ that can be used for developing software for the VSC platform. It can be used for generating both BS-style code and BS_VSC-style code from Normal-style code. For this thesis, we only discuss how BS_VSC-style code can be generated using the AutoBS_lib library.

The code generation works as follows. First, a programmer uses the functions and classes of AutoBS_lib to develop software in Normal-style. The AutoBS_lib allows the programmer to simulate this program as if it were written in BS_VSC-style programming. This is useful because it can be used to develop and debug programs in Normal-style, which is much easier than doing it in BS_VSC-style. Once the functionality has been verified, the library can
be used to generate C code with the same functionality, that is written in BS_VSC-style. This code can be used in place of the original code that was written in Normal-style. The generated code arranges the input data in a bitsliced format and makes sure that only bitsliced operations are used on this data. It also ensures that the direct and complementary data lie in their correct datapaths. We call this generated code as VSC-C code. Note that at this point the generated C code contains normal instructions as opposed to balanced instructions, and also does not automatically incorporate pre-charge. Adding precharge and converting regular instructions to balanced instructions is a straightforward step. An
assembly file is generated from the VSC-C code, and all regular instructions are converted to their balanced counterparts, and precharge instructions are added. This can be easily done using a script file. A design flow of how the AutoBS_lib library is used to generate VSC code is shown in figure 5.1

### 5.2 Programming using the AutoBS_lib Library

This sections describes the functions and classes that are part of AutoBS_lib, and how to use them to simulate BS_VSC code.

- **The void setSliceCount(int n) function:** This function is used to set the number of slices or ‘parallel instances’ that the user wants to run of the program. The argument \( n \) essentially specifies the ‘\( n \)’ in \( n \)-way bitslicing. For BS_VSC-style code, this number specifies the number of direct datapaths. The program is generated assuming \( 2n \)-way bitslicing, and half of the program instances are designated as the complementary data-path.

- **The operand class and its member functions:** This class serves as the interface between the Normal-style program and its BS_VSC-style execution. Objects of this class are a representation of the program data in the bitsliced domain, i.e. each operand object represents \( n \) unsigned int variables in Normal-style code. Any operation performed on an object translates to the same operation happening on all \( n \) variables represented by this object. The program that the programmer wants to execute in the bitsliced way is written in Normal-style using the objects of this class as its variables. The operand class has the following methods for its operation:

  - **The void operand::Input(unsigned* p) function:** This function is used convert \( n \) unsigned int variables into the bitsliced format, and associate the operand with these \( n \) variables. The argument unsigned* \( p \) contains the pointer to the
array that contains the n variables. For BS_VSC-style code generation, the complementary data for all input variables is automatically generated and assigned to the complementary programs.

- Set of Overloaded Operators \& , |, ^ , ~ , << , >>, =: These operators are used to perform operations on objects of class \texttt{operand}. Any operation performed on operands using these operators is converted to its BS_VSC-style equivalent operation.

- The \texttt{void operand::Output(unsigned* q)} function: This function is used to convert the bitsliced data associated with an operand back to n individual unsigned integers. The argument \texttt{unsigned* q} contains the pointer to the array that contains the n extracted variables. For BS_VSC-style code generation, only the direct data is extracted, while the complementary data is discarded.

Programs written with the above functions can be used for both simulating and generating the VSC-C code. We can understand the usage of this library to write programs using the above functions with the help of the following example. Say we want to write a function for adding two 32-bit numbers and run it on the VSC processor. According to the first step of writing BS_VSC-style programs, we need to break down the algorithm into logical operations. The following code snippet shows how two 32 bit numbers can be added using purely logical operations in Normal-style:

```c
void Add32(unsigned a, unsigned b, unsigned *s){
    unsigned carry, d, e, f; // Variables for holding //intermediate results
    int i;
    unsigned Sum; //Contains the final sum at the end
    Sum = 0;
    d = a^b;
    f = a&b;
```
carry = 0;
for (i = 0; i < 32; i++) {
    Sum = Sum | ((carry ^ d) & (0x1 << i));
    e = d & carry;
    carry = ((e | f) & (0x1 << (i))) << 1;
}
*s = Sum; // Store the result back
}

If we wanted to add n different sets of numbers, we would make the function call n times.
Say we want to add 4 different sets of numbers. The following code shows how the function
call will be made:

main() {
    unsigned Arg1[4], Arg2[4];
    unsigned sum[4];
    int i;
    // assign values to the arrays Arg1 and Arg2;
    // for all n, Arg1[n] and Arg2[n] are added,
    // and stored in sum[n]
    for (i = 0; i < 4; i++) {
        Add32(arg1[i], arg2[i], &sum[i]);
        // call to Add32 function
    }
}

As we are adding 4 different sets of numbers, we can execute 4 parallel instances of the
same program using bitslice execution. Furthermore, we can also go ahead and convert it to
VSC style code. The following code shows how to do this using the functions of AutoBS.lib
described above:
void Add32 (unsigned* A, unsigned* B, unsigned* S)
{
    // unsigned* a and unsigned* b
    // contain the 4 unsigned int variables each
    // S is used to store the output

    operand a, b; // operands to store values in arrays A and B
    operand carry, d, e, f; // Variables for holding
    // intermediate results
    operand Sum; // Contains the final sum at the end
    int i;

    a. Input (A); // Convert variables in A to BS_VSC-style format,
    // associate the operand a with it
    b. Input (B); // Convert variables in B to BS_VSC-style format,
    // associate the operand b with it

    Sum = 0; // Operations operate on all 4 variables associated
    d = a ^ b; // with each operand in BS_VSC-style format
    f = a & b;
    carry = 0;
    for (i = 0; i < 32; i++)
    {
        Sum = Sum | ((carry ^ d) & (0x1 << i));
        e = d & carry;
        carry = ((e | f) & (0x1 << (i))) << 1;
    }

    Sum. Output (S); // Extract individual variables from Sum
    // and store in S. S contains the out for all
main() {
    setSliceCount(4); // Set number of parallel instances to 4
    // For VSC, we'll have 4 direct path and
    // 4 complementary paths.
    unsigned Arg1[4], Arg2[4];
    unsigned sum[4];
    int i;
    // assign values to the arrays Arg1 and Arg2;
    // for all n, Arg1[n] and Arg2[n] are added,
    // and stored in sum[n]
    Add32(arg1, arg2, sum); // call to Add32 function
}

The above code will simulate the function Add32 as if it were executed in BS VSC-style. To
do this, the programmer only needs to compile and run the code. Once the programmer is
satisfied that the code functionality is correct, code generation can be done. Code generation
is done by appending a function call to a function called "Code_gen(string func_name)" at
the end of the function for which code needs to be generated:

void Add32(unsigned* A, unsigned* B, unsigned* S) {
    ...

    Code_gen("Add32(unsigned* A, unsigned* B, unsigned* S)");
    // This function generates the BS VSC-style code for this function
A function call to \texttt{Code\_gen} converts each statement (which has been written using the functions and classes of \texttt{AutoBS\_lib}) in the function in which it has been called with its bitslice equivalent in simple C. First, all \texttt{operand} variables are converted into vectors of unsigned int variables that hold the bitsliced data. Next, all operations are replaced with their BS\_VSC-style equivalent operations. Flow-control statements such as the \texttt{for} loop in the above example, are completely unrolled in the generated code. In principle, branch-control statements can also be supported by breaking them down to logical operations (like a multiplexer). The generated code is then placed in a new file under the function name that has been specified in \texttt{Code\_gen}'s argument \texttt{string func\_name}. The generated function has the same argument list as the original function. As in the above example, the same function name as that of the original function can be used for generating code. This allows one to simply plug in the generated function in place of the original function and run the program.

Figure 5.2 shows what the generated code in BS\_VSC-style looks like for some simple statements in written in Normal-style using the \texttt{AutoBS\_lib} library.

Once VSC-C code has been generated, addition of pre-charge instructions and balanced instructions conversion can be done as described earlier.

\section{VSC-AES using \texttt{AutoBS\_lib}}

The \texttt{AutoBS\_lib} was used to develop a VSC implementation of the AES-128 algorithm. This code was run and tested on a Leon3 processor that was modified as per the VSC architecture. The details of this processor are given in [19].

First, all major functions of AES were written in Normal-style using logical operations. The SBOX, which is generally a memory lookup, was broken down into logical operations using the implementation provided in [24]. The number of parallel instances was set to 1. Next,
<table>
<thead>
<tr>
<th>Normal-style Code</th>
<th>VSC-C Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Add32(unsigned* A, unsigned* B, unsigned* S) {</td>
<td>void Add32(unsigned* A, unsigned* B, unsigned* S) {</td>
</tr>
<tr>
<td>( ) setSliceCount(4); ( ) unsigned a0_0; ( ) unsigned a1_0; ( ) operand a is split into a vector of variables that contain the direct and complementary data in bit-sliced format. ( ) f is split into a set of similar variables.</td>
<td></td>
</tr>
<tr>
<td>... ( ) unsigned a0_1; ...</td>
<td></td>
</tr>
<tr>
<td>operand a, f; ( \Rightarrow ) ( ) ( ) operand a is split into a vector of variables that contain the direct and complementary data in bit-sliced format. ( ) f is split into a set of similar variables.</td>
<td></td>
</tr>
<tr>
<td>... ( ) unsigned a0_2; ...</td>
<td></td>
</tr>
<tr>
<td>... ( ) unsigned a0_3; ...</td>
<td></td>
</tr>
<tr>
<td>... ( ) unsigned a0_4; ...</td>
<td></td>
</tr>
<tr>
<td>... ( ) unsigned a0_5; ...</td>
<td></td>
</tr>
<tr>
<td>... ( ) unsigned a0_6; ...</td>
<td></td>
</tr>
<tr>
<td>... ( ) unsigned a0_7; ...</td>
<td></td>
</tr>
<tr>
<td>a.Input(A) ( \Rightarrow ) ( // arrange the data in A into the variables defined above )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ... ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ... ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ... ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ... ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>( f = a &amp; b; ) ( \Rightarrow ) ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>( a2_0 = (a0_0 &amp; a1_0); ) ( \Rightarrow ) ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>( a2_1 = (a0_1 &amp; a1_1); ) ( \Rightarrow ) ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>( a2_2 = (a0_2 &amp; a1_2); ) ( \Rightarrow ) ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>( a2_3 = (a0_3 &amp; a1_3); ) ( \Rightarrow ) ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>( a2_4 = (a0_4 &amp; a1_4); ) ( \Rightarrow ) ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>( a2_5 = (a0_5 &amp; a1_5); ) ( \Rightarrow ) ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>( a2_6 = (a0_6 &amp; a1_6); ) ( \Rightarrow ) ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>( a2_7 = (a0_7 &amp; a1_7); ) ( \Rightarrow ) ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>Sum.Output(S); ( \Rightarrow ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ( // a set of statements that )</td>
<td></td>
</tr>
<tr>
<td>... ( ) ( // a set of statements that )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.2: Examples of generated VSC-C code using AutoBS-lib
the code was simulated using the AutoBS.lib library to verify the correctness of operation. After this, the VSC-C code was generated using the methods described earlier.

For adding pre-charge instructions and converting regular instructions to balanced instructions, the following steps were followed. First, the VSC-C code was compiled to an assembly file using the sparc-elf-gcc compiler. Optimization was set to -O2. Next, all regular instructions in functions belonging to AES were changed to balanced instructions, and a pre-charge instruction was added before each balanced instruction.

The code was then compiled to an executable file and run on the processor. Correctness of execution was verified by observing the outputs of the direct and complementary datapaths, which were found to be complementary. As all regular instructions had been converted to balanced instructions, and the corresponding pre-charge instructions had been added, it was concluded that the code was correct in terms of VSC operation.

Table 5.1 compares the code footprint and execution time between an AES program written in normal style and BS_VSC-style code generated from this program. It can be seen that the generated code is 164 times larger as compared to Normal-style code, and its execution time is 325 times longer. There are three reasons for this. First, BS_style-code has a lot of additional overhead like pre-charge instructions, and conversion of data from bitsliced format and back. Second, every operation in Normal-style code is split into $n$ operations in BS_VSC-style code. Third, the code generation by the AutoBS.lib library is currently geared towards generating functionally correct code, not towards generating optimized code. As a future scope of this work, the library could be upgraded to generate functionally correct as well as optimized code.
Chapter 6

Conclusion

In this thesis we have addressed two challenges that are an inherent part of the efficient implementation of SCA countermeasures. Our work on evaluating different implementations of AES clearly shows the relationship between design choices and the side channel resistance of the system. The contribution of this work is in putting these implementations side-by-side, and demonstrating a practical and comprehensive evaluation. There are many extensions possible on this work. For example, one could consider the impact of side-channel resistant design techniques (such as masking) on performance and side-channel resistance. Or, one can consider additional attacks, such as those based on timing or on faults.

Furthermore, we have identified the problems with developing SCA resistant software on the VSC platform, and have presented a feasible solution to this problem in form of the AutoBS_lib library. Further work in this area would include ungrading the library in order to generate optimized code.
Bibliography


