University of Cincinnati

Date: 6/18/2012

I, Greg Mefford, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
Side Channel Analysis Research Framework (SCARF)

Student's name: Greg Mefford

This work and its defense approved by:

Committee chair: Ranganadha Vemuri, PhD
Committee member: Wen Ben Jone, PhD
Committee member: Carla Purdy, PhD
Side Channel Analysis Research Framework (SCARF)

A thesis submitted to the

Division of Research and Advanced Students
of the University of Cincinnati

in partial fulfillment of the requirements
for the degree of

Master of Science

in Computer Engineering
in the School of Electronic and Computing Systems
of the College of Engineering and Applied Science
University of Cincinnati
June 2012

by

Greg Mefford

B.S.Electrical Engineering
University of Cincinnati, Cincinnati, OH
June 2009

Thesis Advisor and Committee Chair: Ranga Vemuri
Side Channel Analysis Research Framework (SCARF)

Abstract

Getting started in the research of side-channel information leakage from cryptographic systems is difficult and time-consuming. In addition to the time spent learning about the various mechanisms through which information can be leaked and the metrics used to quantify and analyze the leakage, the researcher must develop tools to collect the necessary data and process it in a standard way in order to compare results across multiple experiments. The Side-Channel Analysis Research Framework (SCARF) is presented as a solution to the complexities of designing a software infrastructure capable of managing both simulation and measurement data to produce relevant metrics and plots relating to cryptographic side-channel analysis research. SCARF consists of a software framework built on top of the Ruby programming language and using the open-source Redis database as an efficient in-memory object store. It is distributed with fully-working example code to demonstrate its use in side-channel analysis attacks against a simplified version of the Data Encryption Standard (DES) and against a physical implementation of Microchip’s KeeLoq® technology [13], often used in remote keyless-entry applications for vehicles and garage door openers. Through the implementation of multiple example attacks, we demonstrate SCARF as a useful tool for researching and comparing side-channel attacks on both physical and proposed cryptographic hardware.
Acknowledgments

First, I would like to thank Dr. Vemuri for his superb instruction, which provided me with both a passion for - and a knowledge of - VLSI design and implementation. Further, I would like to thank him for his patience and guidance as I spent longer than I had planned to complete my degree.

I would like to thank Mike Borowczak for helping me to puzzle out the details of performing power-analysis attacks on hardware with all the added complexities of making sense out of real-world measurements and digital communication protocols.

Early work in motivating this thesis and preliminary data collection for the example attack on KeeLoq occurred out of a collaborative project funded in part by a grant secured by Dr. Jintai Ding in the Mathematics Department at the University of Cincinnati. The results of this initial work have been published as an internal white paper [5].

Finally, and most importantly, I want to thank my family who have been extremely supportive and accommodating as I spent many nights and weekends working on this research while also holding a full-time job.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title Page</td>
<td>i</td>
</tr>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 An Example Cipher | 2 |
1.2 Motivation for Thesis | 6 |
1.3 Goals | 7 |
1.3.1 Software Framework | 7 |
1.3.2 Practical Usage Examples | 9 |
1.4 Organization of Thesis | 10 |

## 2 Differential Power Analysis

2.1 Side-Channel Attacks | 11 |
2.1.1 Simple Power Analysis | 12 |
2.1.2 Differential Power Analysis | 14 |
2.2 Leakage Models | 18 |
2.2.1 Hamming Weight | 19 |
2.2.2 Hamming Distance | 20 |
2.3 Measurement Methods | 21 |
2.4 Analysis Methods | 22 |

## 3 Core Framework

3.1 Architectural Overview | 24 |
3.2 Data Models | 25 |
3.2.1 Database Back-End | 26 |
3.2.2 Trace | 26 |
3.2.3 Trace Group | 27 |
3.2.4 Key Guess | 28 |
3.3 Processing Steps | 29 |
3.3.1 Data Import | 29 |
3.3.2 Trace Analysis and Transformation .......................... 31
3.3.3 Attack ......................................................... 32
3.3.4 Post-Process ................................................. 34

4 Simplified DES Example .......................... 35
4.1 The Modified DES Cipher .................................. 36
4.2 Attacks in the Literature ..................................... 38
4.3 Leakage Model ................................................. 39
4.4 Simulated Attack Process ................................. 39
  4.4.1 Data Import ........................................... 39
  4.4.2 Trace Analysis ......................................... 40
  4.4.3 Trace Transformation .................................. 40
  4.4.4 Attack ................................................... 41
4.5 Simulated Attack Results ................................. 41

5 KeeLoq Example .......................... 48
5.1 The KeeLoq Cipher ............................................ 50
5.2 Attacks in the Literature ..................................... 51
5.3 Leakage Model ................................................. 52
5.4 Simulated Attack ............................................. 52
  5.4.1 Simulated Attack Results ................................ 57
5.5 Hardware Attack ............................................. 61
  5.5.1 Experimental Setup ................................... 62
  5.5.2 Device Configuration ................................... 66
  5.5.3 Trace Capture and Alignment .......................... 70
  5.5.4 Trace Analysis and Compression ........................ 74
  5.5.5 Attack on Measured Traces ............................... 76

6 Conclusions .......................... 83
6.1 Raw Trace Analysis .......................................... 83
6.2 Leakage Modeling ............................................ 84
6.3 Statistical Analysis ......................................... 84
6.4 Side-Channel Attack Model ................................. 85
6.5 Post-Processing ................................................ 85
6.6 Source Code Organization ................................... 86
6.7 Practical Usage Examples ..................................... 87
6.8 Future Work ................................................... 88
  6.8.1 SCARF Improvements ................................... 88
  6.8.2 Additional Example Attacks ............................. 89
  6.8.3 Higher-Level Research .................................... 90

Bibliography .......................................................... 91
## List of Figures

1.1 XOR Encryption .................................................. 2  
1.2 XOR Decryption .................................................. 3  
1.3 Lumped-C Model of a CMOS Inverter [18] ....................... 5  

2.1 Simple Power Analysis Example of an RSA Implementation [15] .............. 13  
2.2 Leakage Models .................................................. 15  
2.3 Identifying Sensitive Data ..................................... 16  
2.4 Generating Synthetic Leakage Traces ........................... 17  
2.5 Correlate Measurements with Leakage Model ..................... 17  
2.6 Propagating Key Guesses ...................................... 18  
2.7 Hamming Weight ................................................ 19  
2.8 Hamming Distance .............................................. 20  
2.9 Hamming Distance Formula ................................... 20  

3.1 SCARF Capture Workflow ...................................... 30  
3.2 SCARF Analyze Workflow ..................................... 31  
3.3 SCARF Transform Workflow ................................... 32  
3.4 SCARF Attack Workflow ...................................... 33  
3.5 SCARF Post-Process Workflow ................................ 34  

4.1 Block Diagram of the Modified DES Algorithm ................. 37  
4.2 Distribution of SBOX Output Hamming Weights from 10000 Encryptions .... 40  
4.3 DES Attack Performance Benchmark ........................... 42  
4.4 DES Guess Metrics for 16 Simulated Traces ..................... 43  
4.5 DES Guess Metrics for 64 Simulated Traces .................... 44  
4.6 DES Guess Metrics for 128 Simulated Traces ................... 45  
4.7 DES Guess Metrics for 256 Simulated Traces ................... 46  
4.8 DES Guess Metrics for 512 Simulated Traces ................... 47  

5.1 KeeLoq Encryption [29] ......................................... 51  
5.2 Simulated KeeLoq Leakage Trace Samples ..................... 54  
5.3 Keeloq Performance Benchmark ............................... 58  
5.4 Keeloq Guess Metrics for 2 Simulated Traces .................. 59
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Keeloq Guess Metrics for 50 Simulated Traces</td>
<td>60</td>
</tr>
<tr>
<td>5.6</td>
<td>Sample Keeloq Trace [9]</td>
<td>61</td>
</tr>
<tr>
<td>5.7</td>
<td>Keeloq Experimental Setup</td>
<td>62</td>
</tr>
<tr>
<td>5.8</td>
<td>Keeloq Target Board Schematic</td>
<td>63</td>
</tr>
<tr>
<td>5.9</td>
<td>Keeloq Target Board Photograph</td>
<td>64</td>
</tr>
<tr>
<td>5.10</td>
<td>Keeloq Trace Characteristics - Sample Trace, Mean, and Standard Deviation</td>
<td>72</td>
</tr>
<tr>
<td>5.11</td>
<td>Keeloq Measured Trace Characteristics - Detail</td>
<td>75</td>
</tr>
<tr>
<td>5.12</td>
<td>Keeloq Measured Leakage Trace Samples</td>
<td>77</td>
</tr>
<tr>
<td>5.13</td>
<td>Keeloq Measured Residual Samples</td>
<td>80</td>
</tr>
<tr>
<td>5.14</td>
<td>Keeloq Guess Metrics for 20 Measured Traces</td>
<td>80</td>
</tr>
<tr>
<td>5.15</td>
<td>Keeloq Guess Metrics for 25 Measured Traces</td>
<td>81</td>
</tr>
<tr>
<td>5.16</td>
<td>Keeloq Guess Metrics for 250 Measured Traces</td>
<td>82</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Cryptographic systems are primarily designed to protect sensitive data from being visible to an attacker while allowing the intended recipient of the data to easily recover it. This process is normally accomplished by performing a complex series of operations on a message, guided by the information stored in a secret key. The cryptographic algorithm, called a cipher, is normally a black-box operation, taking in a Plaintext Message and converting it to a Ciphertext Message using the Secret Key for the case of Encryption. The inverse operation is called Decryption. It is normally assumed that the algorithm itself is well-known, but the secret key, plaintext message, and the intermediate data stored during the encryption process are not known.
1.1 An Example Cipher

A simple implementation of this concept would be to use a pre-shared key and a simple encryption cipher that consists of performing an exclusive-or (XOR) between the plaintext message and the secret key in order to produce a ciphertext. The operation of XOR is that each non-zero bit in the binary representation of the key causes the corresponding bit in the plaintext message to be inverted. Zero bits do not invert the corresponding bit in the plaintext. The XOR function is often used as a component of cryptographic algorithms because of its ability to reversibly transform data in a key-dependent way that is easily implemented using digital logic. An illustration of the XOR encryption process using a fixed-size block of data is shown in Figure 1.1. Since the XOR function is reversible using the same secret key, the decryption process can be performed as shown in Figure 1.2.

![Figure 1.1: XOR Encryption](image)

This XOR cipher is easy to conceptualize and understand, but it is also insecure as a cryptographic algorithm. It can be broken in the case where the attacker can predict some properties of the plaintext due to the reversible nature of the XOR function. We assume that the attacker has access to the ciphertext and is able to predict some bits of the plaintext by some means. For example, it may be well-known that the messages consist of ASCII-encoded strings containing printable characters. The attacker can thus immediately rule out any key
hypotheses that, when XORed with a captured ciphertext message, produce a non-printable character since these characters are not allowed to appear in the plaintext. For example, the first character of the key could not be the ASCII ‘1’ character because XORing it with the first character of the ciphertext message, an ASCII ‘9’ character, would produce the byte $0x08$, which represents a backspace and is not printable. If the secret key does not change, by accumulating these guesses of parts of the secret key over several ciphertext messages, the most likely candidates for the entire secret key are easily distinguished from the impossible candidates. How quickly this process converges depends on how much information the attacker has about the plaintext being input to the block cipher. In the degenerate case where the attacker has complete knowledge of both plaintext and corresponding ciphertext, the key can be recovered immediately by a simple XOR of the plaintext and ciphertext, directly producing the secret key itself.

Because of these weaknesses in early ciphers, much stronger algorithms were developed to further decouple the correlation between the plaintext and ciphertext. Many popular ciphers, such as Advanced Encryption Standard (AES) and Data Encryption Standard (DES), make use of the XOR function internally, but apply it to an iteratively permuted intermediate value, which is not visible to the attacker. These iterations are commonly called rounds of encryption (or decryption). The purpose of encryption rounds is to prevent the
attacker from being able to guess pieces of the key as described previously by increasing the entropy of the intermediate value with each round of encryption such that, on the last round, the attacker has no way of predicting the properties of the intermediate value prior to the final XOR operation, regardless of the attacker’s knowledge of the initial plaintext.

The final piece of the story leading up to the material presented in this thesis is the concept of a cryptographic side-channel. A side-channel encapsulates the concept of an additional channel that allows unintended information leakage along with the normal channels containing the plaintext and ciphertext data. In order to discuss a real-world example of side-channel information leakage, it will be helpful to briefly discuss how cryptographic systems are implemented in physical semiconductor integrated circuits.

In physical cryptographic devices, internal state information such as secret keys, plaintext messages, ciphertext messages, and intermediate values are generally stored as discrete voltage levels, either low or high, called binary bits. Each bit is stored in a flip-flop, which maintains its stored state as a voltage on the output of the storage device, which drives the input of other logic gates in the cryptosystem. These single-bit storage devices are often arranged into registers such that parallel processing can be performed according to the algorithm being implemented. All of the bits in a register are updated at the same time at some regular interval, again dictated by the algorithm being implemented. In the case of devices implemented using the Complementary Metal-Oxide-Semiconductor (CMOS) technology, each time a register’s stored value is updated, the device draws an impulse of current through its power supply pins. For each internal bit that switches from a low voltage to a high voltage, the device must pull a dynamic current through the high-voltage supply to charge up the capacitive loads of the buses and inputs of the logic gates being driven by that bit in the register. Similarly, each bit transitioning from high to low must discharge its capacitive load out the device’s ground pin. Additionally, a CMOS
Figure 1.3: Lumped-C Model of a CMOS Inverter [18]

register exhibits a static current related to the value of each bit being stored rather than the transition from one value to another. These properties can be modeled using a Lumped-C Model of the device, as shown in 1.3 for a CMOS inverter.

Based on these properties of CMOS registers, an example of side-channel information leakage can be formed based on the preceding description of a cryptographic algorithm that performs iterative permutation and XORing of an internal data register based on a plaintext and an internal secret key in order to produce a ciphertext. The goal of an attack is to recover the secret key used for encryption assuming that the attacker has access to plaintext, ciphertext, or both. Note that, for a well-designed cipher like AES, the secret key can not be easily recovered using currently available computing equipment even if both plaintext and ciphertext are known. If the attacker has physical access to the cryptographic hardware, an additional source of information may be available in the form of side-channel information leakage such as the instantaneous power consumed by a CMOS device. This power can be measured in a number of ways, such as measuring the voltage across a current-sense shunt resistor in series with the power input pin, or an probe placed
near the pin to sense the magnetic field induced by the current flowing through it. The value of this side-channel is that it gives the attacker some information about the properties of the intermediate values in the device, which would otherwise not be available. From here, the attacker can employ various techniques to guess the values of the hidden intermediate data in the cryptosystem based on the power consumption profile of the device during each instant of the encryption process. Finally, the hypothetical values for intermediate values can be used to guess parts of the secret key in the way described in the conceptual attack in the preceding paragraphs.

1.2 Motivation for Thesis

In order for a researcher to study these the types of attacks and design countermeasures to prevent the exploitation of side-channel information leakage, he or she must have a deep understanding of the algorithm under test, the available technologies for physical implementation, and the characteristics of the side-channels documented in the literature. Additionally, the researcher must design and implement the infrastructure necessary to adequately model, simulate, measure, analyze, compare, and report on the effectiveness of various side-channel attacks on the device under test in order to determine the vulnerability of a design to those particular types of information leakage. This thesis work reduces that burden by developing software infrastructure that allows the researcher to quickly focus on the particular details of the algorithm under investigation instead of building boiler-plate data structures and analysis algorithms.
1.3 Goals

The primary goal of this thesis work is to provide a foundation of software building-blocks that cryptographic researchers can use in side-channel analysis research. The intent is that the building-blocks are generic enough that they will be useful for side-channel research either directly or with minor modifications. The utility of these modules is that the researcher has a working set of open-source software that can be examined and tweaked to match the particular use-case being researched.

In addition to the software framework, a related goal of this thesis is to generate two complete examples of the system being used to study different cryptographic algorithms and experimental setups. These examples not only demonstrate that the software is generally useful and applicable to this field of research, but also give future researchers a starting point upon which to study algorithms or experimental setups that are similar to those shown in the examples. The intent of providing open-source usage examples is that future researchers would contribute examples of their work back to the project, increasing the body of knowledge and making the framework more useful over time.

1.3.1 Software Framework

The primary goal of the software framework is to provide valuable re-usable tools in the following areas:

- **Raw Trace Analysis**

Facilities should be included to allow traces to be plotted and inspected, combined, and permuted through various functions to view mean traces, residual traces, and standard deviation traces.
• **Leakage Modeling**

The framework should allow the user write arbitrary code to describe the leakage model of the system being studied, which will be used in the attack phase. Re-usable functions should be included in the core software for calculating Hamming Weight and Hamming Distance.

• **Statistical Analysis**

Library functions should be available to provide standard statistical methods, such as mean, standard deviation, and Pearson Correlation.

• **Side-Channel Attack Model**

The framework should support a flexible attack workflow, allowing the data traces to be processed in groups and across multiple phases of a complex attack process. The researcher should be able to implement an arbitrary attack process or modify an existing attack without having to fully understand or modify the details of the provided data structures.

• **Post-Processing**

As a final step of the attack workflow, the software should offer some post-processing of the results generated during the attack. This post-processing includes generation of simple tabular data and plots of various metrics used to characterize the attack’s effectiveness.

Another goal of the software framework is that the source code itself be approachable and simple to modify and maintain. This allows the responsibility of maintenance and improvement of the framework to be more easily shared among a group of researchers that evolves over time as new graduate students join the team and others leave the team.
It is not a goal that this software be tuned for computational performance at the expense of the other goals. Care should be taken to ensure a useful level of performance by supporting multi-processing and optimizing the critical sections of the code. However, code clarity and maintainability are valued over performance optimization beyond what is required to run the analysis in a practical amount of time on modern commodity computing resources.

1.3.2 Practical Usage Examples

The goal of the usage examples is to provide a body of knowledge and starting point for future researchers to quickly see how to use the framework to answer real research questions. The examples will initially be modeled on attacks that have been previously documented in the literature in order to demonstrate how the system could have been used in actual scenarios being presented in the research community.

The initial examples presented in this thesis work are an attack on a simplified variant of the Data Encryption Standard (DES) algorithm and the an attack on the KeeLoq algorithm. The DES example is carried out using only software simulation because physical hardware does not exist for this variant of the DES algorithm. The KeeLoq example demonstrates the same attack being performed both in simulation and using measurements collected from a physical implementation of the algorithm in a commercially-available device.
1.4 Organization of Thesis

The remainder of this thesis is organized into the following chapters:

- **Chapter 2** gives an introductory background description of the process of performing a cryptographic side-channel attack using Differential Power Analysis (DPA).

- **Chapter 3** discusses the design and implementation of the SCARF tool.

- **Chapter 4** provides a worked example of SCARF being used to attack a simplified version of the Data Encryption Standard (DES) algorithm.

- **Chapter 5** gives another practical example of using SCARF to carry out a full attack on the KeeLoq cryptosystem, including an attack on simulated traces as well as physically measured traces.

- **Chapter 6** makes observations and conclusions about the use cases for SCARF in terms of the examples presented in this thesis and possible future uses for the tool. Additionally, this chapter describes areas for future improvement in the SCARF tool in particular and in cryptographic side-channel research in general as it relates to this thesis work.
Chapter 2

Differential Power Analysis

This thesis discusses the design and usage of the proposed Side-Channel Analysis Research Framework (SCARF) and the theory supporting its design. The following provides background from previous research and describes the key topics in detail, establishing a foundation upon which SCARF is designed.

2.1 Side-Channel Attacks

A cryptographic Side-Channel Attack (SCA) is an attempt to recover secret information, such as an encryption key, in which the attacker does not only directly analyze the plaintext and/or ciphertext in order to determine the secret key, but also incorporates additional information not accounted-for by the designer and implementer of the cryptographic algorithm. A side-channel can take on many forms, such as the amount of time taken, or the amount of power consumed to perform steps in the encryption process. In the case of power-analysis side-channel attacks, the side channel is the instantaneous power consumption of the physical device during each round of encryption. In devices without countermeasures to mitigate their effectiveness, this type of attack can drastically impact
the security of the cryptosystem.

An additional strength of power-analysis side-channel attacks is that they are non-destructive attacks. This means that the physical device is not necessarily damaged in the attack. For example, the power consumption of the device can be measured by the amount of electromagnetic radiation emitted by its power pins as current pulses through the device with each clock edge. By contrast, a destructive attack would be to dissolve the plastic encapsulation away from the die and monitor the physical metal layers of the device with tiny probes in order to determine the data values traversing the hidden data bus inside the device. Unlike electromechanical methods of physical protection against and detection of destructive tampering, mitigation strategies for power-analysis side-channel attacks are more complex because, in general, devices must receive their power from an outside source, making this side-channel available for analysis. These strategies are an area of active research to achieve optimal effectiveness while minimizing the overhead of implementing them.

2.1.1 Simple Power Analysis

Power-analysis SCAs come in several forms of varying complexity and effectiveness. The simplest type is called a Simple Power Analysis (SPA) attack [16]. In an SPA attack, the attacker uses direct inspection of a power trace to determine information about the cryptographic operations being performed by the device under test, such as the intermediate data values being processed.
SPA attacks are generally most effective against a microprocessor-based system, where different machine instructions often have a distinctive power signature. For example, in RSA encryption, a secret exponent is used in a modular exponentiation, which is primarily composed of a multiplication operation and a square operation in a key-dependent pattern. If the multiplication consumes more power than the square operation, for example because the square is implemented as a binary left-shift, then the pattern of operations can be clearly seen in the power trace as shown in Figure 2.1 [15]. A branch instruction in a microprocessor is also a common target of SPA attacks because of the different power signatures of different execution paths. By visual comparison of traces with different execution paths, it may be possible to determine where in the encryption process the traces diverge in order to guess whether a branch was conditionally taken based on key material.

![Figure 2.1: Simple Power Analysis Example of an RSA Implementation](image)

At a more abstract level of detail, SPA also often allows the attacker to visually inspect one or more power traces to determine where its major features occur. For example, it is often possible to identify a common setup phase, a periodic repetition of N encryption rounds, and a common clean-up phase. If the cryptosystem is designed synchronously, it is often straightforward to delineate exactly the power signature of each clock cycle, which greatly helps in alignment of the important samples for an attack.
Microprocessor systems are particularly vulnerable to SPA attacks because of the relatively small amount of data processed during each clock cycle. Processor-based low-power cryptosystems are often designed using an 8-bit architecture in order to keep the device size small and minimize power consumption. If the attacker can identify the power signature of each clock cycle in an 8-bit microprocessor and can theorize about what operations are being performed during each clock cycle, then a leakage model can be used to hypothesize the value of the data being processed by direct inspection of a single power trace.

2.1.2 Differential Power Analysis

Sometimes, SPA alone is not sufficient to recover secret data from a set of power traces. After SPA has been used to align a set of traces so that corresponding points from each trace are temporally collocated, a more advanced technique called differential power analysis (DPA) can be employed [16]. DPA attacks employ statistical methods to correlate the variations among the set of traces against a leakage model. Compared to an SPA attack, a DPA attack is much more powerful because it can uncover high-order information that can not be directly seen from a single power trace. Since it uses statistical properties of a potentially large set of power traces to perform the attack, it is also much more robust against noise, as the persistent signal rises above the uncorrelated noise with an increasing number of samples. The high-level process steps in a DPA attack are described in more detail in the following paragraphs.
• **Choose a Leakage Model.** This is often a matter of predicting the leakage characteristics of the device under test. Two common choices for leakage models are the Hamming Distance and Hamming Weight of an internal register of interest. More detailed discussion of these two leakage models is forthcoming.

![Figure 2.2: Leakage Models](image)

Figure 2.2: Leakage Models
• **Identify a Small Piece of Sensitive Data that can be Independently Correlated to the Power Trace.** The purpose of this step is to “divide and conquer” the problem space into smaller pieces that can each be attacked independently by correlation against the set of power traces. This limits the number of guesses that the attacker must generate and correlate against the data set. An example of this is to independently attack each Substitution Box (SBOX) used in the DES cipher. By correlating against a single SBOX, the number of partial key guesses required to exhaustively search the solution space is only $2^6 = 64$ for a six-input SBOX rather than $2^{48} = 281,474,976,710,656$ for the entire 48-bit round sub-key. It is possible to correlate against a single SBOX in this case because each of the other SBOXes contributes a component to the leakage metric that is not correlated with the SBOX under attack. These noise components are overcome by aggregating the results of a sufficient number of experiments.

![Figure 2.3: Identifying Sensitive Data](image)

Figure 2.3: Identifying Sensitive Data
- **Use the Chosen Leakage Model to Generate Synthetic Leakage Traces.** For each partial key guess in the solution space identified in the previous step, generate a power trace of the leakage model assuming that guessed key bits were used in the encryption.

![Figure 2.4: Generating Synthetic Leakage Traces](image)

- **Correlate Measurements with Model Traces.** Using statistical techniques, determine the likelihood that each partial key guess matches the real partial key used during encryption. A common technique used for this correlation is the Pearson Correlation Coefficient.

![Figure 2.5: Correlate Measurements with Leakage Model](image)
• **Propagate Partial Key Guesses.** As more small pieces of the key are determined, they are integrated together into larger sections of the overall key being guessed in order to weed out overlapping cases where multiple partial keys correlate equally well, but when permuted together, only certain combinations correlate well with the overall key. Ultimately, this process ends in the degenerate case where model traces representing guesses of the entire secret key are correlated against the measured traces to determine the best candidates for the full key. At this point, if there are several candidates with high correlation, the correct one can be determined by brute force, as the number of candidates will be small at this final stage.

![Figure 2.6: Propagating Key Guesses](image)

2.2 **Leakage Models**

In a power-analysis side-channel attack, it is often very helpful to use a simplified model of the behavior of the complex operation of the device under test. It is important to choose a model that closely matches the actual behavior of the physical device, but abstracts the details away such that processing the correlation between the model and the measurements can be done efficiently. Two popular methods of modeling side channels in semiconductor devices are the Hamming Distance and the Hamming Weight of the data being processed.
2.2.1 Hamming Weight

The Hamming Weight model simply counts the number of non-zero bits in the binary representation of the data being processed. This model closely correlates to real measurements in cases where the physical implementation technology consumes slightly more or less power depending on the logic value of each bit being stored or moved through the system. One example where this would apply is in a logic style where each latch in the system is pre-charged and only zero-bits are discharged on a clock edge. A higher-level example is the case of a processor performing a multiplication. It is conceivable that the implementation of a hardware multiplication would be related to the Hamming Weight of each of the multiplicands because multiplication is often implemented as a series of bitwise shifts and additions.
2.2.2 Hamming Distance

The Hamming Distance model is a second-order variant of the Hamming Weight, defined as the number of bits that are different between two data values. Thus, the Hamming Distance $HD$ between values $DataA$ and $DataB$ is shown in Figure 2.9 in terms of the Hamming Weight function $HW$.

$$HD(DataA, DataB) = HW(DataA \oplus DataB)$$

This model is useful in many semiconductor systems because it correlates well to the power signature of an internal data register changing from one value to another as described in the introduction.
2.3 Measurement Methods

A practical attack on a physical cryptosystem requires that the side-channel being attacked must be measured by some means. For a power analysis attack, this is typically accomplished by means of standard high-speed data acquisition of either the current passing through the power supply pins using a current-sense resistor, or the electromagnetic radiation emitted by the device using an EM probe. Since EM probes are much more expensive than passive voltage measurement probes, this thesis focuses on measuring instantaneous power consumption using a series resistor placed on the power supply pins of the device under test. Using a reasonably high-speed oscilloscope to measure the voltage across a current-sense resistor shunt on the power supply pin of the device allows for a low-cost set of attack hardware. This thesis makes use of an Agilent Infiniium MSO8104A oscilloscope capable of sampling at 4GSa/s with 1GHz of bandwidth, but according to Eisenbarth et al, a much more modest sampling rate of 20 MSa/s is sufficient for this type of attack [10], simply requiring more measurements to be collected.

If a shunt resistor is used to sense the power consumption of the device, it should be placed as near as possible to the power supply pin in order to avoid unnecessarily introducing noise into the measurement. The value of the resistor should be optimized to provide sufficient amplification for the oscilloscope to capture the waveform with acceptable resolution without dropping so much voltage that the device is operated outside its specified supply voltage tolerance, which could cause it to glitch and behave erratically.
Probes are placed on either side of the resistor in order to make a differential measurement of the voltage across the resistor. In this way, the current across the resistor, and thus into (or out of) the device’s power supply pins, can be measured. Another benefit of this differential measurement is that it filters out common-mode noise present in the circuit board due to high-speed switching of the device under test or other devices nearby on the board. Digital design rules normally consider this high-frequency switching current to be a noise source and mitigate it using a bypass capacitor between the supply pin and the ground pin of the device. In the case of tightly-spaced surface-mount circuit designs, this design rule could work to the advantage of the attacker, providing a minimally-invasive point at which to insert a shunt resistor to monitor the power on the power supply pin in cases where the device leads are not otherwise easily accessible, such as on a Ball-Grid Array (BGA) package.

2.4 Analysis Methods

Once a number of measurements have been taken, they are analyzed using statistical methods in order to determine how to most effectively launch an attack on the system. Often, the set of traces will have a common characteristic that represents a baseline operation profile of the algorithm. For example, a spike in current will signify a clock cycle and a repeating pattern of peaks will signify a round of encryption. In this way, the traces can be aligned either by visual inspection or by mathematical means such as a Fourier transform.
With a set of aligned traces, where corresponding samples of each trace represent the same step in the encryption process, more abstract analysis can be employed to draw out the signal from the noise. For example, by subtracting the mean from each of the corresponding samples, each trace becomes an essentially-flat residual trace, with only the data-dependent steps in the process presenting themselves as positive or negative peaks. As more traces are collected, the inherent noise in the measurement and alignment processes are cancelled out, leaving only the signal.

The final step is to decimate the residual trace so that a single data point can be correlated to each of the samples that will be generated by the leakage model during the attack process. Strictly speaking, this step is optional. In practice, though, it is very beneficial to boil down the raw data to only the most useful pieces so that the attack analysis can be carried out most efficiently by only processing the minimum data required to recover the secret key used for encryption.
Chapter 3

Core Framework

In keeping with the goals laid out for the project, SCARF is implemented as a package in the Ruby programming language [26], called a “gem”. As a Ruby gem, the software is easy to distribute as a single file that is aware of its own dependencies. Since the code is written in Ruby, an interpreted scripting language, it is readable and simple for a new researcher to learn how it works and make changes to it.

The gem installs a command-line interface, accessed with the scarf command, which can be used to access some of the accessory utilities. The main functionality of the SCARF gem is accessed using the normal require keyword in a Ruby script, as shown in the examples distributed with the gem. This allows user code written to use the SCARF framework to be tested using normal Ruby testing frameworks, which are numerous and widely-used in the Ruby community.
In addition to the core Ruby gem, the framework comes with some convenience scripts, written in the R scripting language for the R Project for Statistical Computing software package [23], to simplify the generation of plots of the output data from the analysis and attack phases. In order to use these scripts, R must be installed, as it is not distributed with SCARF. R is open-source software and available for download free-of-charge.

3.1 Architectural Overview

The following sections describe the high-level conceptual architecture of the SCARF tool. SCARF is designed primarily to be used as a data-processing pipeline for analyzing side-channel traces, both simulated and physically measured. In order to support a common pipeline of workflow steps, SCARF implements a set of data objects used to pass information between each workflow. This allows the workflows to be de-coupled from one another, simplifying the scope of each script that needs to be implemented by the researcher and facilitating re-use of common components.

3.2 Data Models

All of the data used by SCARF during the attack process is stored in a fast in-memory database using several domain-specific data structures. The data structures offered by SCARF allow the user to easily load side-channel traces and related data into the Trace model, group them together for batch processing using a Trace Group model, and collect ranking information about the progress of the attack using the Key Guess model. Each type of data model will be discussed in more detail in the following paragraphs.
3.2.1 Database Back-End

In order to provide an efficient data store that can be accessed heavily during the attack process, the Redis [24] database was selected. Redis is an open-source in-memory key-value store that offers atomic transactions on high-level data types, such as hash tables and sorted sets. SCARF makes use of these features to provide high performance while allowing multiple instances of the SCARF tool to work in parallel against the same database in a distributed attack model.

Since Redis is a key-value store but also natively supports high-level data structures, each data model used by SCARF must be designed in such a way that the required data can be easily queried and modified during the execution of the attack under investigation.

3.2.2 Trace

The Trace model offers storage for several convenient attributes in addition to the obvious need to store the actual samples of the leakage trace being attacked. It also allows for storage of the cryptographic key used, the plaintext used, the resulting ciphertext, and the residual trace samples calculated during the transformation phase and used during the attack phase.

These data are stored in the Redis database using a database key for each Trace model in the form Trace:<id>, where <id> is a unique identifier for a given Trace model instance. The unique identifiers are guaranteed to be unique by using an atomically incrementing counter stored in the database key called counters, which is a hash table of counter values where the Trace model uses the hash key traces. The Trace model is stored in the database as a hash table, allowing the encryption key, plaintext, ciphertext, trace samples, and residual samples to be accessed by name in the hash.
The encryption key, plaintext, and ciphertext are stored as simple numeric values in the database, but the trace samples and residual samples cannot be stored as a single numeric value because they represent a time-series of values. While Redis does offer a data structure for storing lists of values, it does not allow a list to be stored inside a hash. Also, since Redis stores its values as character strings, there would be a significant amount of storage overhead associated with storing the trace samples and residual samples this way as opposed to a binary format. As a result, the trace samples and residual samples are packed into a binary string as double-precision floating-point numbers for each sample. This allows significant flexibility in the range of measurement data that can be stored without overly complicating the storage solution. Once the samples are packed into a binary string, the string can be stored as a single value in the Trace model instance’s hash table.

3.2.3 Trace Group

The Trace Group model does not offer any additional attributes, but provides a way to atomically request a given number of traces. This is necessary in a situation where multiple SCARF processes are working in parallel, but each process needs to allocate a set of traces to be processed as a group. The Trace Group model instances identify themselves with a unique identifier using the same mechanism as the Trace model, but use the counter hash key `trace_groups` instead of `traces`. A Trace Group is able to safely allocate itself a set of traces using a feature of Redis that allows the unique ID of a trace to be atomically popped off the “Trace Queue” and added to the Trace Group without the possibility of a race condition causing the Trace to also be added to another Trace Group. Traces are added to the “Trace Queue” upon creation specifically for this purpose, but it is also possible to use this queue as a progress indicator, since the number of traces in the queue indicates the remaining unallocated work to be scheduled when a SCARF process becomes available.
3.2.4 Key Guess

The Key Guess model stores several attributes in the same way as the Trace model: the ID of the Trace Group associated with the Key Guess, the Phase of the attack at which this Key Guess was made, the actual value of the Key Guess, and a bitwise Mask of the specific bits in the Key Guess to be treated as being actively guessed. This allows for all of the relevant context data to be stored along with the partial key guess itself, which is useful for post-processing and analysis in addition to querying for sets of key guesses that are related to one another.

In addition to the simple properties that are stored with each Key Guess model instance, SCARF makes use of a more advanced feature of Redis to store the correlation values as a sorted set. In Redis, a sorted set is a data structure that allows values to be stored along with a sorting metric that will be automatically and efficiently kept sorted. In SCARF, the sorted set is used to store Key Guesses in buckets for each Trace Group and Phase pair, using a key of the form guesses:<group>:<phase>. Within the sorted set, the values that are stored use the form <key_guess>:<guess_mask> with a sorting metric equal to the correlation of the leakage model using the associated key guess with the residual trace samples. Redis offers the ability to increment the sorting metric by the specified amount for a given value, which allows SCARF to accumulate the sum of the correlation values for a Key Guess across the traces in the Trace Group.
3.3 Processing Steps

SCARF offers a suite of tools to assist researchers in evaluating the resistance of a cryptographic algorithm or implementation against side-channel attacks, with a focus on Differential Power Analysis. The following processing steps are carried out in order to process either simulated or measured side-channel leakage traces and determine their properties, particularly their susceptibility to attack.

3.3.1 Data Import

Before any data can be processed, it must be imported into a format that SCARF can understand and manipulate. In order to simplify the import process, SCARF offers a simple SCARF::Trace model, with attributes like plaintext, ciphertext, key, and samples. These attributes allow the researcher to easily read in data from an external source, such as a set of CSV files, and allow SCARF to understand and store it for later analysis and processing. This abstraction lets the researcher focus on the data rather than having to think about how it is stored, accessed, and linked together after it has been imported.

Capture Workflow

The first step in the pipeline is normally the Capture Workflow, shown in Figure 3.1. In this workflow, a Pattern Generator is used to generate Plaintext input data vectors, which are passed through the Cipher to produce Ciphertexts and Intermediate Values. The Intermediate Values are passed through a Leakage Model, which produces the Leakage Traces. Both the Ciphertext and Leakage Trace are stored in a Database, usually associated with the Plaintext and/or Ciphertext used to generate them.
The Pattern Generator can be implemented a number of ways. For the case of a simulated attack, the Pattern Generator could simply be a random number generator that creates a file containing some number of Plaintexts. For an attack on physical measurements, the pattern generator could be a physical piece of test equipment that outputs the appropriate waveforms required to stimulate the device under test.

The Cipher is simply an implementation of the cryptographic system under test. For simulation, it is normally a program that converts a plaintext to a ciphertext and a set of intermediate values that can be processed by a leakage model. For physical measurements, the cipher is the physical cryptosystem itself, where the intermediate values are presented as a side-channel to be analyzed, such as the instantaneous current on the power supply pins.

The Leakage Model is one of the primary points of research for side-channel analysis of a cryptosystem because it is often tailored to a particular attack and makes it possible for the simulated attack to be performed nearly identically to an attack on physical measurements.
3.3.2 Trace Analysis and Transformation

Once the data has been imported into the SCARF tool, the raw traces can be plotted individually or in groups to allow the properties of the data set to be explored for patterns. Additionally, the data set as a whole can be inspected using derived traces showing the statistical properties of all traces together, such as a mean trace or standard deviation trace.

These derived traces can be used to determine how to decimate the raw traces down to only the sensitive information of interest to the attack. Once the sensitive information has been identified and extracted by a transformation step, the so-called “Residual Traces” can be re-analyzed to inspect the properties of the sensitive data after removing the irrelevant samples from the traces.

Analyze Workflow

The Analyze Workflow, shown in Figure 3.2, is where the researcher is able to study and experiment with the leakage model developed in the Capture Workflow. Normally, the Leakage Traces are passed through one or more scripts to produce derived traces such as Mean Traces or Standard Deviation Traces, which can be used for visual inspection to help the researcher determine where the Sensitive Information is present in a set of traces.

![Figure 3.2: SCARF Analyze Workflow](image-url)
Transform Workflow

After the Analyze Workflow has identified the interesting parts of the measured, simulated, or derived traces, the Leakage Traces are passed through a Transform operation as shown in Figure 3.3 to distill a Residual Trace containing only the information that is useful for an attack. For physical measurements, this could include alignment, decimation, filtering, or more complex operations. For simulated traces, the researcher may have already designed the Leakage Model such that it only generates useful information and does not require transformation.

![Figure 3.3: SCARF Transform Workflow](image)

3.3.3 Attack

When the sensitive data matching the leakage model has been isolated, an attack can be formulated. SCARF offers abstractions for Traces, Trace Groups, Residuals, and Partial Key Guesses to aide the researcher in building an attack on the specific algorithm or implementation being studied. In addition, several utility functions are included with SCARF to perform calculations such as Pearson’s Correlation Coefficient, Hamming Distance, and Hamming Weight, which are commonly used in side-channel attacks and specifically in power-analysis attacks. These data models and utility functions allow the specifics of the attack process to be defined without having to implement the low-level details of performing these operations on each trace.
**Attack Workflow**

After the data has been prepared by the Transform Workflow, the main focus of side-channel analysis can be performed by the Attack Workflow, shown in Figure 3.4. In the *Attack Workflow*, the sensitive information in the *Residual Traces* is processed by a *Worker* to generate a set of *Partial Key Guesses*. In a scenario where the attacker has access to the Ciphertext and Side Channel (represented by the Residual Trace), the Partial Key Guesses are used along with the Ciphertexts to run the Cipher in reverse, revealing the Intermediate Values that would have been produced during the encryption that produced the Ciphertext if the Partial Key Guess were correct. The Intermediate Values are passed through the Leakage Model just as they were in the Capture Workflow and finally correlated against the Residual Traces produced by the Transform Workflow. The correlation metrics for each Partial Key Guess are incorporated into a Database of *Ranked Key Guesses*, which is iteratively refined until the complete encryption key can be recovered.

![Figure 3.4: SCARF Attack Workflow](image-url)
3.3.4 Post-Process

During the attack, the researcher is given feedback on the status of the attack process as it progresses, but the data is also kept in the database throughout the various phases of the attack. After the attack is complete, this data can be post-processed to study its performance and effectiveness in greater detail. This allows for the creation of plots showing the attack’s effectiveness using some susceptibility metric versus number of traces used or some other key parameter of the attack.

Post-Process Workflow

Following the Attack Workflow, the Post-Process Workflow, shown in Figure 3.5, is used to characterize the effectiveness of the attack. This workflow allows the researcher to analyze the database of Ranked Key Guesses for each phase of the attack to generate plots such as the Distribution of Correlation Metrics and the Partial Success Rate versus number of traces processed.

![SCARF Post-Process Workflow](image)

Figure 3.5: SCARF Post-Process Workflow
Chapter 4

Simplified DES Example

The first example attack presented in this thesis is a known-plaintext attack against a software implementation of a simplified version of the Data Encryption Standard (DES)[7]. The simplifications made to standard DES for this example are that only the first round of encryption was considered and the key-scheduling was ignored. By only implementing a single encryption round, the implementation of the attack is somewhat simpler but still demonstrates that a DES sub-key can be successfully recovered. Since only the first round of encryption is being attacked, the key-scheduling algorithm is ignored by simply choosing a 48-bit sub-key directly since only one sub-key is required.
4.1 The Modified DES Cipher

A block diagram of the Modified DES cipher used in this example is shown in Figure 4.1. It consists of the same set of permutation and substitution operations as the standard DES cipher, but with several simplifications as mentioned above. The 64-bit plaintext is first permuted by the Initial Permutation function, which shuffles the bits around before performing an Expansion operation on the 32 Least-Significant Bits (LSBs) of the resulting permuted value to produce a 48-bit value. The Expansion function works by copying each input bit to one or two output bits such that the duplicated bits cause a 32-bit input value to become a 48-bit output value. The 48-bit value is XORed with the 48-bit Round Sub-Key, which would normally be produced by the DES Key Schedule algorithm but in this example is simply chosen directly. After the Sub-Key has been mixed with the 48-bit value, the result is passed through eight Substitution Boxes (SBOXes) in 6-bit, non-overlapping chunks. Each SBOX produces a 4-bit output from a 6-bit input, thereby reducing the 48-bit value back to a 32-bit value, since there are eight SBOXes. These eight non-linear look-up tables are each defined in the DES specification. The 32-bit SBOX output is permuted again by the Permutation Box (PBOX) before being XORed with the 32 Most-Significant Bits (MSBs) of the original plaintext input value. Finally, 64-bit ciphertext output value is produced by the Final Permutation (FP) using the 32 LSBs of the original plaintext input as the 32 MSBs and the result of the 32-bit XOR as the LSBs of the input to the FP operation. In standard DES, the Expansion, 48-bit Sub-Key XOR, SBOXes, PBOX, and 32-bit XOR would be performed 16 times. In this simplified example, only the first round of encryption is simulated to demonstrate that the first-round sub-key can be recovered under the conditions described.
Figure 4.1: Block Diagram of the Modified DES Algorithm
4.2 Attacks in the Literature

A variety of attacks have been proposed and implemented against DES throughout its history. The first and most basic type of attack is the brute-force method. Due to its relatively short 56-bit key length, DES is more prone to brute-force attacks than some other algorithms. In a brute-force attack, every possible key is tested in turn. For a 56-bit key, there are $2^{56} = 72,057,594,037,927,936$ possible keys to test. Several efforts have demonstrated that, using specialized distributed computing and hardware, this type of attack is practical. In 1997, a DES-cracking contest sponsored by RSA security was won by distributing the calculations across thousands of computers and running for 96 days [28]. In 1998, the Electronic Frontier Foundation [8] demonstrated its DES Cracker [11] based on a custom integrated circuit, capable of recovering a key in an average of 4.5 days at a cost of less than $250,000. In 2006, the Ruhr University Bochum and the University of Kiel in Germany demonstrated COPACOBANA [17], with similar performance to EFF’s DES Cracker, but significantly lower cost due to the use of off-the-shelf Field-Programmable Gate Arrays (FPGAs) instead of a custom silicon design. This solution also has the advantage of being reprogrammable and thus useful for attacks on algorithms other than DES.

In addition to these brute-force attacks, several techniques have been developed to reduce the complexity of the attack such as Differential Cryptanalysis [3], Linear Cryptanalysis [20], and Improved Davies’ Attack Attack [2]. Due to the continued use of DES in Smart Card applications, more recent research has shown DES to be prone to a variety of classes of effective side-channel attacks, including, Simple Power Analysis (SPA) [18], Differential Power Analysis (DPA) [21], Electro-Magnetic (EM) Analysis [1], and Collision Attacks [27]. In the following sections, we will demonstrate a DPA attack against a software simulation of the first round of DES using the SCARF tool.
4.3 Leakage Model

For this example, the leakage model is that the attacker is able to measure the Hamming Weight of the 32-bit SBOX output value using a side-channel such as the instantaneous power consumption during the substitution operation. This type of leakage is common in dynamic CMOS technologies where the characteristics of the pre-charge and subsequent discharge of logic gates and data buses can be inferred.

4.4 Simulated Attack Process

In this attack, it is assumed that the attacker knows the plaintext values being encrypted but not the resulting ciphertext values. Even though the ciphertext is not known, the 48-bit Sub-Key can be recovered by performing a simulated encryption on the known plaintext using a guessed key. The attacker attempts to guess the 48-bit key in 6-bit chunks corresponding to the 6-bit inputs of the eight SBOXes. Ignoring the other seven SBOXes, the attacker is able to correlate a simulated Hamming Weight against the measured value for each given key guess. The guess that best matches the measurements across all of the measured encryption traces with different plaintext values is used as the final guess for those six bits of the key. In this way, the full 48-bit key can be recovered, six bits at a time.

4.4.1 Data Import

The attack begins by capturing the simulated plaintext and leakage trace data. This is accomplished with a brief `simulate.rb` Ruby script to invoke an implementation of the DES algorithm that has been instrumented to calculate the Hamming Weight of the 32-bit SBOX outputs to match the proposed leakage model. Since this attack is only concerned with a single round of encryption, the leakage trace only contains one data point.
4.4.2 Trace Analysis

Since each leakage trace consists of a single data point, there is not very much trace analysis to be done since the mean trace and standard deviation trace are simply the scalar mean and standard deviation of the single point. The distribution of the data can be seen in Figure 4.2, which is the result of 10000 simulated encryptions using random plaintext values and a fixed key. Since the number of logic-high bits in a 32-bit value is effectively a random value between 0 and 32, the Hamming Weight follows a normal distribution with a mean value of 16 as is observed in the figure.

![Distribution of Sample Values](image)

Figure 4.2: Distribution of SBOX Output Hamming Weights from 10000 Encryptions

4.4.3 Trace Transformation

Because this attack is simulated, there is no measurement noise to mitigate and because each trace consists of only a single point, there is no trace compression to be performed. Thus, the transformation step can be skipped altogether because the data is already in an appropriate form to carry out the attack.
4.4.4 Attack

The attack progresses in eight phases, each phase targeting a different SBOX to recover six bits of the key. This is accomplished by exhaustively searching the 6-bit key space, consisting of $2^6 = 64$ partial key guesses. The leakage metric is calculated for each known plaintext value using each partial key guess and correlated against the corresponding leakage traces, which would have been captured ahead of time by the attacker. The partial key guesses are then sorted by their correlation metrics and the top candidate is chosen to propagate to the next phase. In this example attack, only the top candidate is selected, but several of the top guesses could be selected in general. In this case, a ninth phase would be required to select among the permutations of top candidates for each of the eight SBOXes.

In each phase, the top candidate partial key guess from the previous phase is propagated and used for the bits of the key corresponding to the previously-guessed SBOX, but only the output of the single SBOX under attack in that phase is correlated. This is done to reduce the noise generated by incorrect key bits in successive phases.

4.5 Simulated Attack Results

The following plots, automatically generated by the SCARF tool, show the correlation metric and number of correct key bits guessed at the end of each phase of the attack. The darker line with rings around each data point show the relevant information corresponding to the actual key. The lighter data points show the distribution of other candidates for each phase. In Figure 4.4, it is clear that 16 traces are not sufficient to recover the whole 48-bit sub-key. However, since the performance of the attack against each SBOX depends on the non-linearity between the plaintext data and key being processed and its leakage metric, so in the case of Phases 1 and 5, the correct partial keys were easily
identified among the other candidates. Using 64 traces (Figure 4.5), some of the SBOXes were identifiable by a narrow margin, but Phases 4 and 5 failed to correlate sufficiently with the correct partial key. With 128 traces (Figure 4.6), the full 48-bit key was recovered, but it was a very close call in Phase 5, where there is the grey dot of another candidate key visible in the center of the black ring from of the correct partial key guess. Moving on to 256 (Figure 4.7) and 512 (Figure 4.8) traces, it is clear that the correct key has been found for all SBOXes with sufficient margin. Performance benchmark results, shown in Figure 4.3 indicate that the time required for the entire attack process scales linearly with the number of traces processed, consuming approximately 0.25 seconds per trace\(^1\).

\(^1\)Performance benchmark data was collected using Ruby 1.9.2p290 on an Apple MacBook Pro with a 2.5GHz Core 2 Duo processor and 4GB of 667MHz DDR2 SDRAM.
Chapter 4: Simplified DES Example

Figure 4.4: DES Guess Metrics for 16 Simulated Traces
Figure 4.5: DES Guess Metrics for 64 Simulated Traces
Figure 4.6: DES Guess Metrics for 128 Simulated Traces
Figure 4.7: DES Guess Metrics for 256 Simulated Traces
Figure 4.8: DES Guess Metrics for 512 Simulated Traces
Chapter 5

KeeLoq Example

The second example attack that will be presented in this thesis is a practical Differential Power Analysis Attack on physical hardware \(^1\) implementing Microchip’s KeeLoq\(^\text{®}\) technology \[13\], often used in remote keyless-entry applications for vehicles and garage door openers. The KeeLoq system consists of a transmitter and receiver pair with a shared key such that a short encrypted command can be sent from a simple, low-power transmitter to be decrypted, decoded, and validated by the receiver before the command is executed. In order to prevent replay attacks, which are trivial to perform on earlier fixed-code systems found in garage door openers, a so-called “Hopping Code” is used to ensure that a ciphertext will only be validated by the receiver once, preventing an eavesdropper from recording and replaying a successful ciphertext.

\(^1\) Preliminary data collection for this attack was funded in part under a collaborative project described in an internal white paper \[5\].
The physical hardware under test in this example is Microchip’s HCS301 KeeLoq Encoder [12], which is a highly-integrated system in a small package that implements the following major functions:

- an internal oscillator for clocking,
- a serial interface for device configuration,
- discrete inputs pins to be used for buttons,
- internal memory used to store configuration information such as:
  - secret key,
  - device ID,
  - initialization vector,
- an implementation of the KeeLoq cipher used to encrypt commands using the stored key and hopping code,
- a discrete LED driver output pin to acknowledge the user’s command, and
- a Pulse-Width Modulated (PWM) data output, used for radio transmission.
5.1 The KeeLoq Cipher

The KeeLoq cipher is implemented using a simple Non-Linear Feedback Shift Register (NLFSR) to permute a 32-bit data register through 528 rounds of encryption using a 64-bit key. Figure 5.1 shows all the details of the KeeLoq encryption algorithm. The cipher is implemented using two shift registers; a 32-bit register for the data being processed and a 64-bit register for the encryption key. For each of the 528 rounds of encryption, five bits of the data register are passed into the NLFSR function which can be implemented by the lookup table represented by the hexadecimal string 0x3A5C742E. The result of this non-linear function is then combined linearly with two other data bits and one key bit to produce a single bit that is shifted into the most-significant bit of the data register on each round. During each round, the 64-bit key is simply rotated linearly in a separate shift register. Because 528 encryption rounds are used with a 64-bit key being rotated one bit per round, the encryption key ends up right-shifted shifted by 16 bits at the end of the encryption process and must be reset for the next encryption. A detailed cryptanalysis of the KeeLoq algorithm is published in [4].

In practice, this 32-bit block cipher is typically used in remote keyless entry systems to encrypt a predictable 32-bit plaintext message consisting of an incrementing counter value, a static identification value (typically derived from the device serial number), and the status of the buttons on the remote [12]. Along with the encrypted portion, a typical KeeLoq message contains a fixed, un-encrypted portion containing the device serial number and button status.
5.2 Attacks in the Literature

Several attacks have been proposed and demonstrated against the KeeLoq cryptosystem. Early attacks were based on sliding and algebraic techniques in order to recover the encryption key based on a database of known plaintext-ciphertext pairs [6], [4], [14]. Based on these works, more advanced power-analysis side-channel attacks were described [15], [10], [9], attacking both software and hardware implementations of the KeeLoq algorithm.
5.3 Leakage Model

Due to the simple nature of the KeeLoq cipher, the leakage model used for this example is simply the Hamming Distance of the data register between successive rounds of encryption. As described in Section 5.1, each round of encryption depends on a single bit of the 64-bit key, so only 64 of the 528 data points are required to recover the full key, though more samples could be used to increase the confidence level of the key guess. In this example, leakage samples are generated in small groups based on the Hamming Distance of the data register between the next few rounds of encryption for each of the candidate partial key guesses.

5.4 Simulated Attack

The first step in evaluating an attack using a given leakage model is to carry out the attack in simulation so that the projected data will exactly match the leakage model. This allows the effectiveness of the attack to be evaluated quickly without having to design and use a physical trace acquisition setup to collect sufficient data for an attack. Simulation also eliminates the effects of noise on the measurements. If the attack is not effective in simulation, it allows the researcher to tweak the leakage model or approach the system in a more effective way rather than spending a lot of time trying to reduce measurement noise or improve trace alignment only to find that the attack is simply not feasible even with perfect measurements.

For the KeeLoq attack, the simulated power traces are generated by simply performing a simulated encryption on an incrementing initialization vector. This is not required for the attack to be successful, but it is a sensible default since we know that the HCS301 device under test does in fact operate this way [12]. The simulated encryption is used to
generate traces that exactly match what the leakage model would predict to see during the encryption being simulated. This allows the effectiveness of the attack to be evaluated in an ideal case without noise or error being added to the signals. These simulated traces are stored in the SCARF database just as measured traces would be, to allow the rest of the framework to be used in a simulated attack on the traces. A sampling of several such traces superimposed is shown at the top of Figure 5.2, with a single trace shown in black and others in grey to demonstrate both the range of values and the trend of a particular trace over time.

Due to the way the KeeLoq cipher uses a Non-Linear Feedback Shift Register (NLFSR) to encrypt the ciphertext, only one bit of the data register changes each encryption round when the data value is shifted. This is the reason for the regular stair-step shape of the simulated leakage trace in the top plot of Figure 5.2, changing the Hamming Distance by a maximum of one bit for each encryption round. If we plot the distribution of the difference of successive points in the trace, we can clearly see this property, shown in the bottom plot of Figure 5.2. The three distinct lobes in the distribution represent a decreasing, constant, and increasing Hamming Distance, respectively.
Figure 5.2: Simulated KeeLoq Leakage Trace Samples
Chapter 5: KeeLoq Example

After the traces have been generated and stored in the database during the simulate phase, they are processed by the transform phase to produce a residual trace, which is stored in the database for each simulated trace. For the KeeLoq example, the mean value of the corresponding samples in each simulated trace is calculated, resulting in a mean trace $M$, where $M_i$ is the mean value of the samples at position $i$ in all of the traces. To generate a residual $R$ for each trace $T$, the mean trace is subtracted from the measured trace such that $R_i = T_i - M_i$. The purpose of calculating the residual by subtracting the mean is that this type of attack is using the information encoded in the variations of the power consumption based on the data being processed. By subtracting the mean, we remove the components of the power trace that are not data-dependent so that they will not skew the correlation result.

After several residual traces are stored in the database, the main attack phase can begin. In the KeeLoq example, residuals are processed in independent groups, allowing for multiple processors to be used in the attack, as long as each has access to the database of residuals and partial key guesses used during the attack process. The attack proceeds by successively guessing several key bits at a time. Since KeeLoq uses one successive key bit per encryption round, this attack exhaustively searches the possible partial key guesses corresponding to the last few rounds of encryption and then correlates the leakage model of each guess against the samples from the actual or simulated encryption.

In the case of the simulated attack, the leakage model and the simulated traces are identical since the leakage model was used to generate the simulated traces. It is, however, important to note that this perfect match between the leakage model and the simulated trace does not guarantee that the key can be trivially recovered from a single simulated trace. For a small subset of the key bits, it is possible that there are multiple keys that will generate an identical or nearly identical power signature for certain plaintext
values since the power signature is influenced both by the key and the data being processed. When more traces are considered together, the probability of collision between signatures decreases increasing the confidence that the correlation between leakage and measurement accurately predicts the true key.

In order to deal with this issue, the KeeLoq example makes use of SCARF’s Key Guess object, which allows a partial key to be guessed by specifying a guess mask. The key guesses are made in a 12-bit-wide window that slides four bits to the right (that is, toward the least-significant bits) for each phase of the attack. The result is that four new bits of the key are exhaustively guessed in each phase of the attack, but the previous eight bits are re-confirmed during the correlation. By attacking four bits of the key at a time, we use information from four rounds of encryption since each round of KeeLoq encryption depends on only one key bit. We perform the attack in four-round batches because we can determine which of 16 guesses are most likely to be correct for each four-bit chunk more reliably than we can determine each single bit in isolation. This algorithm offers a form of back-tracking because each guessed key bit in KeeLoq depends on all of the previous bits of the guessed key. This makes the attack more resilient to noise because we can skip over some key bits that do not correlate properly as long as the other bits in the sub-key guess cause the leakage to correlate well overall.

Since we use the Hamming Distance model of the KeeLoq cipher for this example, each sample is either equal to the previous sample or exactly one unit greater or less. If more key bits are guessed, the likelihood of collision decreases, but at the cost of exponentially more key guesses being required to increase the confidence that the correct key bits have been found. This guessing and re-verifying process has an efficiency benefit because only 16 refined key guesses are introduced for each of the top guessing in the previous phase, but some of the previously-guessed key bits are also considered in the correlation, eliminating
the false correlation that occurs when only four samples are correlated in isolation. This helps to ensure that forward progress is maintained by preventing an incorrect partial key guess with a high-correlation to direct the search down the wrong path.

As the attack progresses, key guesses are accumulated in the database and combined across each residual trace in the trace group. This allows the correct key to be iteratively recovered from a set of traces processed in isolation. Using multiple processors or even separate computers on a network to process several groups in parallel, the results can be combined to process the full data set more quickly. The attack continues until the residual traces in a group are exhausted, reporting their correlation results throughout the process. The correlation metrics give the user an indication of how confident the SCARF tool is that it has correctly guessed the partial key guess in that particular phase. With very few traces, the first few bits of the key are recovered with high confidence. Then, the successive traces take advantage of the results of the earlier traces to make better guesses, resulting in additional key bits being guessed with high confidence. This continues until the full key is recovered with high confidence.

5.4.1 Simulated Attack Results

In the simulated attack on KeeLoq, the key can be recovered with high confidence using only one or two traces, depending on the key and initialization vector used for the attack. Once the attack has settled on a particular guess for the partial key guess for a phase of the attack, it trends toward higher correlation as additional residual traces re-confirm the result. For the case where only one trace is used, there may be several partial key guesses that correlate perfectly with some subset of the trace, making it impossible to identify the true key, though it is among this group of perfectly-correlated guesses. When taking multiple traces into account, however, this is no longer a concern because only the
correct guess will correlate well with all of the traces in the group. Figure 5.4 shows the correlation and partial success metrics for two traces while Figure 5.5 shows the same for 50 traces. Both figures demonstrate that the key is fully recovered (Partial Success Rate reaches 64 bits, the full length of the key), but with more traces, it is easier to distinguish the correct guess from the incorrect guesses due to the larger gap between the correlation metrics. This is why the attack is more likely to succeed when more traces are considered.

Performance benchmark results, shown in Figure 5.3 indicate that the time required for the entire attack process scales linearly with the number of traces processed, consuming less than 3 seconds per trace\(^2\).

\(^2\)Performance benchmark data was collected using Ruby 1.9.2p290 on an Apple MacBook Pro with a 2.5GHz Core 2 Duo processor and 4GB of 667MHz DDR2 SDRAM.
Figure 5.4: Keeloq Guess Metrics for 2 Simulated Traces
Figure 5.5: Keeloq Guess Metrics for 50 Simulated Traces
5.5 Hardware Attack

Based on the successful attack on the software simulation of the KeeLoq algorithm, we perform the following attack on physical KeeLoq hardware. Based on previous research in the literature [9], we know that the overall power signature of an HCS-family KeeLoq encryption device will look like Figure 5.6(a). The major functions of the device are evident from the power trace: power-on and initial calculations, writing updated values to internal EEPROM, KeeLoq encryption, and digital transmission. Figure 5.6(b) shows a detailed view of only the encryption phase of interest to the attack.

![Figure 5.6: Sample Keeloq Trace [9]](image)

(a) From power up to start sending

(b) Encryption part

Figure 5.6: Sample Keeloq Trace [9]
5.5.1 Experimental Setup

The device under test in this example is the Microchip HCS301-I/P, an integrated KeeLoq code hopping encoder device in an 8-pin DIP package. In order to configure and control the KeeLoq encoder device, an Agilent 16823A logic analyzer with built-in pattern generator was used to control the I/O in an automated way. An Agilent Infiniium MSO8104A mixed-signal oscilloscope was used both to capture the encoded data output from the device on a digital measurement channel and to capture the power traces on an analog channel. This experimental setup is shown in Figure 5.7.
Figure 5.8: Keeloq Target Board Schematic

A simple, low-cost target board was developed in order to interface with and measure the HCS301 KeeLoq encoder device. The schematic for the board is shown in Figure 5.8 and a photograph of the board is shown in Figure 5.9. The target board is powered directly from one of the pattern generator’s output so that the device can be easily switched on and off using the same voltage as the data interfaces.
The device is powered by the pattern generator through connectors J2 (VSS) and J3 (VDD). The VSS pin is connected to the J3 connector input through resistor R2, which acts as a current-sense resistor for collecting power traces with the oscilloscope. In this example, a 150 Ohm resistor is used to convert the current used by the device (on the order of 1 mA) into a voltage on the order of 150 mV during peak current draw. By inspection, it was determined that during encryption, the power trace measurement could be captured by setting the offset to 100 mV and the full-scale range to 10 mV, allowing the peaks and valleys of the power trace to be inspected during encryption.

In addition to the primary input power connectors J2 and J3, connectors J7, J8, and J9 are used as additional power connection points by externally connecting J2 to J7 (which is also connected to J9) and J3 to J8. This offers multiple places to connect equipment in order to share a common signal ground between the pattern generator and oscilloscope.
The target board is designed so that all relevant inputs and outputs are accessible from connectors, with inputs on the left side of the board and outputs on the right side. A slight complication to this strategy is that the PWM pin is used as an input during programming and an output during normal operation. This issue was mitigated by connecting the configuration input (J10) to the PWM pin through a series resistor (R3), which allows the PWM pin to be controlled when it is in high-impedance input mode for configuration, but does not contend with the output when in normal operation, allowing the PWM output to still be read on the output connector (J5).

During configuration, a serial clock is provided on the S2 pin through connector J11 and serial data on the PWM pin through connector J10. The device is configured as described in the following “Device Configuration” section, according to the procedures described in the HCS301 datasheet.

After configuration and verification is completed, the device is power-cycled to return it to normal operating mode. In normal operating mode, the device is commanded by driving the S0 pin through connector J1 to simulate a key-press one the “S0” button in a KeeLoq system. The falling edge of the LED pin is captured on connector J4 to trigger the capture of the power trace on the VSS pin through connector J6. The encrypted data is subsequently collected from the PWM pin on connector J5.
Chapter 5: KeeLoq Example

5.5.2 Device Configuration

Since the attack we are performing is based on a correlation between the dynamic power consumption of the chip and the corresponding cipher text, the input patterns will be used to cause the KeeLoq encoder chip to continually perform encryptions on a fixed schedule, so that we can capture the output using the oscilloscope.

In order to configure the HCS301 device, the Agilent pattern generator was configured as follows to output the necessary patterns to perform the device-programming operations.

1. Set the output period to 1 microsecond since the timescale for programming the HCS301 is in the microsecond range.

2. Program the HCS301 chip with a known configuration so that we can verify the attack at the end.
3. The configuration data programmed into the chip for this example is as follows:

<table>
<thead>
<tr>
<th>Word</th>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>KEY_0</td>
<td>0x5110</td>
<td>64-bit encryption key, word 0 (LSBs)</td>
</tr>
<tr>
<td>1</td>
<td>KEY_1</td>
<td>0x619B</td>
<td>64-bit encryption key, word 1</td>
</tr>
<tr>
<td>2</td>
<td>KEY_2</td>
<td>0xB22A</td>
<td>64-bit encryption key, word 2</td>
</tr>
<tr>
<td>3</td>
<td>KEY_3</td>
<td>0x7446</td>
<td>64-bit encryption key, word 3 (MSBs)</td>
</tr>
<tr>
<td>4</td>
<td>SYNC</td>
<td>0x157C</td>
<td>16-bit synchronization value</td>
</tr>
<tr>
<td>5</td>
<td>RESERVED</td>
<td>0x0000</td>
<td>Reserved, set to 0x0000</td>
</tr>
<tr>
<td>6</td>
<td>SER_0</td>
<td>0x0B72</td>
<td>Device Serial Number, word 0 (LSBs)</td>
</tr>
<tr>
<td>7</td>
<td>SER_1</td>
<td>0x38D8</td>
<td>Device Serial Number, word 1 (MSBs)</td>
</tr>
<tr>
<td>8</td>
<td>SEED_0</td>
<td>0xBCA9</td>
<td>SEED Value, word 0 (LSBs)</td>
</tr>
<tr>
<td>9</td>
<td>SEED_1</td>
<td>0x4DDF</td>
<td>SEED Value, word 1 (MSBs)</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
<td>0x0000</td>
<td>Reserved, set to 0x0000</td>
</tr>
<tr>
<td>11</td>
<td>CONFIG</td>
<td>0x0110</td>
<td>Reserved, set to 0x0000</td>
</tr>
</tbody>
</table>
The CONFIG word has the following format:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Discrimination Bit 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Discrimination Bit 1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Discrimination Bit 2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>Discrimination Bit 3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Discrimination Bit 4</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>Discrimination Bit 5</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>Discrimination Bit 6</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>Discrimination Bit 7</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>Discrimination Bit 8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>Discrimination Bit 9</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>Overflow Bit 0 (OVR0)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>Overflow Bit 1 (OVR1)</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>Low-Voltage Trip Point Select (VLOW SEL)</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>Baud rate Select Bit 0 (BSL0)</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>Baud rate Select Bit 1 (BSL1)</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>Reserved, set to 0</td>
</tr>
</tbody>
</table>
4. Cycle the chip power to complete the programming process.

5. Repeat indefinitely:
   
   (a) Wait 1 second
   
   (b) Simulate a button-press on the KeeLoq chip for 5 seconds
   
   (c) Release the button-press
   
   The pattern generator is loaded with this configuration, then set to run continuously through the menu options: “Run/Stop” → “Output” → “Run Repetitive”
5.5.3 Trace Capture and Alignment

Once the input patterns are configured and running continuously, the oscilloscope is used to capture the encrypted output data and the associated encryption power traces. This entire process is performed through a standard Transmission Control Protocol (TCP) connection between the capturing computer and the oscilloscope using a standard wired Ethernet network, in this case running at 100 Mbps.

When one of the buttons is pressed on a KeeLoq device, the encrypted data is transmitted repeatedly until the button is released. However, the encryption only takes place once, just before the first transmission. Because of this, it is important to capture the power trace associated with the encryption process right away, and continue to hold down the button until the associated cipher-text can be captured as well, since the same oscilloscope is being used to capture both waveforms and each capture requires a drastically different time scale. This is the reason for the 5-second delay in the main loop of the input patterns described above.

One very important point to be made about this KeeLoq example is that several KeeLoq encoder devices were evaluated: Microchip’s HCS101, HCS201, and HCS301. The interface to each of these devices is nearly identical, but the HCS301 offers an additional feature that made it a much easier target for this type of attack. Because the HCS301 offers an LED output driver to signify a confirmation of a button-press and subsequent command transmission, this signal was used to trigger the oscilloscope to capture the power trace. In the other devices, since the oscillator is embedded in the encoder device, the only reference available is the encrypted data output, which is not perfectly synchronized with the encryption process like the LED indicator is. Using this LED indicator as an additional side-channel for information leakage, it is possible to trivially synchronize to the internal oscillator with no additional alignment step.
Capturing the Power Trace

A MATLAB [19] script is used to configure the oscilloscope over a TCP socket such that it will trigger exactly at the end of the encryption process and transfer the captured trace over the Ethernet network to the controlling computer. Only the encryption portion of the power trace is captured due to the memory limits of the oscilloscope. The characteristics of 250 captured traces are shown in Figure 5.10, which shows an example of a single trace, the mean, and the standard deviation of the 250 traces for each point in time. From this figure, the vertical spikes in the power trace represent the clock cycles of the internal oscillator, one clock cycle per round of KeeLoq encryption. This trace captures many more clock cycles than are required for the attack; only the final 64 peaks are required to perform the attack because one key bit can be inferred from each round of encryption. Due to the jitter of the internal oscillator, the traces do not align perfectly, and the alignment gets worse for samples farther from the trigger point (which is at the end of the encryption process, on the right side of the plot in the figure). This is evidenced by the decreasing variance trace caused by the sharp peaks being more and more aligned as they near the trigger point. At the trigger point, the peaks are perfectly aligned because the oscilloscope is set to trigger on the edge of the LED control output, which is assumed to occur at a clock edge.
Figure 5.10: Keeloq Trace Characteristics - Sample Trace, Mean, and Standard Deviation
Capturing the Data Trace

After capturing the power trace, another MATLAB script is responsible for capturing the ciphertext that is the digital result of the KeeLoq encryption that has been performed. The commands in this script are similar to those in the power trace capture script, except that the data trace capture script is configured to trigger at the beginning of the encrypted data transmission phase, which is repeated several times while the button-press is being continuously held by the pattern generator. The oscilloscope used in this example is limited to only being able to transfer the full 32MB of internal memory for an acquisition of the digital pod where the cipher-text serial data stream is being captured, rather than being able to capture only the one required channel. Thus, it takes a few seconds to transfer the captured serial data to the host computer before moving on to the next trace. The combined time it takes for both the power trace and the data trace to transfer over the network comprise several seconds, which is the reason for the 5-second delay in the input patterns described previously. This delay could conceivably be minimized by implementing a software filter capable of decoding the serial data stream in the oscilloscope in order to remove the bottleneck of having to transfer the raw data to the host computer for further processing.

After the raw data trace has been captured, the high-resolution serial waveform representing the PWM-encoded transmission is down-sampled and converted into a simple binary string representing the transmitted data. This is accomplished by measuring the width of each pulse in the serial pulse train. According to the KeeLoq datasheet, the wider pulses represent binary zeros and the narrower pulses represent binary ones.
5.5.4 Trace Analysis and Compression

Looking closely at the last few clock cycles of the encryption, it is evident in Figure 5.11 from the standard deviation trace that the data-dependent samples coincide with the peaks in the power trace. More specifically, each alternate peak in the trace corresponds to a peak in the standard deviation trace. In order to distill the trace to only the significant peak samples, the distance between peaks is estimated by inspection, then a local maximum value is successively chosen in the neighborhood of the samples the expected distance from the previous peak value until the required 64 peaks are found. This algorithm is slightly more complicated than straight-forward down-sampling, but helps overcome the clock jitter by re-synchronizing to each peak before searching for the next peak and also requires a less-accurate estimate of the distance between peaks.
Figure 5.11: Keeloq Measured Trace Characteristics - Detail
5.5.5 Attack on Measured Traces

Once the significant peaks are identified in the raw trace, only those peaks are imported into the SCARF database to save processing time. From there, the attack on the physical measurements is carried out identically to the simulated attack.

A sample of several residual traces from the measured data set is shown in the top of Figure 5.12. Due to measurement noise and other factors not accounted-for in the attack model, these measured traces do not exhibit the same smoothly-flowing stair-step pattern as observed in the simulated traces in Figure 5.2.

Plotting the distribution of the differences between adjacent peaks in the measured leakage traces, we confirm in the bottom plot of Figure 5.12 that the distribution is not as clean and predictable as the simulated results in Figure 5.2. The results improve significantly when the same plots are generated for the residual trace, which has had the mean trace subtracted, as shown in Figure 5.13. With the mean subtracted, the distribution shows that there are at least two clearly-visible lobes where we expect three.
Figure 5.12: Keeloq Measured Leakage Trace Samples
Figure 5.13: KeeLoq Measured Residual Samples
In spite of the poorer quality of the measured traces compared to the simulated traces, the secret key was still recovered using on the order of 25 measured traces, validating the attack model. Figure 5.14 shows the results from an analysis of 20 traces where the attack progresses until eventually the correct key guess fails to correlate well enough in phase 11 and no longer propagates to subsequent phases. When considering 25 traces, as in Figure 5.15, the attack successfully recovers all 64 key bits, but it is clear that in phases 4 and 5, the correct key guess was not the top-correlating result and so was only propagated because the top four candidates were propagated from each phase to the next. As expected, the measurement noise in the physical system required more traces to be analyzed than the simulated attack in order to converge on the correct key, but once the correct key had been found, the confidence level continued to increase just as in the simulated attack. Figure 5.16 shows the attack results when considering 250 traces, where it is clear that the correct key is the top correlation result in each phase.
Figure 5.14: KeeLoq Guess Metrics for 20 Measured Traces
Figure 5.15: Keeloq Guess Metrics for 25 Measured Traces
Chapter 5: KeeLoq Example

Figure 5.16: Keeloq Guess Metrics for 250 Measured Traces
Chapter 6

Conclusions

In this section, each of the goals of the SCARF tool will be addressed to show how it has been accomplished. Each of the following sections will additionally describe some of the steps in the development process that led to the resulting design decisions and implementation.

6.1 Raw Trace Analysis

SCARF offers a simple interface for storing raw traces as an arbitrary-length series of double-precision floating-point numbers, with example code for importing data from the common-place comma-separated-values human-readable data format. A similar importer could be trivially implemented for another data source that the researcher requires by parsing out the data as an array of values that are passed into SCARF and subsequently stored in the internal database format. Trace data is, however, available as an output from the SCARF tool in human-readable CSV format for simple external analysis and manipulation in tools like R.
Once stored, SCARF offers a simple pipeline-style interface for analyzing and manipulating the raw Traces to produce Residuals. This allows arbitrary code to be used in the data transformation while abstracting away the details of storing the data and maintaining the relationship between Traces, Residuals, and other associated attributes.

6.2 Leakage Modeling

In order to facilitate the development of a Leakage Model, SCARF offers reference implementations of common operations such as Hamming Weight and Hamming Distance metrics. Because the details of the Leakage Model are usually specific to the attack being performed, SCARF does not currently offer a high-level abstraction for it besides these utility methods. It is possible that, as more examples are developed, common themes may present themselves as opportunities for re-factoring into the framework itself.

6.3 Statistical Analysis

Rudimentary support for statistical analysis of traces is included in the SCARF tool to allow the mean and standard deviation of corresponding samples on the Raw Trace and Residual data-sets. This enables the researcher to more easily determine which points in the raw traces to select for the residuals, and to further study the properties of the selected points for correlation and data-dependence.
6.4 Side-Channel Attack Model

Like the Leakage Mode, the Side-Channel Attack Model is an abstract concept that is the essence of a specific attack and could not be easily encoded programmatically based on only a few attacks. Perhaps a higher-level abstraction can be developed as more example attacks are implemented with similar processes. The one generic aspect of a side-channel attack that was encoded into the framework is the way that the attack progressively collects correlation information to incrementally improve its partial key-guessing strategy. This process is generically useful for many attacks that have been presented in the literature, including the examples detailed in this thesis work.

6.5 Post-Processing

Post-processing of the attack results can be performed due to the standard storage format of the key guess and correlation data collected throughout the attack process. Because this data is all persisted in the database, it is available for post-processing to calculate various metrics describing the attack’s effectiveness and performance. For example, all of the correlated key guesses are preserved in the database, allowing a plot to be generated to show the correlation value of the correct key at each phase of the attack, given that the researcher knows what the correct key is during the post-processing phase.
6.6 Source Code Organization

The source code organization went through several iterations during the development of the example attacks. In experimenting with building a domain-specific programming language on top of Ruby, it was determined that the additional complexity of the framework itself was greater than the benefit offered by the simple language. The domain-specific language would limit researcher to the operations that the framework was designed to do, which is directly counter to the goal of making the framework flexible enough to implement arbitrary attacks. Also, the custom domain-specific language implementation made testing more difficult using the many excellent Ruby automated testing frameworks.

Instead of being implemented as a domain-specific language, SCARF is used as a standard Ruby gem package, offering data models that are accessed from regular object-oriented Ruby scripts. This allows normal testing and mocking frameworks to be used to test both the framework itself and the user code to implement the specific attacks. Because it is standard Ruby, it is also easy for a Ruby developer to get started using the framework by reading the documentation.
6.7 Practical Usage Examples

The main feature of the SCARF tool is the example attacks provided along with the framework. Having fully-worked examples gives future researchers a starting point to understand how to use SCARF in another similar attack. For example, the KeeLoq example attack, which was developed first, took several weeks to work through all the details of building a functional attack based on a prototype attack that itself took several weeks to develop. The KeeLoq attack was then modified to produce a working attack on the Simplified DES cipher in only two days. Additionally, as more example attacks are developed and added to the repository, SCARF can be updated as necessary to support the needs of future attacks that were unforeseen in the initial implementation of the tool. In this way, the library of examples is used as a set of regression-test cases for SCARF, ensuring that previously-supported features are not inadvertently removed from the code base as new features are added or re-factored for better code organization.
6.8 Future Work

This section will discuss some of the identified shortcomings of the SCARF tool and possible avenues for future research and improvement. The content is divided into sections pertaining to the SCARF tool itself, the examples that have been developed for SCARF, and the new areas of research that could be built on top of SCARF.

6.8.1 SCARF Improvements

The primary shortcoming of the SCARF tool itself is probably its installation process. Being built on Ruby using the Gem packaging system, the bulk of the code should be portable between operating systems. Since the tool is designed to work on both Ruby version 1.8 and version 1.9, it should also be simple for a researcher to get it running in their environment. Since it is distributed as a Ruby gem with dependancies listed, the standard installation should ensure that all required libraries are available, assuming the machine is connected to the Internet to download the required gems. The weak point is the dependancy on the Redis database. While Redis is a standard C program that compiles easily using widely-available tools such as make and gcc, this part of the installation is not automatically performed during the installation of the gem. If Redis is not properly installed and available, SCARF will crash with an error when it tries to make use of the database. Redis was originally designed to run on POSIX systems, but Microsoft has recently released\(^1\) a prototype version of the code-base that runs on the Windows platform [25]. Hence, SCARF support is primarily limited to POSIX systems such as Linux, Mac OSX, BSD, and Solaris until the Redis on Windows project matures.

\(^1\)http://blogs.msdn.com/b/interop/archive/2012/04/26/hers-s-to-the-first-release-from-ms-open-tech-redis-on-windows.aspx
Chapter 6: Conclusions

The other shortcoming of SCARF is that, to date, it has had limited use and improvement except by the original developer. As more researchers begin to use SCARF and develop additional example attacks, more features will certainly be added to the core functionality. In addition to these core changes, there may also be a need for a robust plug-in API to assist researchers in developing improvements to the framework without having to dig into all of the core code. At this time, however, the core code is small and simple enough that a plug-in architecture would unnecessarily complicate it.

Finally, additional work could be done to provide more plots and raw data dumps from SCARF. Currently, SCARF is able to automatically generate several useful plots and associated raw data points that have been shown in this thesis. Many other types of plots and information are likely to be both desirable and common to many types of attacks. These types of data-gathering functions could be developed along with the development of more advanced attacks and metrics, and integrated into SCARF for use with future attacks.

6.8.2 Additional Example Attacks

The most obvious future improvement to the SCARF tool is the need for additional example attacks to be developed. Not only would additional attacks be directly relevant to cryptography researchers, but they would also help to identify core weaknesses or missing features in the framework. As more examples of side-channel attacks are added to the knowledge base, common themes can be extracted and abstracted into features available to future attacks built on SCARF.

As for specific example attacks on particular algorithms, the example attack on the single round of DES could be easily extended to an attack on the rest of the DES algorithm, followed by Triple-DES. From that foundation, the AES algorithm would be another logical step away, as it is commonly-used and well-documented. With examples of attacks on these
common ciphers, the SCARF tool will have a useful platform from which researchers can create further examples that may be similar to parts of one or more of the existing attacks but focused on a particular proprietary cipher.

Finally, another shortcoming of the example attacks that have been implemented so far is that they all attack an un-protected implementation of the algorithm under test. An area of possible future improvement would be to develop extensions of these attacks against protected implementations in order to see how the success metrics are impacted by various protection techniques and also to see whether any of these extensions could be simplified by changes to the architecture of the SCARF tool itself.

### 6.8.3 Higher-Level Research

Perhaps the most interesting avenues for future research with SCARF are in the development of higher-level metrics and techniques that can be used to design more effective side-channel attacks and mitigation strategies. In order to research and develop higher-level metrics, SCARF can be used to create a common platform from which to compare and combine the results from existing metrics in a consistent format. Also, because several example attacks have already been provided, future research can focus on the leakage metrics associated with the attack instead of on implementing a successful attack itself. Once the attack has been implemented for a particular algorithm, SCARF allows the researcher to quickly swap out the measurement data in order to compare attacks on different physical implementations, or swap out the leakage model to study the means by which information may leak from the algorithm under various implementations.
Bibliography

sidechannel(s). In Burton Kaliski, etin Ko, and Christof Paar, editors, Cryptographic
Hardware and Embedded Systems - CHES 2002, volume 2523 of Lecture Notes in Com-
puter Science, pages 29–45. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-36400-
5_4.


[5] Mike Borowczak, Greg Mefford, Ranga Vemuri, and Jintai Ding. Correlation meth-
ods used in side channel attacks. Internal research report, University of Cincinnati,
February 2010.


[9] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salma-
sizadeh, and Mohammad Shalmani. On the power of power analysis in the real world:
A complete break of keeloq code hopping scheme. In David Wagner, editor, Advances
in Cryptology CRYPTO 2008, volume 5157 of Lecture Notes in Computer Science,

[10] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salma-
sizadeh, and Mohammad T. Manzuri Shalmani. Physical cryptanalysis of keeloq code

Crypto/Crypto\_misc/DESCracker/.


