A Monitoring Framework for Side-Channel Information Leaks

by

Michael John Lescisin

A Thesis

Submitted in Partial Fulfillment of the Requirements for the

Degree of Master of Applied Science

in

Electrical and Computer Engineering

University of Ontario Institute of Technology

Oshawa, Ontario, Canada

April, 2019

THESIS EXAMINATION INFORMATION

Submitted by: **Michael Lescisin**

Master of Applied Science in **Electrical and Computer Engineering**

Thesis Title: A Monitoring Framework for Side-Channel Information Leaks

An oral defense of this thesis took place on March 8, 2019 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Dr. Akramul Azim
Research Supervisor	Dr. Qusay Mahmoud
Examining Committee Member	Dr. Ramiro Liscano
External Examiner	Dr. Khalil El-Khatib, UOIT - FBIT

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Abstract

A Monitoring Framework for Side-Channel Information Leaks

Michael Lescisin
University of Ontario Institute of Technology, 2019

Advisor:
Dr. Qusay. H. Mahmoud

Security and privacy in computer systems is becoming an ever important field of study as the information available on these systems is of ever increasing value. The state of research on direct security attacks to computer systems, such as exploiting memory safety errors or exploiting unfiltered inputs to shells is at an advanced state and a rich set of security testing tools are available for testing software against these common types of attacks. Machine-learning based intrusion detection systems which monitor system activity for suspicious patterns are also available and are commonly deployed in production environments. What is missing, however, is the consideration of implicit information flows, or *side-channels*. One significant factor which has been holding back development on side-channel detection and mitigation is the very broad scope of the topic. Research in this topic has revealed side-channels formed by observable signals such as acoustic noise from a CPU, encrypted network traffic patterns, and ambient monitor light. Furthermore, there currently exists no portable method for distributing test cases for side-channels - as does for other security tests such as *recon-ng* for network footprinting. This thesis introduces a framework based on interoperable components for the purpose of modelling an adversary and generating feedback on what the adversary is capable of learning through the monitoring of a myriad of adversary-observable side-channel information sources. The framework operates by monitoring *two* data streams; the first being the stream of adversary-observable side-channel cues, and the second being the stream of private system activity. These data streams are ultimately used for the *training* and *evaluation* of a selected *machine learning classifier* to determine its performance of private system activity prediction. A prototype has been built to evaluate the effects of side-channel information leaks on *five* common computer system use cases.

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor Dr. Qusay Mahmoud, for his support and guidance throughout my graduate studies. I would also like to express my sincere gratitude to Dr. Akramul Azim, who has provided me with guidance on the applications of my research towards real-time and embedded systems. Both Dr. Mahmoud and Dr. Azim have provided me with excellent opportunities which have allowed me to refine my research and development skills, and the results presented in this thesis would not be possible without them.

Table of Contents

Abstract **ii**

Acknowledgements **iii**

Table of Contents **iv**

List of Tables **ix**

List of Figures **x**

List of Abbreviations **xiii**

1 Introduction **1**

1.1 Motivation 3

1.2 Contributions 3

1.3 Thesis Outline 6

1.4 Summary 6

2 Related Work **7**

2.1 Side-Channels 7

2.2 Hardware Implementation Artifacts 8

2.3 Data and Instruction Caches 9

2.4 Data Compression 10

2.5 Program Analysis Tools 14

2.6 Generic Frameworks for Data Analysis 16
2.7 Gaps in the State-of-the-art 17
2.8 Summary 19

3 Framework Design **20**

3.1 Architecture 20
3.2 Data Gathering Layer 25
3.2.1 Bash Shell UDP Tagger 25
3.2.2 Firefox Addon UDP Tagger 26
3.2.3 Traffic Tagger For Tunnelled VNC 26
3.2.4 Traffic Tagging Mumble Bot Client 27
3.2.5 Instrumentable Testbench Virtual Machine 28
3.3 Feature Extraction Layer 29
3.3.1 UDP Labelled Sequence Extractor 29
3.3.2 SSH Labelled Sequence Extractor 29
3.3.3 Labels Filter 30
3.3.4 NBurst Filter 31
3.3.5 Time Density Filter 31
3.3.6 Convert Label to Times 32
3.3.7 Convert Labels to Spans 32
3.3.8 Label Cluster 32
3.4 Machine Learning Layer 33
3.4.1 Balanced Label Data Splitter 33
3.4.2 Entity Histogram Creator 33
3.4.3 Decision Tree Builder 35
3.5 Threat Modelling Layer 35
3.5.1 SSH Command Prediction Evaluator 36
3.5.2 HTTPS Page Load Prediction Evaluator 36

3.5.3 VNC Key Press Prediction Evaluator . . . 37
3.5.4 Mumble Subtitle Index Prediction Evaluator . . . 37
3.5.5 String Entropy Calculator . . . 37
3.6 Reactive Layer . . . 40
3.6.1 Warning Logger . . . 40
3.6.2 Polar Gauge Renderer . . . 40
3.6.3 Bar Graph Renderer . . . 41
3.6.4 Entropy Target Renderer . . . 41
3.6.5 Report Webpage Generator . . . 41
3.7 Summary . . . 42

4 Implementation **43**
4.1 Component Implementation . . . 43
4.2 Docker . . . 45
4.3 Scapy . . . 46
4.4 Bochs . . . 48
4.5 scikit-learn . . . 49
4.6 Matplotlib . . . 50
4.7 Interoperability . . . 51
4.8 Summary . . . 52

5 Evaluation **53**
5.1 Analysis of SSH Console Access Traffic . . . 54
5.1.1 Data Gathering Layer . . . 54
5.1.2 Feature Extraction Layer . . . 54
5.1.3 Machine Learning Layer . . . 55
5.1.4 Threat Modelling Layer . . . 56
5.1.5 Reactive Layer . . . 57

5.2 Analysis of HTTPS Web Browsing Traffic 59
5.2.1 Data Gathering Layer 59
5.2.2 Feature Extraction Layer 60
5.2.3 Machine Learning Layer 60
5.2.4 Threat Modelling Layer 62
5.2.5 Reactive Layer 63
5.3 Analysis of SSH Tunnelled VNC Traffic 64
5.3.1 Data Gathering Layer 64
5.3.2 Feature Extraction Layer 65
5.3.3 Machine Learning Layer 65
5.3.4 Threat Modelling Layer 66
5.3.5 Reactive Layer 66
5.4 Analysis of Mumble VoIP Traffic 70
5.4.1 Data Gathering Layer 71
5.4.2 Feature Extraction Layer 72
5.4.3 Machine Learning Layer 72
5.4.4 Threat Modelling Layer 72
5.4.5 Reactive Layer 73
5.5 Analysis of CPU Timing when Entering Password 76
5.5.1 Data Gathering Layer 77
5.5.2 Feature Extraction Layer 79
5.5.3 Machine Learning Layer 80
5.5.4 Threat Modelling Layer 80
5.5.5 Reactive Layer 80
5.6 Summary 85

6 Conclusion and Future Work **86**

Bibliography **89**

List of Tables

4.1 Each functional component described in Chapter 3 is implemented for evaluation in Chapter 5 using the third-party software described in this chapter. . 44

List of Figures

2.1 The selected 35 images from ImageNet 12

2.2 The size of the encrypted images closely follows the size of their plaintext counterparts. 13

2.3 Encrypting uncompressed images generates uniformly sized files and therefore hides entropy information which could reveal which image was fed into the cipher. 13

3.1 The framework high-level architecture. Captures both public (green) and private (red) events. 21

3.2 An example deployment of the framework for auditing network traffic for side-channel information leaks. Network traffic is labelled with private activity. These labels are used for model training by the framework and then discarded. 22

3.3 The five layers of the proposed framework. 24

3.4 Packet 269 represents a key press, packet 277 represents a key release, the SSH packets in between represent the associated generated SSH traffic. . . . 27

3.5 When analyzing a stream of SSH encrypted traffic for side-channels, the plaintext and corresponding encrypted stream may be captured at A or B. 30

3.6 Examples of 10%, 20%, and 30% *leading correctness* of an authentication string. 38

3.7 Bayes' theorem finds the probability that an event occurred given the signals that were received. 38

5.1 The Reactive Layer can render an HTML dashboard describing how well an adversary can learn private information. 57

5.2 Similar to the UNIX command prediction example, this dashboard displays how well an adversary can predict which Wikipedia article was loaded. . . . 63
5.3 The probability of success of a wire-tapping adversary predicting keys typed based on observing encrypted network traffic features. 67
5.4 Information flows from the *bot* client to the *murmur* server at a *fixed* bitrate. 70
5.5 The Mumble client can be configured to only transmit when audio *amplitude* exceeds a *threshold* level. 71
5.6 By classifying based on traffic burst length, several subtitles can be accurately predicted by a wire-tapping adversary. Tick marks begin at subtitle 1 and end at subtitle 224. 73
5.7 When the minimal Linux operating system is idle, lengths of sections of instructions processed between `HALT`s can be clustered in the regions of 0 to 20000 instructions and around 57500 instructions. 78
5.8 When the minimal Linux operating system is executing a CPU consuming userspace process, instruction segments of lengths that lie outside the *idle* cluster occur. 79
5.9 The rendered info-graphic shows that the entropy of the authentication code is greatly reduced through side-channels as the *green* area is much smaller than the *red* area. 81

List of Abbreviations

API - Application Programmer Interface

ASCII - American Standard Code for Information Interchange

CI - Continuous Integration

CPU - Central Processing Unit

CSI - Channel State Information

CSS - Cascading Style Sheets

DRAM - Dynamic Random Access Memory

DVB - Digital Video Broadcasting

GPG - GNU Privacy Guard

GUI - Graphical User Interface

HIL - Hardware In the Loop

HTML - Hypertext Markup Language

HTTP - Hypertext Transfer Protocol

HTTPS - Hypertext Transfer Protocol Secure

IP - Internet Protocol

IT - Information Technology

JPEG - Joint Photographic Experts Group

JSON - JavaScript Object Notation

LLVM - Low-Level Virtual Machine

OS - Operating System

PC - Personal Computer

PCAP - Packet Capture

PCM - Pulse Code Modulation

PDF - Portable Document Format

RAM - Random Access Memory

REPL - Read Evaluate Print Loop

REST - Representational State Transfer

RFB - Remote Framebuffer

RGB - Red Green Blue

RSA - Rivest Shamir Adleman

SIL - Software In the Loop

SSH - Secure Shell

SSL - Secure Sockets Layer

SQL - Structured Query Language

SRAM - Static Random Access Memory

TCP - Transmission Control Protocol

TLS - Transport Layer Security

UDP - User Datagram Protocol

URL - Uniform Resource Locator

VM - Virtual Machine

VNC - Virtual Network Computing

VoIP - Voice over Internet Protocol

VPN - Virtual Private Network

Chapter 1

Introduction

Good software development practice teaches the fundamental rule that software security should be *integrated* into the complete development cycle of the software system and should *not* simply be an afterthought, a final step, or a *layer* isolated from all other system concerns. The software development industry, as well as open source software communities, have acknowledged this design principle for many *types* of security vulnerabilities and have found effective ways of integrating countermeasures against well-known types of software vulnerabilities. For example, when developing a REST based API that interacts with an SQL database it is well known that the API developer should thoroughly examine how data passed over the HTTP calls ends up in the SQL query strings in order to prevent a *SQL injection* attack. To aid in this process, tools are available such as *sqlmap* [1] which can automatically test code for *SQL injection* vulnerabilities.

Despite the good efforts made by many software developers to keep the concern of software security *well integrated* into the development life-cycle, when considering well known types of vulnerabilities, there is one broad category of software vulnerability that has not been thoroughly addressed, that is, *side-channels*. A *side channel* is any type of *unintended* flow of information from a computer system to an adversary [2]. It must be noted that this thesis focuses exclusively on *side-channels* - that is information flows that result *incidentally* as

opposed to *covert channels* where the transmitting and receiving processes have previously *colluded* to build a hidden communication channel [3].

How the information flows from the computer system over a side-channel varies greatly depending upon the *threat model*. Side-channel *transmitters* of information can include, but are not limited to; patterns of encrypted network traffic [4], ambient monitor light [5], CPU/motherboard noise [6], and computer system power consumption [7]. Side-channel information transmitters such as these, combined with realistic threat models, have been shown to have notable consequences for the security of applications. For example, using *power analysis*, researchers were able to derive private keys used by a transit system smart card and thus illegitimately increase the card's monetary balance [8]. Additionally, researchers have shown that by analyzing the *encrypted* network traffic patterns generated by *commercial* smart-home devices, an Internet provider could determine; motion detected by a security camera, as well as the power state of an appliance connected to a remotely switched outlet [9].

As the various means of *side-channel information flow* are so broad in scope it is difficult to simply *blame software developers for not integrating side-channel prevention into their development life-cycle*. Furthermore, the presence of a side-channel can be very dependant upon *computer hardware* or *user behaviour* thus increasing the difficulty for software developers to provide *generalized* side-channel immunity to their programs. Furthermore, side-channels can often result from performance optimizations such as caches or data compression techniques and therefore a software developer is likely to be reluctant to decrease the efficiency of their system in order to satisfy the security requirements for a small minority of the userbase. For these reasons, it is very difficult to make the claim that *all* side-channel information leaks could be fixed if software developers simply put more effort into software quality control.

While the concern of side-channels in software systems is not *entirely* the responsibility of the developers, its effects *do* fall *entirely* on the *users*. Also, in between the *software*

developer and *software user*, there is often a *software system integrator* or a *system/network administrator*. As the *system integrator* or *system administrator* is responsible for taking more *general* third-party software and delivering it to users for a more *specific* application, they too have a share in the responsibility for ensuring the *secure* operation of the software.

Through the above discussed examples, it is evident that there is no *quick fix* for all *side-channels*. Therefore, in this thesis, a *layered framework* is proposed for detecting the presence of side-channels in software systems, *given* a set of *threat models*. This *framework* functions by executing *scenarios*. A *scenario* is a short program employing some or all of the *framework layers* thus generating information on a potential *side-channel attack*. This thesis work therefore improves resistance to *side-channel* information leaks in software systems as it allows *anyone*, from *software developer* to *system administrator* to *hardware engineer* to *end-user*, to develop *scenarios* or extend *framework layers* and thus be able to detect *side-channels* simply by executing these *scenarios*.

1.1 Motivation

The *motivation* for this work is to *facilitate* the process of *side-channel detection* to a level comparable to more *traditional* types of software vulnerabilities such as *SQL* or *OS command injections*. By facilitating this process it will help to settle the dispute of whom the responsibility for the existence of a side-channel falls upon by providing more comprehensive information about potential side-channels through the creation and distribution of side-channel attack *scenarios*.

1.2 Contributions

The main contribution, therefore, of this thesis is a layered framework for evaluation of potential side-channel attacks against a computer system. The vision of this thesis is that the framework designed, implemented and evaluated will serve as a *standardization* for testing

the effectiveness of various side-channel attacks on a variety of system types. To show how this thesis achieves this goal, the following research contributions are presented:

- A five-layer framework for the rapid construction of side-channel attack scenarios complete with a layer for reporting on the *severity* of the information leak.
- Example attack scenarios demonstrating information that can be learned by an adversary monitoring *encrypted* traffic flows from:
  - SSH UNIX console access
  - Web browsing over HTTPS
  - VNC traffic tunnelled through SSH transport
  - Mumble VoIP
- An example attack scenario demonstrating information that can be learned by an adversary monitoring *instantaneous power consumption* of a simulated embedded Linux computer executing a password checking algorithm.

The use cases for this framework vary greatly with respect to the threat model. For example, consider a laptop user who logs into their UNIX server over SSH from public Wi-Fi hotspots. When they are using this *untrusted* network connection it is important that they are aware of the potentially private information that could be learned by an adversary. In this instance, it would be sensible to run this framework *locally* on the users laptop so that they could be alerted in real-time if side-channel information becomes available to a potential adversary that is sufficient to breach the security model of the SSH communication with the server.

For other applications, running the framework on a server-side application could be very insightful to a security analyst. Consider the developer of a web application that has security requirements which demand that an adversary *must not* be able to know what the *user to application* interactions are. With this framework in place on the production server, the

application security team could be quickly made aware if correlations appeared between private user behaviour and network traffic patterns.

As a generalization of the two previously discussed application scenarios, the framework would also fit nicely inside a *continuous integration (CI)* pipeline. In order for a code commit to be accepted into the *master* branch of the project repository, the side-channel security requirements must be met. These security requirements could either be *hard-coded* or they could be *guessed* through *unsupervised learning*. For example, if at the latest commit of a repository, running a test case generates encrypted network traffic such that nothing can be learned through side-channels and after committing to this repository information *can* be learned, an automated alert could be sent to the security team before the commit is accepted.

The final application of this framework which will be discussed is for *cloud computing* providers. For example, if cloud facilities are rented to two different customers from the same cloud provider it is important that one customer must not be able to learn about the activities done by the other customer. Research has shown side-channels to be found linking cloud computing virtual machines rented by different customers which have resulted from shared hardware such as memory caches, hard drive caches, and network caches. As a consequence of the popular *container-based* cloud architecture found in platforms such as *Kubernetes* the scenario occurs when one container moves to another host over network in order to achieve more optimal load balancing. As this *private* inter-container information has the potential to be moving across the *public* Internet, it is important that this communication be encrypted, authenticated, *and* free of side-channels. Therefore, a cloud computing provider could use our framework to verify that both events occurring in one VM cannot influence events occurring in another customers VM and that network traffic patterns resulting from private inter-container communication reveal nothing about the internal state of the VM or container.

Results from this research have been published in the following papers:

- M. Lescisin and Q. Mahmoud, "Tools for active and passive network side-channel

detection for web applications", in *12th USENIX Workshop on Offensive Technologies (WOOT 18).* Baltimore, MD: USENIX Association, 2018.

- M. Lescisin and Q. Mahmoud, "Dataset for web traffic security analysis", in *44th Annual Conference of the IEEE Industrial Electronics Society.* Washington, DC: IEEE Industrial Electronics Society, pp. 2700-2705. 2018.

1.3 Thesis Outline

The remainder of this thesis is structured as follows. **Chapter 2** covers the related work discussing important background information on the concepts relevant to this work. **Chapter 3** describes the design of the side-channel detection framework explaining the role of each of the *five* layers. **Chapter 4** describes the implementation of the framework and justifies the selection of incorporated libraries and tools. **Chapter 5** discusses the evaluation of our framework demonstrating its application to each of the listed example side-channel attack scenarios. **Chapter 6** concludes the thesis summarizing the results gathered from the design, implementation, and evaluation of the framework. Lastly, suggested future work tasks are presented.

1.4 Summary

This chapter introduced the problem of *side-channels* and has provided preliminary discussion on their difficulty of detection and avoidance when compared to more popular types of security vulnerabilities. To this end, this chapter has introduced the core goal of this thesis - that is, to design a framework for monitoring a software system for *side-channel* information leaks. Chapter 2 discusses the related work on *side-channel* classification and detection.

Chapter 2

Related Work

Side-channels have been an active area of research ever since the first examples from the early 1940s when electromagnetic signals emitted by encrypting teletypewriters were found to leak plaintext information [10]. In [11], Spreitzer et al. provide an overview of the characterization and history of computer system side-channels which has thus served to provide direction into the investigation of the types of side-channels discussed in this thesis. In terms of a *threat model*, the authors categorize side-channel attacks into three categories; *local*, *vicinity*, and *remote*.

2.1 Side-Channels

Local side-channel attacks refer to attacks where the attacker must have physical access to the device under attack. For example, a side-channel attack that requires measurement of the electrical potential of the device chassis would be considered to be a *local* side-channel attack.

Vicinity side-channel attacks refer to attacks where the adversary is required to be in *some* physical proximity to the device under attack. For example, in [12], the researchers demonstrate how by monitoring the channel state information (CSI) values of a Wi-Fi link, an adversary could gain insight into which smartphone keys were typed as the position of the

users hand has an influence on the state of the Wi-Fi link. This attack would be considered to be a *vicinity* side-channel attack as the adversary must be within the range of the victim's Wi-Fi signal.

Remote side-channel attacks are the most severe type of side-channel attack as the adversary is not constrained to a physical distance from the system but rather is *global*. For example, measuring response times from the *public* services offered by a server could inadvertently leak *private* information about the server.

2.2 Hardware Implementation Artifacts

In [11], the authors also discuss the notion of a *software-only* side-channel attack. As the paper is focused on mobile devices, the authors state that the variety of sensors that are available in *modern* smartphones, if accessed by a *remote* adversary, could be used to conduct the same side-channel exploits that would otherwise require a *local* adversary to be present with sophisticated measuring equipment. For example, as a video signal is sent to a computer monitor, the amount of power consumed by the monitor varies slightly with respect to the content of the video signal. This subtle but high frequency variation in power consumption can cause electronic components in the monitor's power supply to emit sound. Normally the exploitation of side-channel would be categorized as *local* as it would require an attacker to be physically close by with a microphone. However, researchers [13] have shown that if audio obtained from a built-in microphone during a video conferencing session is obtained by an adversary they would be able to distinguish between different activities done on the computer screen, even though the user has only consented to share their microphone, not their screen. This side-channel is therefore an example of how an unintentional hardware interaction can result in a *remotely* exploitable side-channel that undesirably *bridges* two separate security domains.

It is important to note that by following the academic and industrial research for side-

channels, there are *recurrent* system design patterns which often result in *exploitable* side-channels. These system design patterns are usually put in place for the purpose of *performance optimization* and work by using statistical methods for predicting future events so that the system may be prepared and therefore capable of responding efficiently.

2.3 Data and Instruction Caches

All forms of computer data storage, whether it be network resources, local hard drives, or random access memory (RAM) are affected by the same trade-off - for a given component cost, data can *either* be available in large quantity *or* it can be rapidly accessible *but* it cannot simultaneously satisfy both demands. In order to obtain the most *optimal* outcome from this *necessary* trade-off, system designers use a *hybrid* approach referred to as *caching*. A *cache* is simply a *smaller but faster* storage device that is placed in between the main system and the *larger but slower* storage device [14].

For example, there are *network resource cache servers*. These servers store a copy of the frequently requested network resources so that they can be served to requesting clients as quickly as possible. Modern operating systems implement *file system caches* so that files which have been accessed recently are held in memory so that subsequent reads can skip the time-consuming step of reading the file from a physical hard disk. Random access memory in modern computer systems also implements a cache. As the main memory in a computer system is built from slower but less expensive *dynamic* RAM (DRAM) cells, a smaller cache is built from faster but more expensive *static* RAM (SRAM) cells and stores the contents of frequently accessed memory locations. Similar to the *RAM cache*, many CPUs have a separate *cache* used for program instructions so that program sections which are heavily used (such as loops) can be *rapidly* accessed.

Regardless of the *type* of cache, all caches impose the same property on systems that use them - that is, if a data resource is *quick* to be read, it must have been accessed recently but

if a data resource is *slow* to be read, it must have *not* been accessed recently. This creates what is referred to as a *cache timing side-channel* as it allows an attacker who measures the response time for requested resources to learn the patterns of resource use by a victim. Through the profiling of victim resource use, an adversary is often capable of learning private information from the victim. One popular example of *data caches* being exploited to extract information from a *private* process is the *Flush+Reload* attack. Due to the lack of privilege-based restrictions on the *clflush* instruction in the Intel x86 architecture, any *unprivileged* process may execute this instruction to evict a memory cache entry. This is the *flush* phase of the attack. After completing the *flush* phase, the attacker waits for a period of time and then executes instructions to *reload* the same memory location into the cache. If the *reload* occurs quickly, the attacker can infer that the memory location was accessed by the victim process. If the *reload* occurs slowly, the attacker can infer that the memory location was *not* accessed by the victim process. By measuring memory access patterns of a victim process, researchers were able to learn 96.7% of the bits of a 2048-bit RSA key used for a signature operation in GnuPG as demonstrated in [15].

Research has shown that these types of *cache-based* side-channels are capable of exploiting more than just cryptographic operations. In [16], the author describes how by measuring which functions in a program were called as well as how much time was spent in each function, an attacker running an *unprivileged* process on the same computer could; determine which one of the top 100 Wikipedia articles were visited using the *Links* browser (94% accurate), and determine which one of a set of 127 PDF documents were rendered by the Poppler library (98% accurate).

2.4 Data Compression

Another computer system optimization, especially effective for data that must move across networks or be stored in long-term storage is *data compression*. There are various data

```
for i in *.jpg
do
        convert $i -scale 150 $i
        convert $i -crop 90x90+0+0 $i
done
```

Listing 2.1: Each image from ImageNet is scaled and cropped to a resolution of 90×90.

```
for i in *.jpg
do
        cat $i | gpg -c --passphrase 1234 --batch \
                --compression-algo none > $i.gpg
done
```

Listing 2.2: Encrypting all JPEG compressed images with GPG

compression algorithms with different optimizations for different types of data, but at the core of each algorithm is the idea of removing all redundant information from the input data source that is to be compressed [17]. As each *unique* data input source is likely to contain different amounts of redundant information, the compressed output size is likely to be *unique* to each input data source.

To explain how this is problematic from a *security* standpoint, a demonstration is presented. Thirty-five images from ImageNet [18] were selected (Figure 2.1). Each image was cropped and scaled to a resolution of 90×90 pixels (Listing 2.1). As can be seen in Figure 2.3 when each image is uncompressed by being converted to *RAW RGB* format all file sizes are identical. However, as these images are *compressed JPEGs*, a variance in file size is observed. By encrypting each *compressed* image with GPG (Listing 2.2), a variance in file size is once again observed (Figure 2.2). The results of this simple experiment have shown that an adversary, by simply measuring the *size* of the GPG encrypted data, can learn which one of the thirty-five images was transmitted - something that they would be incapable of learning if the GPG inputs were *uncompressed* images.

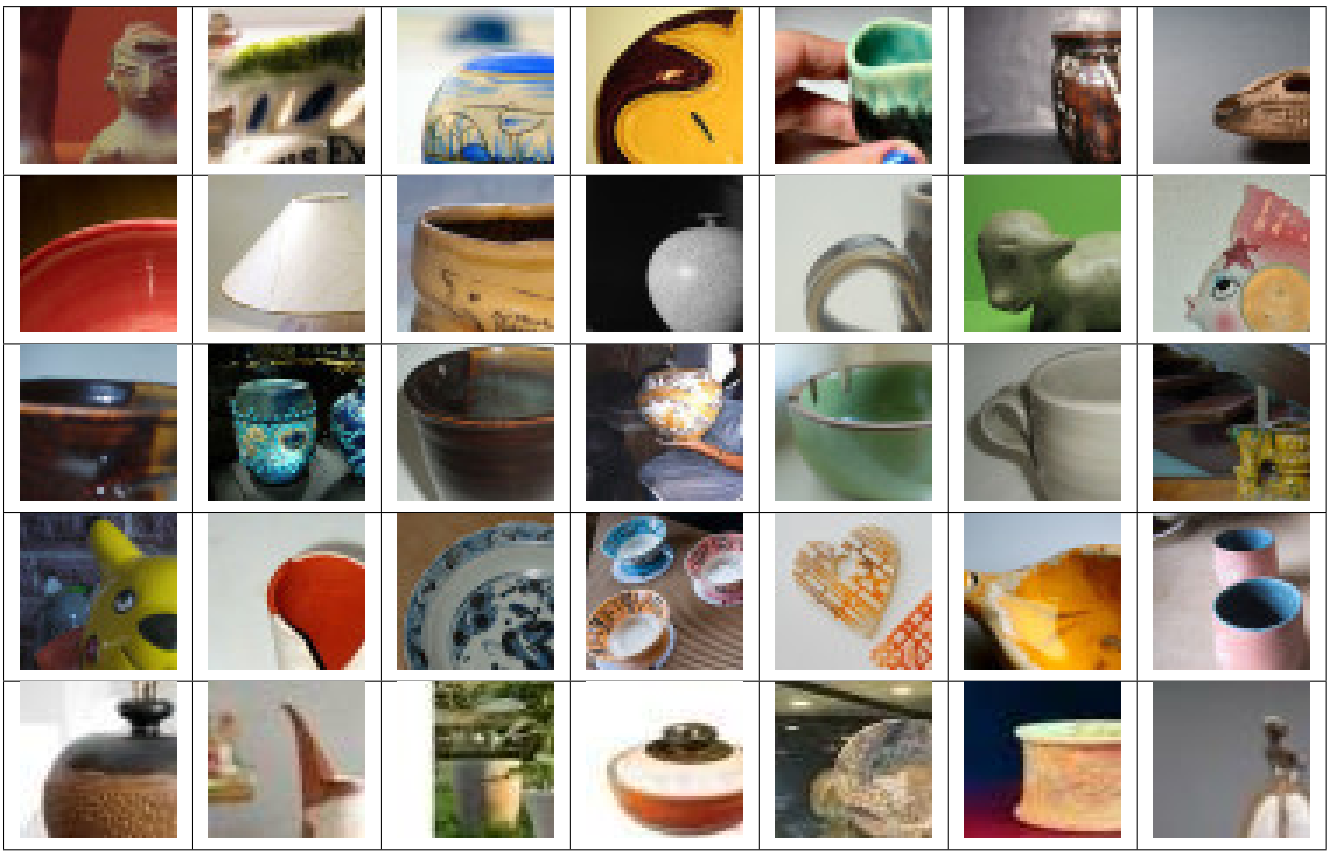

Figure 2.1: The selected 35 images from ImageNet

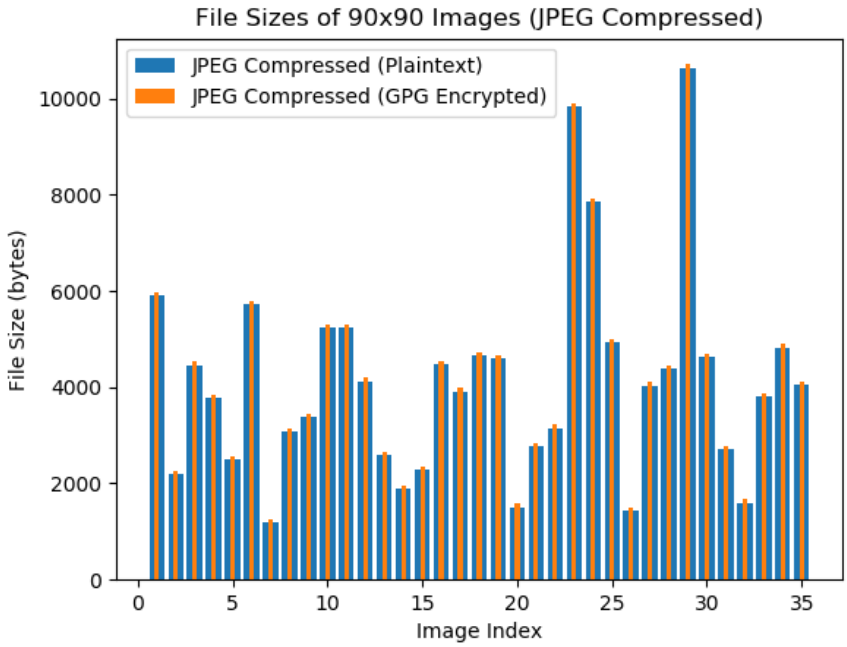

Figure 2.2: The size of the encrypted images closely follows the size of their plaintext counterparts.

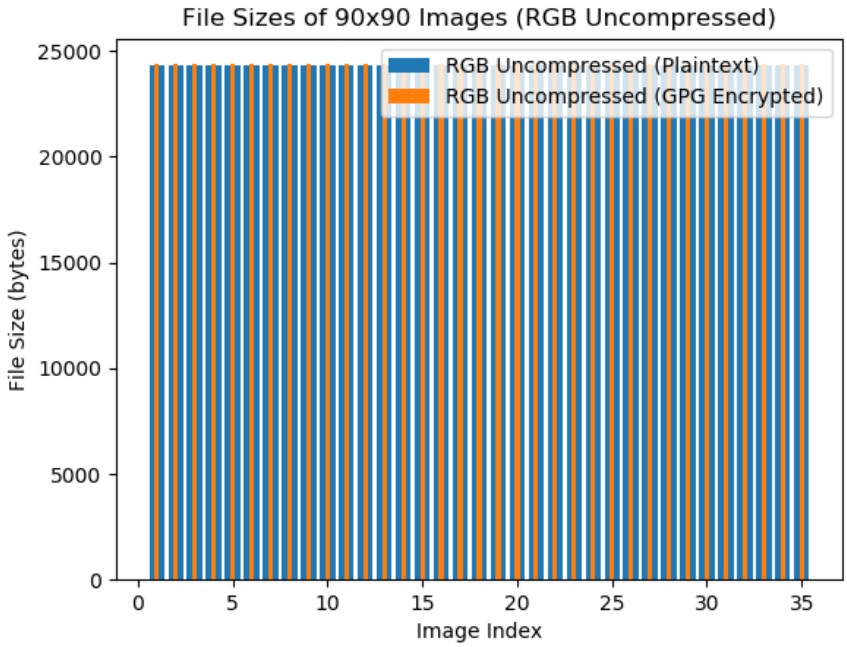

Figure 2.3: Encrypting uncompressed images generates uniformly sized files and therefore hides entropy information which could reveal which image was fed into the cipher.

2.5 Program Analysis Tools

As side-channels pose a serious threat to computer security and privacy, research has been directed to the development of *program analysis tools* for checking a program for *specific* types of side-channel information leaks.

SideBuster [19] is a tool created by Zhang et. al. for detecting private information leaks through encrypted traffic patterns in *web applications*. SideBuster works by performing *static* taint analysis on web application source code to determine when private information is sent to the server *and* when information is transmitted to the server on a condition that is dependant upon the value of a *private variable*. SideBuster then generates test cases to cause these events that handle private information, while the associated network traffic is recorded. After the user-interaction/network traffic dataset has been built, the *quantification* phase then begins. The metric which SideBuster uses to quantify the *severity* of side-channels is the loss of *information entropy* for the set of *user / web application* interactions. For example, if there are 16 possible interactions, a perfectly private system would generate traffic patterns with an entropy of $H = -log_2(\frac{1}{16}) = 4$bits. However, if a traffic pattern provides features that indicate that any one of 4 different interactions may have occurred the *entropy* would then be $H = -log_2(\frac{4}{16}) = 2$ bits. Therefore, SideBuster would quantify the *severity* of this side-channel information leak to be $4 - 2 = 2$ as it carries sufficient information to reduce the *entropy* of the possible interactions set by two bits.

In [20], the authors examine the work done with SideBuster but instead of using *static* analysis for obtaining the points in the application at which client-server communication occurs, a *dynamic* analysis approach is instead used. In this approach, the authors employed the *Crawljax* web crawler to generate a state-machine model, including page state changes done by JavaScript, of the web application under test. As this approach is purely *dynamic*, no application source code is needed. After the *crawling* phase is complete and the state-machine model has been generated, the application then proceeds on to the *leak quantification* phase. This is the phase where the *severity* of the leak is quantified. The authors discuss the

shortcomings of *information entropy* based measurements for *leak quantification* and claim that due to the fact that *nearest-centroid classifiers* were used, selecting the *radius* of the *decision boundary* is an unclear task and often this radius value is simply chosen arbitrarily. Using this methodology, if the centroid for data of class B lies within the circle with the center being the centroid for data of class A and the *radius* being R the classes A and B would be considered *indistinguishable*. However, if this radius is decreased even by a small amount to $R - \epsilon$ the centroid for class B may now be outside this circle and the classes would considered *distinguishable*. Due to this fragility, the authors suggest using the *Fisher criterion* instead.

The *Fisher criterion* is a measurement of the ratio of *variance between classes* to *variance within classes*. A side-channel free system will have the property of the *Fisher criterion* being equal to zero. This value of zero is a result of either the *variance between classes* (numerator) being equal to zero as all actions generate the same network traffic pattern or the *variance within classes* (denominator) approaching infinity as all actions generate completely random traffic patterns.

Central to the idea of *program exploration*, be it a *web application* or a *system program* is the notion of *symbolic execution*. When a program is executed *symbolically*, instead of *concrete* values being entered as inputs, *symbolic* values are entered instead. As the instructions of the program are processed, the *expressions* representing the state of the CPU are updated. For example, if a function `add(int a, int b)` is to be executed *symbolically* the following would happen. Symbolic variables for a and b would be created. Suppose that in the machine language implementation of this program the first two arguments to a function are passed through the registers `R0` and `R1` respectively. The *symbolic execution engine* would hold the *expression* `a` in `R0` and `b` in `R1`. Suppose the next instruction in this function to be executed is `ADD R0, R1`. This would cause `R0` to hold the symbolic expression `a+b`. Symbolic execution allows a developer to see how a program's state space is affected by its input variables.

CANAL: A Cache Timing Analysis Framework via LLVM Transformation [21] is modelling framework for modelling the cache timing behaviours of a program. CANAL works by instrumenting LLVM bitcode with code that gathers and reports the statistical properties of memory-cache hits and misses. CANAL can be combined with symbolic execution tools in order to detect timing side-channels. The authors of CANAL define a program, $P(k)$, to have a timing side-channel leak if there exists any value of k such that $\tau(P, k_1) \neq \tau(P, k_2)$ where $\tau(P, k)$ is the execution time of the program P under the input k. The authors then use the KLEE [22] symbolic execution engine to execute a function-under-test twice - once with symbolic variable `input1` and again with symbolic variable `input2`. Using the symbolic execution data from KLEE, the authors then place an assertion that the amount of cache *hits* from `input1` must be equal to the amount of cache *hits* from `input2` and the amount of cache *misses* from `input1` must be equal to the amount of cache *misses* from `input2`. If KLEE is capable of finding *concrete* values for `input1` and `input2` such that the amount of hits and misses differ, then the program contains a timing side-channel.

2.6 Generic Frameworks for Data Analysis

As several of the previously discussed tools used for detecting side-channels in web applications involve *data analysis* it is worthwhile investigating the frameworks available for *general purpose* data analysis.

Orange [23] is a general purpose data analysis framework created at the University of Ljubljana which leverages Python's scientific computing libraries for creating a dataflow-based visual programming language allowing for the simple creation of data analysis and visualization programs. Programs are built within *Orange* by connecting together nodes (referred to as *widgets*) where each *widget* is essentially a *function* which calculates an output based on one or more inputs. *Widgets* are available for data acquisition, data filtering, model training, model evaluation, and data visualization. *Orange* is primarily targeted towards data

scientists with little to no programming knowledge. Although not strictly a security tool, *Orange* can be extended through Python and therefore many of the functional components discussed in this thesis could be ported to *Orange* without requiring large amounts of code to be re-written.

Another popular general purpose data analysis framework is *Waikato Environment for Knowledge Analysis (Weka)* [24]. *Weka* was created at the *University of Waikato* with the goal of providing a graphical interface for user interaction with dataset visualization, filtering, clustering, classification, and regression. Similar to *Orange*, *Weka* also provides a visual dataflow language referred to as *Knowledge Flow*. Support for the execution of *Python* code in *Weka Knowledge Flow* programs is supported and therefore, similar to *Orange*, many of the side-channel specific functional components presented in this thesis could be augmented with Weka support relatively easily.

2.7 Gaps in the State-of-the-art

The reviewed research on the current state of side-channels and their detection provides a great variety of motivating examples for considering these types of vulnerabilities when designing secure software systems. The research conducted describing the *static* and *dynamic* analysis techniques used for checking programs for side-channel information leaks is something that should be considered and implemented by testers of secure software. Indeed, the current state-of-the-art of side-channel detection can catch many of the implementation issues that result in information leaked via side-channels. For a *common* type of side-channel, such as a cache-timing attack or analysis of encrypted web application traffic, there exist tools capable of detecting if observable activity patterns are capable of distinguishing between user/application behaviours. However, if an example of a *less common*, but potentially severe, type of side-channel is discovered, the security community must build a detection tool from the ground up. This unnecessary effort leads to a lack of available side-channel detec-

tion tools and thus side-channel information leaks remain prevalent in computer systems. Lastly, the available side-channel detection solutions consider exclusively *machine* behaviour and omit the effects of *human* behaviour [25]. For example, when detecting side-channels formed by the size of network traffic bursts generated when downloading web pages over HTTPS all of the reviewed solutions considered exclusively the *features* associated with the traffic bursts. In reality, there is a great amount of information available in the time gaps between traffic bursts. These time gaps, in the context of web traffic, can result from *human* behaviour such as how much time was spent on a webpage depending upon how *interesting* the content was or which links on a page a user is most likely to click. This lack of ability to accurately emulate *human* behaviour is indicative of the current state of research which is limited to modelling common machine configurations such as caches and encrypted network tunnels.

In [11], the authors discuss *reproducibility and responsible disclosure* as important steps into the research of side-channels. The authors claim that publishing the frameworks for reproducing side-channel attack scenarios would be helpful to security researchers. This goal forms a core component of this thesis which describes a *layered* framework which can be used to gather side-channel cue information (eg. network traffic, ambient light, etc.) and report on how accurately private information can be predicted. In this way, components of the proposed side-channel detection framework can be *reused* to build detectors for new types of side-channels, *and*, given that this proposed framework is driven by a data stream of observable system characteristics and their corresponding *private* labels, the deployment of the proposed framework is capable of detecting side-channels that result from *human* behaviour.

The need to build a *new* framework is justified as using available *machine-learning* tools does not provide the necessary functionality for robust side-channel detection. Specifically, readily available machine learning tools are not aware of *data features* specific to different network traffic types nor do they contain a variety of side-channel data acquisition methods.

Lastly, nothing from the realm of generic *machine-learning* tools is available to model the *specific* security requirements of an application and then automatically take the appropriate action should any of these requirements be unmet.

2.8 Summary

In this chapter, the research on the various types of side-channels, their causes, and methods for their detection, has been explored. After the relevant literature has been reviewed, the shortcommings of the current state of research, specifically the lack of interoperable components for constructing side-channel test cases, including those involving *human* user interaction, are presented as challenges for which the presented framework strives to solve. The design of the proposed framework is presented in the next chapter.

Chapter 3

Framework Design

This chapter presents the *design* of the side-channel detection framework. After presenting an overview of the *layered* architecture, each framework *layer* is described in full detail. When reading the descriptions of each layer, it is important to note that the functional components contained within each layer are merely those which are necessary to conduct all evaluations discussed in this thesis. The *architecture* of the framework is designed so that future work may extend each layer with more functional components. While reading the descriptions of each functional component, it must be emphasized that this *chapter* simply describes the *functionality* requirement of each component. The discussion for how each functional component is implemented can be found in *Chapter 4 - Implementation*.

3.1 Architecture

The presented side-channel detection framework must be *installed* at a point where both *publicly observable* and *private* information flows can be monitored (Figure 3.1). In this thesis, the *publicly observable* side-channel information sources are encrypted network traffic patterns and CPU power consumption. Private system events examined in this thesis include commands executed in remote shells and keys typed into a remote desktop session. Therefore, the framework must be able to log both of these private (accessed through privi-

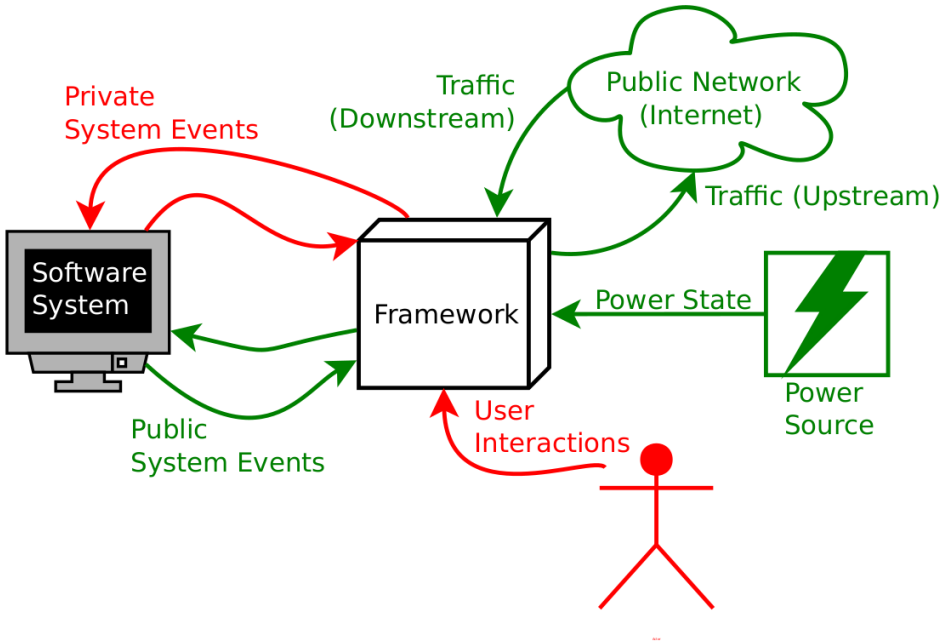

Figure 3.1: The framework high-level architecture. Captures both public (green) and private (red) events.

leged operations) and public (accessed through the monitoring adversary observable signals) flows of information so that the ability of a potential side-channel adversary inferring *private* events using only *public* events may be measured. Figure 3.2 shows a possible deployment where the framework is deployed to a gateway server which receives *public* Internet traffic labelled with *private* event labels and uses this information to train a model side-channel adversary. At the gateway server, the *private* labels are removed so that a *real* adversary would not gain an advantage in learning *private* system activity patterns.

The framework is designed as a *layered* architecture (Figure 3.3). Although the presented framework contains design patterns similar to a *pipe and filter* architecture, there are several aspects which run contrary to *pipe and filter* design and thus the framework architecture is *layered*. One of these aspects is the *SSH Labelled Sequence Extractor* which belongs to the *Feature Extraction Layer* but invokes functional components from the *preceding Data Gathering Layer* thus going against the unidirectional flow of data required by *pipe and filter* systems. Furthermore, future work could extend the *Reactive Layer* to adjust parameters

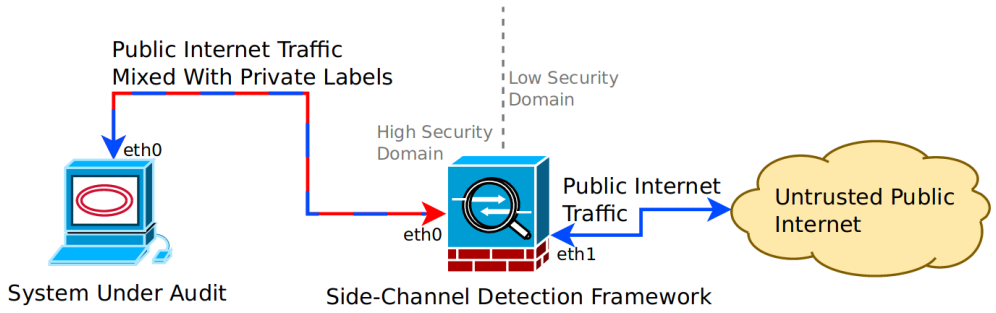

Figure 3.2: An example deployment of the framework for auditing network traffic for side-channel information leaks. Network traffic is labelled with private activity. These labels are used for model training by the framework and then discarded.

in the preceding layers for the purpose of accuracy improvement, thus once again violating the *pipe and filter* property of unidirectional data flow. The reason for choosing a *layered* architecture is for the promotion of component reuse. A popular example of a *layered* software architecture is the TCP/IP stack. When a developer wishes to create a new network service they only need to work on the appropriate layer of the stack. For example, if a developer is designing a RESTful API, they only need to write the HTTP server back-end as the methods for ensuring communication between this server and its clients are left to the lower levels of this stack thus *reusing* the software written by other developers. In a similar manner, when developing a side-channel attack *scenario* using the proposed framework, components along the attack workflow may be reused. For example, a framework component which gathers ambient light information to be later used to detect how successfully an adversary could learn which application is in use could be replaced with a component that gathers information on instantaneous power consumption [26] for the purpose of determining the success of a power analysis side-channel adversary on learning which application is in use.

Splitting the workflow components of a side-channel attack into *generalizeable* and *orthogonal* concerns, the proposed framework allows for *rapid* design of side-channel attack models for the purpose of ensuring that software security model requirements are being met. Specifically, the *concerns* are split into: a *data gathering layer* where both private system

events and public system events are logged, a *feature extraction layer* where captured data is filtered thus creating a representation of the private/public system behaviour that is well suited for training machine learning classifiers, a *machine learning layer* where classifiers are trained and evaluated on their ability to predict *private* events given observed *public* side-channel events, a *threat modelling layer* where the performance measurements from the evaluated machine learning classifiers are evaluated against the system *threat model*, and a *reactive layer* where an action is performed based on the result from the *threat modelling layer*.

Figure 3.3: The five layers of the proposed framework.

3.2 Data Gathering Layer

The *data gathering layer* consists of the components which are necessary for gathering the *raw* data from a source that, under the system security requirements, is considered to be observable by an adversary. The data which is gathered *must* contain both the observable traits *and* the associated information which according to the system's security requirements is intended to remain *private*. The format of this data stream is not specified and is likely to vary depending upon the nature of the data being captured. The only requirement for the formatting of the data sourced from the *data gathering layer* is that there is a module in the above *feature extraction layer* that is capable of extracting tuples of $(featurevector, label)$ from this stream.

While the number of potential components for the *data gathering layer* is ever growing due to the ever growing body of research on side-channels, the proposed framework contains the following data gathering tools capable of detecting many of the types of side-channels discussed in the *related work* section of this thesis.

3.2.1 Bash Shell UDP Tagger

The purpose of the *Bash shell UDP tagger* is to add UDP packets (*traffic tags*) to the set of captured packets to denote the beginnings and endings of traffic bursts that flow from server to client and *are the result of* the execution of a shell command. This shell provides the same interface as a regular UNIX shell but has the property that it labels network traffic, via UDP packets, denoting the beginning and end of each command execution. Therefore, after logging into into a remote host over SSH and obtaining a *bash* shell, this specialized *tagging shell* is then launched thus generating pairs of encrypted network traffic samples *and* the information about the command which caused the generation of these traffic samples.

The UDP tags contain the following fields:

- `state`: denotes if this is the start (`begin`) or end (`end`) of a shell command execution.

- `command`: the shell command that was executed.
- `context`: the directory which this command was executed in.
- `timestamp`: the time at which the command execution begun.

3.2.2 Firefox Addon UDP Tagger

The purpose of the *Firefox addon UDP tagger* is similar to the purpose of the *Bash shell UDP tagger* except instead of tagging shell commands executed, website URLs are tagged. All that is required to gather data pairs of HTTPS traffic samples *and* website URLs is the installation of this addon in Firefox.

This addon generates UDP label packets with the following fields.

- `state`: denotes if this is the start (`begin`) or end (`end`) of a web page load.
- `url`: the webpage URL that was loaded.
- `timestamp`: not a true timestamp but rather a unique identifier showing that a `begin` tag is matched to an `end` tag.

It is worth noting that the *URL* is not the only potentially confidential piece of information which a user would want to hide from an adversary. The web page content that is loaded can also be dependant upon cookies and thus if information is learned about the cookie-dependant page then information is also learned about the private cookies and the actions which set them the way that they are. In order to include this information into a side-channel detection scenario, the *Firefox addon* would need to be extended to capture the important *private* elements of the page content and add this information to the UDP tags.

3.2.3 Traffic Tagger For Tunnelled VNC

As will be discussed later in this thesis, the *feature extraction layer* provides a method named *get_pcap_ssh_encrypted_sequence* whose purpose is to monitor network traffic at the

endpoints of SSH tunnels and isolate sections of the SSH encrypted data stream according to *selected* events from the plaintext data stream entering or leaving the SSH tunnel. The role therefore of this *traffic tagger for tunnelled VNC* is to *select* relevant events from a plaintext TCP stream carrying Virtual Network Computing Remote Framebuffer Protocol (VNC RFB) data. Specifically, the events of *key press* and *key release* are of interest as they have the potential to manipulate the remote display in ways distinguishable through network traffic side-channels. By following the documentation for the VNC RFB protocol [27], it is known that if the length of a TCP payload is *8 bytes* with the leading byte holding the value `4` then the given TCP payload is the description of a keyboard event. The second byte in this payload indicates if the keyboard event is a *key press* (`1`) or a *key release* (`0`). The last byte in this TCP payload holds the ASCII value of the key that was pressed or released. Therefore, this traffic tagger can be used in conjunction with the *get_pcap_ssh_encrypted_sequence* method to create a *filter* which will isolate sections of an encrypted VNC traffic stream based on the state of the plaintext VNC traffic stream parser which is controlled through the reading of VNC events from the plaintext traffic stream (Figure 3.4).

269	13.233405	172.17.0.1	172.17.0.3	VNC	74	
270	13.233422	172.17.0.3	172.17.0.1	TCP	66	5900 → 37060 [ACK] Seq=3033 Ack=451 Win=227 Len=0 TSval=31
271	13.233558	172.17.0.3	172.17.0.2	SSH	110	Client: Encrypted packet (len=44)
272	13.233619	172.17.0.2	172.17.0.3	TCP	66	22 → 56674 [ACK] Seq=3873 Ack=2553 Win=292 Len=0 TSval=224
273	13.279922	172.17.0.2	172.17.0.3	SSH	598	Server: Encrypted packet (len=532)
274	13.280108	172.17.0.3	172.17.0.1	VNC	564	
275	13.280689	172.17.0.1	172.17.0.3	VNC	76	
276	13.280797	172.17.0.3	172.17.0.2	SSH	110	Client: Encrypted packet (len=44)
277	13.296331	172.17.0.1	172.17.0.3	VNC	74	

Figure 3.4: Packet 269 represents a key press, packet 277 represents a key release, the SSH packets in between represent the associated generated SSH traffic.

3.2.4 Traffic Tagging Mumble Bot Client

In order to experiment with the generation of traffic patterns from encrypted Voice Over IP (VoIP) systems, a bot client was written for the *Mumble* voice chat system. This bot client is designed to stream an audio file to a *Mumble server* (referred to as *murmur*) while

simulating the effect of using the *raw amplitude detection* method for detecting when the speaker is speaking and when they are not. Along with the audio file which this *bot client* "speaks" to the *murmur server*, a list of subtitles for the audio file is also passed. Therefore as the *bot client* streams the audio file over the network thus simulating a human user speaking, UDP packets denoting the beginnings and endings of speech segments are streamed as well. The result of this is quite similar to the *Firefox addon tagger* and the *Bash shell tagger* - the above *feature extraction* layer receives a network traffic stream of encrypted data and its corresponding labels for which feature vectors are generated so that machine learning models can be used to simulate the effects of a side-channel adversary.

3.2.5 Instrumentable Testbench Virtual Machine

So far, the members of the *data gathering layer* have consisted exclusively of methods which gather data related to network traffic. The emphasis on investigating network traffic based side-channels is justified as the cost barrier to entry for these types of attacks is relatively low and the adversary need not be physically close to the target machine. However, as the goal of this thesis is not *exclusively* the detection of network traffic based side-channels, other types of side-channels such as those discussed in the *related work* section indeed must also be considered.

In order so that the effects of hardware based side-channels can be *reliably* simulated while requiring a minimal amount of *simulation code* to be written, an *instrumentable testbench virtual machine* is included in the *data gathering layer* of the proposed framework. This instrumentable virtual machine allows high-level scripts to be written to simulate user interaction as well as gather data from simulated hardware and potentially alter its operation. Ultimately, the high-level goal of this instrumented virtual machine is the same as the previously discussed *data gathering tools* - that is, to generate a log file of *private interactions* and correlated *observable system behaviours*. In the same manner as previously described, these generated logs are parsed by members of the above *feature extraction layer*

where feature vectors are generated so that machine learning algorithms can evaluate threat models against the simulated system.

3.3 Feature Extraction Layer

The *feature extraction layer* consists of the preliminary data processing that is required to be performed on the *raw* captured data in order to generate *feature vectors*. Through the studying of the related work as well as understanding how network applications work from high-level to low-level, methods are implemented in this layer which generate the features, which have been shown to be highly applicable for side-channel detection, from the supplied *raw* data. The following subsections describe the modular functional components found at this layer.

3.3.1 UDP Labelled Sequence Extractor

This functional component proves itself to be useful when handling raw streams of encrypted network traffic labelled by UDP packets such as that which is generated by the *Bash shell UDP tagger* or the *Firefox addon UDP tagger*. Simply put, this method reads through a capture of network traffic and produces pairs of the label of the *private* activity *and* the *ordered list* of TCP payload sizes which are transmitted over the network during the time period for which the *private* event occurs.

3.3.2 SSH Labelled Sequence Extractor

This functional component is instrumental for creating scenarios which evaluate the security and privacy of tunnelling plaintext TCP protocols over SSH. This method is designed to work in the situation where network traffic to be analyzed is captured at a host which serves as the *entry point* or *exit point* of an SSH tunnel (Figure 3.5). Specifically, this method excepts the following parameters:

- The set of captured network traffic packets.
- The source of the plaintext traffic (IP, port)
- The destination of the plaintext traffic (ie. SSH tunnel entry point) (IP, port)
- The source of the encrypted traffic (ie. SSH tunnel entry point) (IP, port)
- The destination of the encrypted traffic (ie. SSH tunnel exit point) (IP, port)
- A reference to a function which will label the encrypted traffic stream based on the events detected in the plaintext traffic stream

This method returns a list of tuples in the form (e, p) where `e` is the event descriptor for a *private* event and `p` is the set of TCP packet sizes of the SSH encrypted traffic stream that ensues the occurrence of the private event.

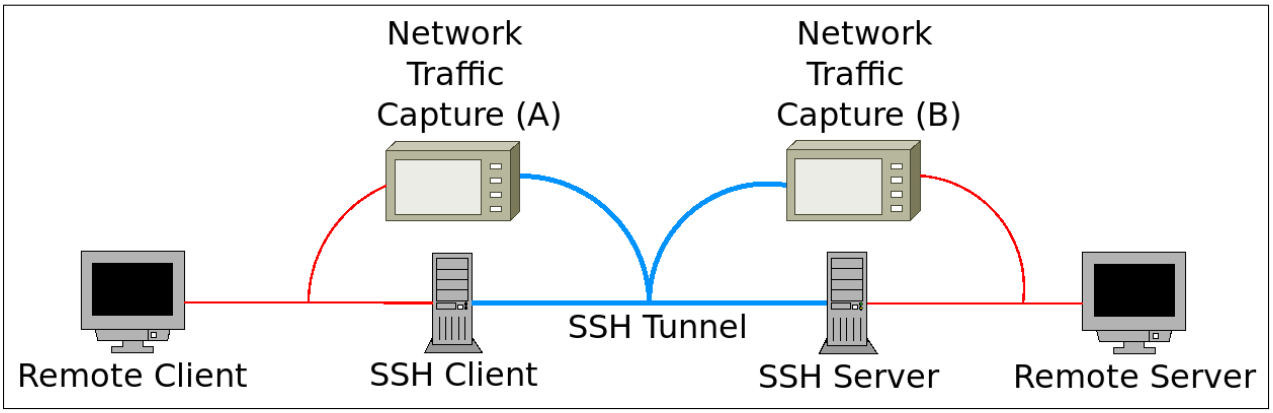

Figure 3.5: When analyzing a stream of SSH encrypted traffic for side-channels, the plaintext and corresponding encrypted stream may be captured at A or B.

3.3.3 Labels Filter

This functional component simply works to only keep the desired $(key, value)$ pairs for a given dictionary object. For example, a dictionary object may exist which describes the loading of a webpage. This dictionary may contain keys which describe the `url` that was loaded along with the HTML page `title`, and the associated HTTP cookies. Consider a

security model which only considers the HTML page `title` to be *private*. This method could therefore be used to reduce this dictionary to one which only contains the `title` key.

3.3.4 NBurst Filter

This functional component is instrumental for building scenarios which involve analyzing encrypted *web* traffic. This is because when web objects (ie. HTML documents, JavaScripts, images, videos, etc...) are downloaded over HTTP they typically generate continuous sequences of full length TCP payloads. This is a consequence of the fact that the sizes of typical web objects greatly exceed that of the maximum data unit of a TCP datagram [4]. This method accepts a list of integer numbers representing the sizes of frames/packets/payloads, along with the *target* packet size, referred to as N. The method then returns the list of byte counts for each continuous run of N byte packets in the input sequence plus the size of the packet trailing the N byte run.

Suppose the input sequence is `[43, 54, 17, 1370, 1370, 121, 324, 1370, 1370, 1370, 12]` and *NBurst filtering* is performed with $N = 1370$, the filtered output sequence will be `[2861, 4122]`. Ultimately, the goal of *NBurst filtering* is to recover the approximate sizes of the web objects that were downloaded by a client.

3.3.5 Time Density Filter

Similar to the goal of segmenting a traffic stream based on packet size patterns as is done with *NBurst filtering*, *time density filtering* works by segmenting a traffic stream based on the *time intervals* between packets. This functional component accepts four parameters:

- The set of captured network traffic packets.
- A reference to a function that returns a *Boolean* value indicating if a given packet matches the criteria for consideration (ie. correct source/destination address).
- The target time spacing between packets.

- The tolerance of the time spacing.

Suppose packets are stored in a file with the given timestamps; `[0.0, 2.0, 4.0, 6.0, 9.0, 18.0, 25.0, 27.0, 29.0]`. If *packet timing density filtering* is applied with a *target time spacing* of 2.0 and a tolerance of 50% the following sequence will be returned; `[[0.0, 2.0, 4.0, 6.0, 9.0], [25.0, 27.0, 29.0]]`. The ultimate goal of using *packet timing density filtering* is to create *clusters* of packets based on bursts of network traffic. This functionality is instrumental for building scenarios involving *real-time* applications as this *clustering* reveals the presence of these *real-time* events for which further features can be extracted (such as total bytes transferred) in an attempt to learn *private* information about the event.

3.3.6 Convert Label to Times

This functional component accepts a file of captured network traffic packets labelled with UDP labels and returns a list of tuples in the form (t, L) where t is the *time* of the label and L is the dictionary holding the $(key, value)$ pairs for the given label. The goal of this method is to facilitate, the later described, labelling of packet clusters.

3.3.7 Convert Labels to Spans

This functional component accepts a list of tuples generated by *Convert Label to Times* and matches `begin` labels with `end` labels in order to generate a list of tuples in the form (t_i, t_f, L) where t_i is the time at which an event begins, t_f is the time at which the event ends, and L is the descriptor of the event.

3.3.8 Label Cluster

The *Label Cluster* functional component forms the third and final step of assigning event labels to the detected bursts of network activity. This functional component accepts the

spans generated from the *Convert Labels to Spans* method along with a *start time*, t_i, and an *end time*, t_f. This method then returns the set of possible labels (ie. labels which have a start time greater than or equal to t_i and an end time less than or equal to t_f) for the given (t_i, t_f) pair.

3.4 Machine Learning Layer

The core role of the *machine learning layer* is to determine the accuracy of predicting private information for an adversarial party following a given machine learning algorithm. The *machine learning layer* is fed with the *extracted features* from the preceding *feature extraction layer* along with the *private* event labels. From this layer is generated a probabilistic model for the prediction of all events in the captured event space. It is the role of the following, *threat modelling layer*, to determine if the success of a machine learning adversary violates the *security model* for the system or application.

3.4.1 Balanced Label Data Splitter

This functional component splits a set of *feature vectors* and its corresponding labels into two *disjoint* sets for which one is used for *model training* and the other for *model testing*. In order to train the machine learning model to recognize *feature vectors* as accurately as possible, this method tries to keep the same proportions of labels in both the *training* and *testing* sets so that the machine learning classifier does not learn an incorrect *a priori* probability.

3.4.2 Entity Histogram Creator

This functional component provides a means to map a list of feature vectors of varying dimensionality into a list of feature vectors of uniform dimensionality. This is required for training machine learning models as most machine learning models expect all feature vectors to be of uniform dimensionality. Specifically, this functional component works by

first building a set of all integers in the *inclusive* range of the smallest number from all feature vectors to the largest number from all feature vectors. Otherwise put, each feature vector in the set is replaced with its smallest dimension and the smallest value in this set is selected. Similarly, in another copy for the set, each feature vector is replaced with the largest dimension and the largest value in this set is selected. From this, a new set is built of all integers in the *inclusive* range of these two numbers.

Now that this *range* set has been built, which is of fixed length, each feature vector is mapped to a new vector where the value of each dimension is the count of times each number appears. As a result of the *fixed length* range set, all returned *histogram* vectors are of *uniform* dimensionality. The application of this algorithm is best illustrated with an example; suppose there is an input set of feature vectors of *non-uniform* dimensionality, `[[1, 2], [5, 6, 7, 7], [12, 7, 2]]`. The *minimum* value is 1 and the *maximum* value is 12. Therefore the range set is `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]`. Generating new vectors where each vector is the *histogram* of its input vector yields the following; `[[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`, `[0, 0, 0, 0, 1, 1, 2, 0, 0, 0, 0, 0]`, `[0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1]]`. It should be noted that this process looses information as the original order of sizes in each input vector is lost (eg. `[1, 2, 1]` and `[1, 1, 2]` would both map to `[2, 1]`). Therefore, should this loss of information be sufficient to cause incorrect results, the uniform histogram vector can be augmented with additional information of uniform dimensionality describing the original order of the input data.

Practically speaking, and despite the incurred information loss, this functional component has proven itself to be useful when processing estimated object sizes downloaded in an HTTP session - there may be different amounts of objects downloaded (varying dimensionality) but the distribution of object sizes can often accurately distinguish one webpage from the others.

3.4.3 Decision Tree Builder

This functional component receives a *training* dataset and *testing* dataset and constructs a *set* of *decision tree* trained classifiers from the *training* data.

The choice to use a *decision tree* classifier as opposed to other types of machine learning classifiers such as *KNeighbours* or *Naive Bayes* is not considered as a research contribution to this thesis. The *decision tree* type of classifier was chosen as it has empirically shown itself to provide accurate results for the evaluations discussed in this thesis. Switching to a different type of classifier would be a trivial task as the classifier interfaces (eg. training data/labels, testing data/labels) are all identical.

These *descision tree* classifiers provided by this functional component are automatically evaluated based on the provided *testing* data and tuples are generated in the form (T, P) where T is the *true* value provided by the *testing dataset* and P is the value *predicted* by the trained model. Once this set of tuples has been generated, the role of the *machine learning layer* is completed and the workflow proceeds to the *threat modelling layer* where the simulated adversary results are measured up against the program's *security model*.

This component finds itself in all test scenarios as it provides a very *general purpose* role - that is, to determine the set of rules which predict an output from a given input. In the context of this framework, the *inputs* are the *processed*, *adversary observable* behaviours of a system, while the *outputs* are the internal system states deemed to be *private* by the security model.

3.5 Threat Modelling Layer

The role of the *threat modelling layer* is to determine if the results from a trial run of the machine learning adversary are sufficient to violate the requirements set out by the system's *security model*. As every activity conducted by a computer involving a network resource will generate a network traffic pattern, it therefore *must* be determined whether or not the

presence of a specific pattern violates any security requirements. In terms of *Open Web Application Security Project (OWASP)* definitions, the types of threats described in this thesis would be categorized as *sensitive data exposure* [28] and it is therefore the role of the *threat modelling layer* to determine if the information exposed through the network traffic pattern is sensitive or not.

It is here, at this layer, where the framework methods become *protocol specific*. Therefore, when a class of side-channel vulnerability is discovered, for example the ability to predict commands executed over SSH [29], a *threat modelling method* is created which, based on the success of the *machine learning adversary* for predicting the execution of commands, will determine if a side-channel information leak exists for a particular command.

3.5.1 SSH Command Prediction Evaluator

This functional component accepts a *machine learning model* generated from the preceding *machine learning layer* and a *shell command*. Based on the *evaluation run* that is conducted during the construction of the *machine learning adversary*, this functional method returns the ratio of *true positives* to the total number of tests conducted corresponding to the given *shell command*. It is important to note that *only* the *shell command* is considered and not all parameters associated with a label. For example, another parameter, *context*, is included in all SSH event labels. This parameter contains the current *working directory* of the remote shell. Therefore, using this functional component will only return *the probability that a shell command was executed* and not *the probability that a shell command was executed in a given directory*.

3.5.2 HTTPS Page Load Prediction Evaluator

This functional component accepts a *machine learning model* generated from the preceding *machine learning layer* and a *web url*. Using the data from the *evaluation run*, the probability of successful adversarial prediction of a specific URL is returned. Similar to the *SSH*

Command Prediction Evaluator functional component, *only* the *web url* is considered and not other parameters such as the *page title* or a *user's login state*.

3.5.3 VNC Key Press Prediction Evaluator

Working in the same manner as *SSH Command Prediction Evaluator* and *HTTPS Page Load Prediction Evaluator*, this functional component returns the probability of successful adversarial prediction of a given *key* typed on the keyboard in a VNC session.

3.5.4 Mumble Subtitle Index Prediction Evaluator

Also working in the same manner as the above discussed functional components, this functional component returns the probability of successful adversarial prediction of a given phrase being spoken by a *mumble* client.

3.5.5 String Entropy Calculator

The goal of this functional component is to measure the *unpredictability* of an authentication string (such as a password) requested by a system *given its side-channels*. To accomplish this, first the results obtained from the evaluation of a machine learning model at the *machine learning layer* are processed. These results show the mapping between the *actual* percentage of the correct leading symbols of the authentication string *and* the *predicted* percentage of the correct leading symbols of the authentication string given the system's side-channels. Therefore, the machine learning model must be trained with authentication strings of *known leading correctness* (Figure 3.6) and the corresponding traces of side-channel information.

The algorithm for estimating the *unpredictability* of the authentication string then proceeds as follows. The labels describing the *percent leading correctness* of a given trace are considered. For example, suppose that the *label set* is as follows; {`0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%`}. For each *label* in the *label set*, there is a *proba-*

bility that the *side-channel works properly*. Mathematically, the correct functionality of the side-channel for a given *percent leading correctness*, is expressed as $P(True|Predicted)$ where $True$ represents the *event* of the system processing an authentication string of given *percent leading correctness* and $Predicted$ represents the *event* of the trained machine learning classifier returning the given *percent leading correctness*. In order to calculate $P(True|Predicted)$, Bayes' rule (Figure 3.7) [30] is employed on the results obtained from the *evaluation* of the trained classifier. Thus, a list is obtained describing how *effective* the side-channel is at measuring the *percent leading correctness* of an entered string.

```
HelloThere      (Original)
HelloThere      (10% Leading Correctness)
HelloThere      (20% Leading Correctness)
HelloThere      (30% Leading Correctness)
```

Figure 3.6: Examples of 10%, 20%, and 30% *leading correctness* of an authentication string.

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B|A) * P(A) + P(B|\bar{A}) * P(\bar{A})}$$

Figure 3.7: Bayes' theorem finds the probability that an event occurred given the signals that were received.

The next step in the algorithm is to find all *permutations* of *working/non working* side-channel detections and their associated probabilities. For example, consider the previously mentioned *label set* of {`0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%`}, and suppose that the *associated probabilities* are; {`0.2, 0.2, 0.4, 0.8, 0.4, 0.6, 0.9, 0.9, 0.7, 0.7, 1.0`}. This means that an adversary could detect that the leading 0% of the authentication string was correctly entered with a probability of 0.2, that the leading 10% was correctly entered with a probability of 0.2, that the leading 20% was correctly entered with a probability of 0.4 and so forth. Therefore, one of the *permutations* that would be generated is {`F,F,F,F,T,T,T,F,F,T,T`}, implying that the adversary is only able to detect the correctness of the first 40%, first 50%, first 60%, first 90%, and first 100% of

the authentication string. The *associated probability* of this *permutation* therefore would be, $(1-0.2)\times(1-0.2)\times(1-0.4)\times(1-0.8)\times 0.4\times 0.6\times 0.9\times(1-0.9)\times(1-0.7)\times 0.7\times 1.0 = 0.000348$.

The third step in this algorithm, now that the *permutations* and their *associated probabilities* are known, is to calculate the estimated number of guesses required to obtain the correct value of the authentication string. To do so, each section between *working side-channel markers* is considered. For example, if the *permutation* was {`F,F,F,F,T,T,T,F,F,T,T`}, the sections between *working side-channel markers* would be, $(0\%, 40\%)$, $(40\%, 50\%)$, $(50\%, 60\%)$, $(60\%, 90\%)$, $(90\%, 100\%)$. Assuming that it is known in advance that the authentication string is 100 symbols in length, the segments are of the following *respective* lengths 40, 10, 10, 30, 10. Assuming also that the *alphabet* consists of *ten* symbols, the calculation proceeds as follows.

- Trials required for breaking first 40: 10^{40}.
- Trials required for breaking second 10: 10^{10}.
- Trials required for breaking third 10: 10^{10}.
- Trials required for breaking forth 30: 10^{30}.
- Trials required for breaking fifth 10: 10^{10}.
- Total trials required for this permutation: 1.0×10^{40}.
- Total *expected* trials given *permutation probability*: $1.0 \times 10^{40} \times 0.000348 = 3.48 \times 10^{36}$.

This *expected trials calculation* is repeated for *all* permutations and their results are summed together thus obtaining the *expected* number of trials required for breaking the authentication string *under* the side-channel affected system. Lastly, this number of *expected* trials is converted to a corresponding *information entropy* by evaluating the *base-2* logarithm function against it.

To use this functional component, the *side-channel* attack scenario must be able to provide the *trained and evaluated* machine learning model from the preceding layer, the number of symbols in the *alphabet* of the authentication string, as well as the symbol length of the authentication string.

3.6 Reactive Layer

The role of the *reactive layer* is, as implied by the name, to *react* to any differences between a system's security model and the results of the execution of a scenario's *threat modelling layer*. This layer closes the *quality control feedback loop* by providing the *next step* for the mitigation of side-channel information leaks.

3.6.1 Warning Logger

This functional component accepts a `warning message`, a `warning threshold` and the evaluated `threat model score`. If the `threat model score` exceeds the `warning threshold`, the `warning message` is simply printed to the *standard output*. This simple *threshold-based* logic could be extended so as to alert the software developers through a team communication service such as *Slack* [31].

3.6.2 Polar Gauge Renderer

This functional component accepts a `minimum value`, a `maximum value`, a `reading value`, and an output `filename`. This functional component then renders a circular gauge image for the range between the `minimum value` and `maximum value` and draws the reading needle at the `reading value`. This rendered image is then written to the output file specified by `filename`. A popular type of goal for the images rendered by this functional component is the creation of software performance dashboards for *continuous integration* systems.

3.6.3 Bar Graph Renderer

Similar to the *Polar Gauge Renderer*, this functional component accepts a `minimum value`, a `maximum value`, a set of `categories` (x-axis data), a set of `values` (y-axis data) and a `filename`. This functional component renders to the image specified by `filename`, a *bar graph* where the `values` are plotted with respect to the `categories`. Just like the *Polar Gauge Renderer* functional method, the images rendered by this method are also suitable for performance dashboards.

3.6.4 Entropy Target Renderer

This functional component generates an image which visually represents the *reduction* of the *search space* when determining the value of a private variable using side-channel information. This is visualized as a *two colour target* where the *reduced* search space is painted in *green* overtop of the *original* search space (red). The area of each coloured region is proportional to the size of the search space, therefore, if the monitored side-channels reveal little to no *private* information, the info-graphic will be almost entirely *green* but if the monitored side-channels reveal *effective hints* into the value of a *private* variable the generated info-graphic will be, in large part, *red*.

This functional component accepts the values of *expected private variable entropy*, *measured private variable entropy*, a *title*, and the *filename* for which the generated info-graphic will be written to.

3.6.5 Report Webpage Generator

`Report Webpage Generator` is a functional component within the *reactive layer* that allows for the rendering of an HTML webpage based *side-channel health dashboard*. This component allows for the creation of *infocards* which contain a *title* and an *image file* to be displayed. For example, the *title* could be *predictability of executing the command* `ls /dev` and the

image file could be a *gauge image* measuring this *predictability*. After all *infocards* have been created, this functional component renders these *infocards* as a HTML document. This HTML document, and its associated resources, can be served with a static webpage server thus providing a *dashboard* describing the *security performance* of a system.

3.7 Summary

This chapter has presented the *design* of the side-channel detection framework. After describing the *five layers* of concerns for detecting the presence of a side-channel information leak, methods for *data gathering*, *data filtering*, *prediction*, *risk quantification*, and *risk reaction* have been presented. In Chapter 5 (Evaluation), it will be evident how these methods are used to construct test cases for detecting side-channel information leaks in popular deployments. The implementation details are presented in the next chapter.

Chapter 4

Implementation

The presented *side-channel* detection framework was implemented using several open-source tools and libraries. This section of the thesis details these components as, for each component, a summary of its design is presented, along with its role in the presented framework, followed by the reasons for why it was chosen.

4.1 Component Implementation

In Table 4.1, functional components from the *Framework Design* chapter of this thesis are mapped to their implementations which are used for the tests conducted in the *Evaluation* section of this thesis. The table also lists the third-party software packages that were used to realize the designed functional components. The functional components from the *data gathering layer*, however, are not listed in this table as, with the exception of the *Instrumentable Testbench Virtual Machine* which uses *Bochs*, all other *data gathering* functional components are either simple Python scripts (eg. *Bash Shell UDP Tagger*), or are JavaScript browser plugins (eg. *Firefox Addon UDP Tagger*).

Functional Component (Chapter 3)	**Implementation Name Name (Chapter 5)**	**Use of Third Party Software**
Labelled Sequence Extractor	`get_pcap_labelled_sequences`	Scapy
SSH Labelled Sequence Extractor	`get_pcap_ssh_encrypted_sequence`	Scapy
Labels Filter	`filter_labels_list`	Python
NBurst Filter	`NBurst_detector`	Python
Time Density Filter	`packet_timing_density_detector`	Python
Convert Label to Times	`get_label_times`	Python
Convert Labels to Spans	`labels_to_spans`	Python
Label Cluster	`label_cluster`	Python
Balanced Label Data Splitter	`split_data_balanced_labels`	Python
Entity Histogram Creator	`make_entity_histogram`	Python
Decision Tree Builder	`make_decision_tree_model`	scikit-learn
SSH Command Prediction Evaluator	`ssh.evaluate_command_prediction`	Python
HTTPS Page Load Prediction Evaluator	`https.evaluate_page_loaded_prediction`	Python
VNC Key Press Prediction Evaluator	`vnc_over_ssh.evaluate_key_type_prediction`	Python
Mumble Subtitle Index Prediction Evaluator	`mumble.evaluate_subtitle_index_prediction`	Python
String Entropy Calculator	`string_compare.calculate_string_entropy`	Python
Warning Logger	`simple_log_warning`	Python
Polar Gauge Renderer	`polar_gauges.render_gauge_image`	Matplotlib
Bar Graph Renderer	`bar_graph.render_bar_graph`	Matplotlib
Entropy Target Renderer	`entropy_target.render_entropy_target`	Matplotlib
Report Webpage Generator	`ReportWebpage` `add_infocard` `render_html`	Python

Table 4.1: Each functional component described in Chapter 3 is implemented for evaluation in Chapter 5 using the third-party software described in this chapter.

4.2 Docker

Docker is a framework which employs and abstracts capabilities in the Linux kernel to create *containers* which isolate processes from each other and from the regular host processes. In addition to providing *isolation*, Docker's *copy-on-write* mechanism allows containers to be built based on other containers all while saving hard drive space as only the *differences* between the *parent* and *child* container need to be saved to disk [32]. For example, the *parent* image could be the official Docker *node.js* image and the *child* image could be a web application written in *node.js*. The result of the *copy-on-write* mechanism is that only *one* copy of the *node.js* Docker image *as well as* the *differences* between this image and the custom web application image need to be stored to disk. Due to this notion of building more *specific* containers based on more *general* containers, Docker images are typically built using *Dockerfiles* - that is, container description files instructing the Docker engine as to which image it should start with as its *parent* image and what changes need to be made to it such as, adding files, installing packages, or executing commands, so that this newly built container may perform all the necessary roles for the target application.

The role which *Docker* plays in the implementation of the proposed framework is both the provision of Linux hosts each performing the necessary roles for a side-channel attack scenario (*client*, *server*, *adversary*), as well as the packaging of the side-channel data analysis tools, specifically the *feature extraction layer*, *machine learning layer*, *threat modelling layer*, *reactive layer*, and depending upon the attack scenario, the *data gathering layer*.

Considering the framework requirement of simulating the network of a *client*, *server*, and *adversary*, Docker lends itself very well to this task [33]. Specifically, using Docker, a network interface is by default created which can be monitored thus capturing the network traffic that is *exclusively* associated with the container. To illustrate this, suppose a web browser and *only* a web browser is running in a Docker container. By executing the `tcpdump` utility in this Docker container, only the network traffic associated with the web browser, and not the traffic from other processes running on the same physical host, is captured for

later analysis.

Docker also outperforms the use of virtual machines in several ways. First, the computational overhead associated with running an application in a Docker container is much less than that associated with running an application in a virtual machine, as the Docker container need not emulate an underlying hardware stack and an independent operating system kernel. Secondly, the *Docker Hub* provides a rich library of container images and it is highly probable that any popular open source networked application will be available through *Docker Hub*. This same type of image repository does not yet exist for virtual machines. Third, Docker provides a well-defined method for extending pre-existing images by defining the required modifications as instructions in a *Dockerfile*. Although this same goal could be accomplished using an IT automation tool, such as *Ansible* [34], on virtual machines, the result would be a separate large virtual machine image and not simply one additional image layer of much lower size. Lastly, Docker is rising in popularity in comparison to virtual machines, in the cloud computing scene [35]. Given that tools such as *Docker Swarm* or *Kubernetes* [36] are commonly used to automate the deployment of *software defined environments*, using *Docker* in the implementation of this proposed framework facilitates the task of integrating *side-channel monitoring* into the given *software defined environment*.

4.3 Scapy

Scapy is a powerful network packet generation, manipulation, and analysis tool which may be used independently through a *read-evaluate-print-loop* (REPL) shell or by a Python program through the form of a Python module [37]. The network traffic data to be analyzed may be captured directly from a network interface using *Scapy* or it may be read from a *PCAP* file. When analyzing a *PCAP* file, as is done by the proposed framework, *Scapy* exposes the information in this file using the typical means for working with Python data structures. The `rdpcap` method from the *Scapy* module reads a *PCAP* file and returns the ordered set

```
from scapy.all import *
network_traffic = rdpcap('network_sample.pcap')

len(network_traffic)
>>> 81

network_traffic[0]
>>> <Ether  dst=02:42:5a:2a:bb:d6 src=02:42:ac:13:00:02 type=IPv4 |
<IP  version=4L ihl=5L tos=0x0 len=200 id=21197 flags=DF frag=0L ttl=64
 proto=tcp chksum=0x8bd1 src=172.19.0.2 dst=172.217.2.163 options=[] |
<TCP  sport=45348 dport=https seq=1825245643 ack=2040074521 dataofs=8L
 reserved=0L flags=PA window=5030 chksum=0x5c4c urgptr=0
 options=[('NOP', None), ('NOP', None),
 ('Timestamp', (2116361709, 1342950345))] |
<Raw  load="\x17\x03\x03\x00\x8f\x00\x00\x00\x00...\xca" |>>>>

network_traffic[0][TCP]
>>> <TCP  sport=45348 dport=https seq=1825245643 ...

len(network_traffic[0][TCP].payload)
>>> 148

network_traffic[0].time
>>> 1517870219.491492
```

Listing 4.1: Using the Scapy module to analyze a PCAP file of captured network traffic.

of contained network packets as a *Python list*. Each network layer within each packet can be accessed in the same manner as accessing the values held within a *Python dictionary*. The *payloads* are accessed as *Python strings* and therefore passing the *payload string* to Python's `len()` method will return the byte size of the payload (Listing 4.1).

As many of the types of *side-channels* evaluated in this thesis are *network traffic* based side-channels, *Scapy* performs an important role on the *feature extraction layer*. Specifically, when a traffic stream is tagged with UDP packets denoting the beginnings and endings of events, *Scapy* is used along with a *stateful* parser to *slice out* the traffic packets in between the *start* and *end* tags thus building a dataset of traffic *labels* and traffic *samples*. Further-

more, as most of the network traffic which is analyzed in this thesis is in *encrypted* form, the actual *values* of the bytes in the traffic stream are, in general, of little relevance. The more significant features of the traffic, when reading *side-channel* information are the *sizes of the application layer payloads* and the *times of the packets*. In the implementation of the proposed framework, *Scapy* is used for *transforming* a *specific* packet from an encrypted stream sequence into a more *general* representation of a *payload size* or an *event time*. Lastly, for detecting *side-channels*, the framework will have to, at times, analyze *unencrypted* protocols. This typically occurs at the *entry points* and *exit points* of SSH tunnels. Therefore, in the implementation of the proposed framework, *Scapy* is used to write analyzers of plaintext protocols to detect the presence of specific events so that the encrypted traffic stream carrying these events may be tagged with the appropriate event labels and later analyzed.

Due to the fact that *Scapy* is capable of performing all of these required functionalities, is well documented, and interfaces with Python, which will be required for other implementation components, this package was chosen to perform the above discussed roles in the framework.

4.4 Bochs

Bochs is an x86 PC emulator, with a minimalist design requiring no host hardware acceleration thus making it easily portable to different architectures and operating systems [38]. In the implementation of Bochs, every emulated computer functionality (eg. memory access, instruction execution, secondary storage access, etc.) is implemented by a *C++* method and therefore modification of these functionalities is a relatively simple task. In addition to the simplicity of functional modification of the Bochs emulated computer system, the other great advantage provided by the design of Bochs is that the exact amount of emulated CPU instructions can be counted, as no translation occurs, and therefore greater insight can be obtained on *timing* side-channel properties, without interference from CPU resources used

by host processes.

In the proposed framework, *Bochs* is employed whenever the security model of an application requires that the application be free of some hardware based side-channel. Using the modified version of Bochs that is integrated with the implementation of the proposed framework, one may develop scenarios where observable events such as hard drive accesses are monitored and correlated with unobservable events such as specific memory location accesses, thus building a machine learning adversary model describing how well an adversary could predict these private events.

Due to the simple design of Bochs, along with the existence of quality documentation on its modification [39], Bochs was chosen to be the virtual machine used for simulating hardware side-channels in this framework.

4.5 scikit-learn

Scikit-learn is a Python module which provides interfaces for a wide variety of machine learning algorithms [40]. Data is passed to and from these machine learning algorithms using *Numpy* arrays [41] thus allowing for compatibility with many other Python modules as *Numpy* is a commonly used format among scientific computing modules for Python. The *Scikit-learn* module is designed strongly around the concept of *object oriented interfaces*. The *central* object of *Scikit-learn* is the `estimator` object. Simply put, the `estimator` object, is that which receives the input data to be analyzed through its `fit()` method. If the `estimator` is, for example a *dimensionality reducer* such as `sklearn.decomposition.PCA`, a `transform` method will be implemented which will return the set of input feature vectors transformed to a lower-dimensionality space based on the rules learned in the computation of the `fit()` method.

The type of `estimator` which is most commonly used in the proposed framework of this thesis is the *classification* `estimator`. This `estimator` *implements* a `predict()` method

which accepts a set of input feature vectors and returns a set of predicted labels based on the training data that was passed through the objects `fit()` method. By virtue of the design of *Scikit-learn*, *all* supervised learning classification algorithms implement these `fit()` and `predict()` methods and therefore it is simple to switch from one classification algorithm (eg. Decision Tree Learning) to another (eg. KNeighbors).

The role which *Scikit-learn* plays in the proposed framework is the provision of the *classification* machine learning algorithms used by the *machine learning layer*. The *classification* sub-discipline of machine learning is appropriately suited to the task of *side-channel* detection as the result of classifying captured publicly observable system behaviours, and comparing the predicted results to the true *private* events will indicate if a *side-channel* is present.

Due to the simple, well documented interfaces of *Scikit-learn*, as well as its strong interoperability with other modules of the Python language, such as the previously discussed *Scapy*, *Scikit-learn* was chosen to provide the machine learning algorithms for the implementation of the proposed framework.

4.6 Matplotlib

Matplotlib is a Python module which produces *production quality* mathematical figures while following the same interface design patterns as *MATLAB* [42]. Just like *Scikit-learn*, *Matplotlib* is fully *interoperable* with *Numpy* arrays.

The role which *Matplotlib* plays in the implementation of the proposed framework is the generation of visualizations in the *reactive layer*. For example, if it is required to generate a bar graph representing the probability of successful detection of a finite set of events, the *reactive layer* can employ *Matplotlib* to perform this task.

The main reason why *Matplotlib* was chosen and not a solution where *JavaScript* or *CSS* renders the visualizations in a browser is to avoid limiting the generated graphics to

the scope of the web browser. Using *Matplotlib* provides a simple interface to drawing high quality figures that are exportable to a wide variety of formats.

4.7 Interoperability

Lastly, it is important to note that the proposed framework is designed to not be limited to the tools discussed in this section of the thesis.

At the *data gathering layer*, tools could be added to monitor additional sources of *side-channel* information. In [43], Goller et. al. have shown that by using an inexpensive DVB-T *"dongle"* interface, they were able to capture information from the electromagnetic emanations of an Android smartphone running the RSA algorithm that was sufficient to obtain the private key used in this operation. Therefore future expansions to the *data gathering layer* could include interfaces for capturing data from a section of the electromagnetic spectrum using an inexpensive DVB adapter.

Furthermore, after capturing analog data from a *data gathering* source such as a DVB-T adapter, the *wide-band* signal needs to be filtered so that the important features can be extracted and used on machine learning models. For example, the *wide-band* signal could be passed through a *band-pass* filter so that only the range of frequencies that are of interest are analyzed. After the *band of interest* is extracted from the *wide-band* signal it could be further processed for the detection of activity using an offline change point detection algorithm, such as the Python module *ruptures* [44]. These change points could then form a *feature vector* that could be passed up to the *machine learning layer*.

In the design and implementation of the proposed framework, it has up to this point been assumed that the developer of the side-channel attack scenario has domain-specific knowledge of the *side-channel* being exploited and thus is capable of manually deciding which feature extraction algorithms should be used. For example, it is known that when loading a webpage, each object of significant size generates a set of uniformly sized network

payloads. Another example of this is that it is also known that when entering keys into a remote SSH console, each keypress generates a network packet with minimal delay from the time the key is pressed to the time the packet is emitted.

In the future, the *machine learning layer* of the proposed framework could be expanded to include a *neural network* library such as *Keras* [45] so that *data features* need not be manually engineered but rather could be extracted automatically by means of a trained neural network. As the framework implementation is written in Python and *Numpy* is already used for moving data into and out of the *Scikit-learn* modules of the *machine learning layer*, integrating *Keras* into the framework would not require large efforts.

Following in the same vein of *automation*, future improvements to the *reactive layer* could include automatic code generation to mask the *side-channel* information cues. For example, a side-channel attack scenario might reveal that the execution of certain shell commands can be perceived by analyzing the sizes of network payloads. In order to prevent this type of attack, the *reactive layer* could potentially generate obfuscation traffic to hide the *true* private events should this *reactive layer module* be alerted from the *threat modelling layer* of the vulnerability.

4.8 Summary

This chapter has presented the *implementation* of the presented side-channel detection framework. Each software package which plays a *major* role in the framework has been discussed in terms of its role as well as the reasons for why it was chosen. As *cooperation* between framework layers, both presently implemented as well as future work, is a critical property for the presented framework, *interoperability* has been stressed as a concern for both framework design *and* implementation. In the next chapter, the *evaluation* of the implemented framework is presented.

Chapter 5

Evaluation

This chapter presents the evaluation of the layered side-channel detection framework. In this *evaluation* section we demonstrate the complete stack working on five different attack scenarios - analysis of SSH console access traffic, analysis of HTTPS web browsing traffic, analysis of the VNC protocol tunnelled over SSH, analysis of the *Mumble* encrypted VoIP communication system, and analysis of simulated power consumption of a password entry system.

The side-channel attacks presented in this chapter were chosen as they involve commonly used *production ready* software. Furthermore, research has already been conducted involving the presented side-channel attacks. The main research contribution made with these evaluations, in addition to the confirmation of previously conducted research, is the ability to build a framework encompassing these various types of side-channel attacks. When following the presented examples in this chapter, reuse of framework methods can be observed thus demonstrating how the presented framework accelerates the development of side-channel security test cases.

For the scenarios of *SSH console access traffic*, *analysis of HTTPS web browsing traffic*, *analysis of the VNC protocol tunnelled over SSH*, and *analysis of the Mumble encrypted VoIP communication system* the resultant values for probability of successful prediction appear

to be distributed in discrete clusters. This is due to the *deterministic* nature of computer systems - if a computer generates a traffic pattern that is misclassified, it is likely to generate that same traffic pattern which will be consistently misclassified.

5.1 Analysis of SSH Console Access Traffic

It is well known that the timing characteristics of information streams wrapped with SSH are not obfuscated. In this evaluation, the data stream of SSH traffic for a *remote* user interacting with the *Bash* console of a Linux server is investigated.

5.1.1 Data Gathering Layer

In order to train a model with the criteria necessary for determining what the *console user interaction* was, a set of *labels* describing the underlying user interaction associated with each traffic pattern is required. To generate these labels, UDP packets are sent from the *remote* server on port *5006* on the beginnings and endings of each *Bash* command executed thus associating the traffic pattern with a user activity.

5.1.2 Feature Extraction Layer

The next step in this data processing pipeline is to convert this stream of labelled captured network traffic packets into a set of lists where each list in the set corresponds to the sizes of SSH stream packets sent from server to client for a given execution of a Linux command. An *associated set* provides the description of the Linux command associated with each traffic pattern. The code snippet in Listing 5.1 shows how the `get_pcap_labelled_sequences` method from the *feature extraction layer* can be used to produce the associated sets of SSH traffic patterns and their associated *Bash shell* commands.

```
import FeatureExtractionLayer

TAG_UDPPORT = 5006
SSH_STREAM_PORT = 22
PCAP_EXAMPLE = "capture_4.pcap"
LABELS_KEEP_LIST = ["command", "context"]

"""
Use the FeatureExtractionLayer to extract the labeled sequences of
encrypted network traffic.
"""
traffic_labels, traffic_data = \
        FeatureExtractionLayer.get_pcap_labelled_sequences(
                                                        PCAP_EXAMPLE,
                                                        TAG_UDPPORT,
                                                        SSH_STREAM_PORT
                                                        )

"""
Use the FeatureExtractionLayer to simplify the labels
"""
simplified_traffic_labels = list(
                FeatureExtractionLayer.filter_labels_list(
                traffic_labels, LABELS_KEEP_LIST))
```

Listing 5.1: Source code for using the Feature Extraction Layer to extract the list of network traffic features associated with each label.

5.1.3 Machine Learning Layer

After extracting the *features*, the next step in side-channel adversary simulation is to split the extracted features dataset into two distinct sets with one for adversary *training* and the other for adversary *testing*. In this evaluation example, 70% of the dataset was used for *training* while the remaining 30% was used for *testing*. The adversary in this example is simulated by a *decision tree* trained classifier and thus after building this *machine learning adversary model* a list of tuples of the form $(truevalue, predictedvalue)$ is returned. The code in Listing 5.2 describes how the extracted features are split into *training* and *testing* datasets followed by the training of a *decision tree classifier* thus returning the results of a

```
import MachineLearningLayer

"""
Use the MachineLearningLayer to generate a predictive model for labels
given traffic patterns.
"""

#...but first split the data into training and test sets
model_training_data, model_training_labels, \
model_test_data, model_test_labels = \
                MachineLearningLayer.split_data_balanced_labels(
                                        traffic_data,
                                        simplified_traffic_labels,
                                        0.70)

#...train the model
adversary_model = \
        MachineLearningLayer.make_decision_tree_model(
                                                model_training_data,
                                                model_training_labels,
                                                model_test_data,
                                                model_test_labels)
```

Listing 5.2: Source code for using the Machine Learning Layer to create a decision tree classifier to simulate an adversary predicting UNIX commands from SSH traffic patterns.

wire-tapping adversary using this type of attack model.

5.1.4 Threat Modelling Layer

In the *threat modelling layer*, the security requirements of the system are evaluated against the *machine learning adversary model*. During this evaluation, a *security model* has been defined with the requirement that the execution of the UNIX commands $\{pwd, ls, ls/dev\}$ shall not be detectable by a wire-tapping adversary. As shown in Listing 5.3, from the `ssh` submodule of the `ThreatModellingLayer` module, the method `evaluate_command_prediction` is used to return the probability of the *machine learning adversary model* successfully pre-

dicting the execution of the selected UNIX commands.

5.1.5 Reactive Layer

Lastly, after the comparison of the *security requirements* with the *machine learning adversary model*, this proposed framework must react to the result of this comparison. In this evaluation, the *reaction* is to generate a web-based dashboard which graphically describes the probability of successful UNIX command prediction by the simulated adversary. As shown in Listing 5.3, a single call to the `render_gauge_image` method of the `polar_gauges` sub-module of the `ReactiveLayer` module generates an image describing the probability of successful command prediction by the adversary. This image can then be used as a component of a *security dashboard* (Figure 5.1).

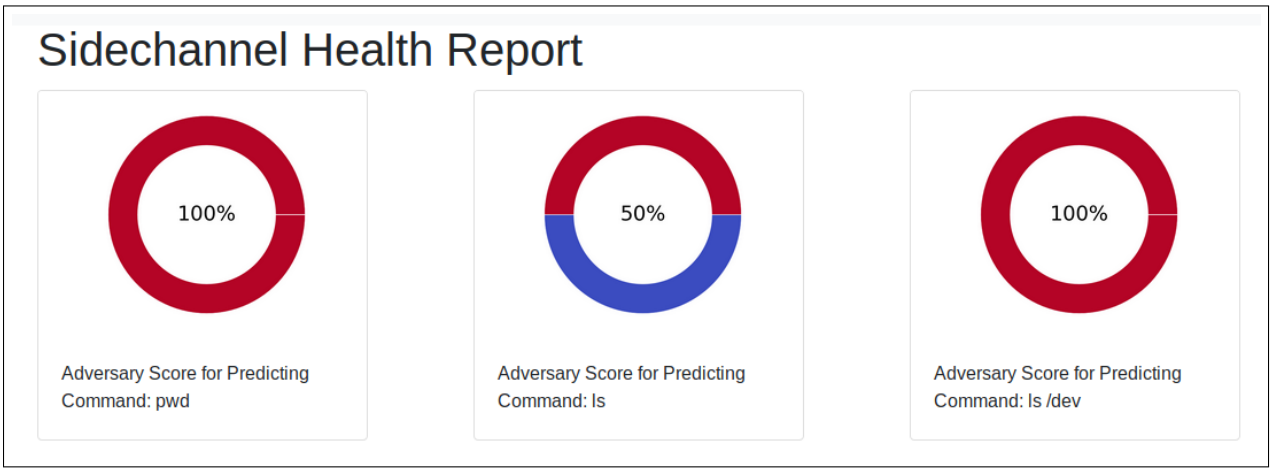

Figure 5.1: The Reactive Layer can render an HTML dashboard describing how well an adversary can learn private information.

```
import ThreatModellingLayer
import ReactiveLayer

"""
Use the ReactiveLayer to render a webpage with gauges showing the
machine learning modelled adversary's success at exploiting the
side-channel cues.
"""

commands = ["pwd", "ls", "ls /dev"]

#Create the object of the report webpage
report_webpage = ReactiveLayer.ReportWebpage()

for cmd in commands:
        cmd_prob = ThreatModellingLayer.ssh.evaluate_command_prediction(
                                                        adversary_model,
                                                        cmd)

        guage_name = "/img/Predict_CMD_{}.png".format(
                                ReactiveLayer.calculate_sha256(
                                cmd))

        ReactiveLayer.polar_guages.render_guage_image(
                        0, 100, int(100*cmd_prob), "/www" + guage_name)

        #Add this information to the report webpage
        report_webpage.add_infocard(
        "Adversary Score for Predicting Command: {}".format(cmd),
        guage_name
        )

#Render the HTML page
report_webpage.render_html("/www/index.html")
```

Listing 5.3: Using the Threat Modelling and Reactive layers to describe the ability of an adversary to predict the execution of the UNIX commands `pwd`, `ls`, and `ls /dev`.

5.2 Analysis of HTTPS Web Browsing Traffic

The design of the HTTPS protocol works solely to encrypt and authenticate the underlying HTTP traffic stream and the properties of hiding when traffic streams begin and end or the amount of data exchanged in a session are not requirements for this protocol. Research has shown that under many circumstances being able to learn the byte size of an HTTP session is sufficient to identify the webpage that was loaded. Furthermore, if the web page that is loaded is a result of a private user interaction (eg. form submission) then the private user interaction can also be learned by the adversary [4]. In this evaluation, the success level of a wire-tapping adversary learning what *Wikipedia* pages were loaded over HTTPS is evaluated.

A singular run of the test is as follows; the user loads the main page of Wikipedia at *https://en.wikipedia.org/wiki/Main_Page*. This evaluation was conducted on *September 17th 2018* when the main page contained a hyperlink to the Wikipedia article on *Toronto* [46]. In this article, the user then clicks on *CN Tower* and loads this article. Next, the user clicks the link for, and visits the article on *First Canadian Place*. Lastly, the user then follows the link for the article *PATH*. The URLs visited in this run are as follows:

- `https://en.wikipedia.org/wiki/Main_Page`
- `https://en.wikipedia.org/wiki/CN_Tower`
- `https://en.wikipedia.org/wiki/First_Canadian_Place`
- `https://en.wikipedia.org/wiki/PATH_(Toronto)`

This test is evaluated for a total of four runs.

5.2.1 Data Gathering Layer

In order to collect *automatically labelled* network traffic samples from the loading of these article pages, the framework Firefox addon was used to inject UDP packets on port *5005*

carrying payloads marking when web page loads begin and when they end along with their associated URLs.

5.2.2 Feature Extraction Layer

The first step of the feature extraction phase taking place in this example is identical to the example of predicting executed UNIX commands. The `get_pcap_labelled_sequences` method is called thus generating a dataset mapping URLs visited to HTTPS stream (port 443) packet sizes moving from server to client. After simplifying the labels extracted from the labelled traffic stream, the second phase of *feature extraction* occurs. In this phase, for each list of packet sizes associated with the loading of a URL, the feature extraction layer is employed to transform this list of packet sizes to a list of approximate sizes of HTTP objects. To accomplish this, the `NBurst_detector` method from the feature extraction layer is used to extract the list of sums of TCP payload sizes for all continuous runs of *1370 bytes* (Listing 5.4).

5.2.3 Machine Learning Layer

As the trained classifiers require the set of input data samples to be of uniform dimensionality, a transformation of HTTP object size lists of various lengths to a feature vector of *fixed dimensionality* is required. To accomplish this, the `make_entity_histogram` from the `MachineLearningLayer` is used. This transforms each list of approximate sizes of session HTTP objects to a *histogram* describing the *size distribution* of objects loaded in a session. As each histogram (vector) has the same length, the entire set of histograms can, as usual, be split into *training* and *testing* datasets to be used for the *training* and *evaluation* of a machine learning classifier. However, to boost accuracy, each uniform length vector is augmented with the *count* of web objects downloaded in a session. (Listing 5.5).

```
import FeatureExtractionLayer

TAG_UDPPORT = 5005
HTTPS_STREAM_PORT = 443
PCAP_EXAMPLE = "wikipedia_eval_09_17_2018.pcap"
LABELS_KEEP_LIST = ["url"]

"""
Use the FeatureExtractionLayer to extract the labeled sequences of
encrypted network traffic.
"""
traffic_labels, traffic_data = \
                FeatureExtractionLayer.get_pcap_labelled_sequences(
                        PCAP_EXAMPLE,
                        TAG_UDPPORT,
                        HTTPS_STREAM_PORT)

"""
Use the FeatureExtractionLayer to simplify the labels
"""
simplified_traffic_labels = list(
        FeatureExtractionLayer.filter_labels_list(
                        traffic_labels,
                        LABELS_KEEP_LIST))

"""
Use the FeatureExtractionLayer to convert a list of payload sizes to a
list of approximate sizes of web objects.
"""

approx_webobj_sizes = []
for i in range(0, len(traffic_data)):
        this_approx = list(
                FeatureExtractionLayer.NBurst_detector(
                        traffic_data[i],
                        1370))

        approx_webobj_sizes.append(this_approx)
```

Listing 5.4: Extracting data features from HTTPS traffic requires the additional step of calculating the approximate sizes of the downloaded HTTP objects.

```
import MachineLearningLayer

"""
Use the MachineLearningLayer to generate a predictive model for
labels (aka URLs) given traffic patterns.
"""

#...convert sizes list to sizes histogram
histogram_sizes = list(
        MachineLearningLayer.make_entity_histogram(
                approx_webobj_sizes))

#Consider also the count of objects downloaded in a session
for i in range(len(approx_webobj_sizes)):
        histogram_sizes[i].insert(
                0,
                len(approx_webobj_sizes[i]))

#...but first split the data into training and test sets
model_training_data, model_training_labels,\
model_test_data, model_test_labels =\
        MachineLearningLayer.split_data_balanced_labels(
                histogram_sizes,
                simplified_traffic_labels,
                0.50)

#Create the machine-learning adversary model
adversary_model = MachineLearningLayer.make_decision_tree_model(
        model_training_data,
        model_training_labels,
        model_test_data,
        model_test_labels)
```

Listing 5.5: Before applying a machine learning classifier to the list of approximate sizes of HTTP objects, each list is transformed to an object size histogram augmented with the count of objects downloaded in a session.

5.2.4 Threat Modelling Layer

The *security* model for this example is relatively simple - based on the traffic patterns generated from the loading of each of these four articles, an adversary should not be able to

determine which one was loaded beyond the random guess probability of *25%*. Therefore, the *threat modelling layer* in this example simply calculates the probability of an adversary correctly guessing which article was loaded.

5.2.5 Reactive Layer

Similar to the example where the execution of UNIX commands was predicted by a machine learning simulated wire-tapping adversary, the *reactive layer* for this example renders an HTML dashboard showing the probabilities of successful *article URL* prediction by a similar type of adversary (Figure 5.2).

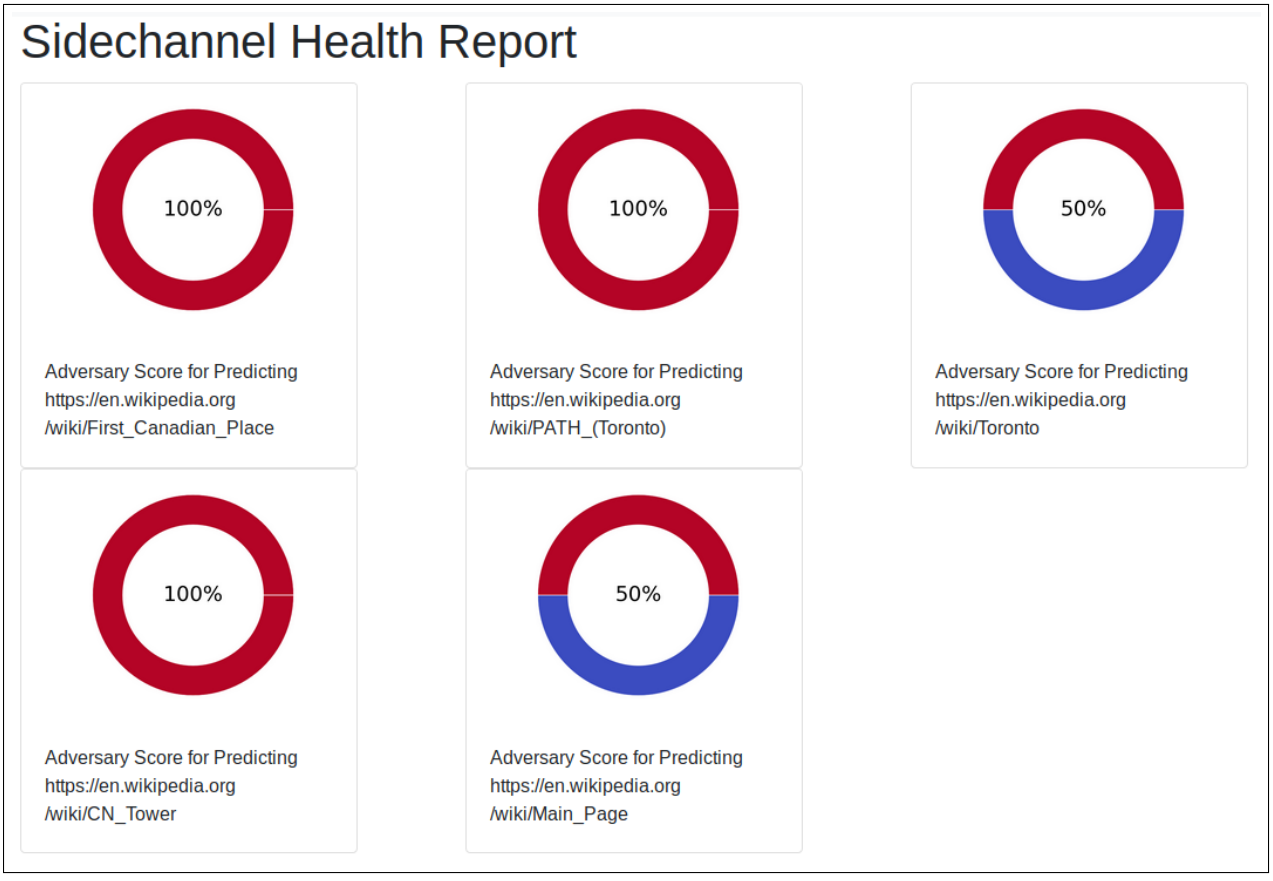

Figure 5.2: Similar to the UNIX command prediction example, this dashboard displays how well an adversary can predict which Wikipedia article was loaded.

5.3 Analysis of SSH Tunnelled VNC Traffic

In this evaluation, the security of the popular practice of tunnelling *unencrypted* Virtual Network Computing (VNC) traffic over the *encrypted* SSH protocol is examined. When designing this attack scenario it is important to keep in mind that SSH tunnelling of TCP connections does *not* hide their *timing* or *size* properties. The design of the VNC protocol must also be considered. As the VNC protocol is designed to be *bandwidth efficient*, only the sections of the remote desktop screen which have *changed* are updated. To further improve bandwidth efficiency, these sections of the screen which are updated, referred to as *tiles*, are transmitted in compressed form. While this protocol design is effective for minimizing required network bandwidth, it suffers from a major security flaw. Due to the fact that the region to be changed of a screen can vary greatly in size, the amount of network traffic generated strongly correlates with the amount of space updated on the screen. Also, if the geometric size of the updated region of the screen is known *a priori*, the amount of network traffic generated is likely to suggest what the screen update was as the image compression algorithm is likely to assign unique data sizes to different image tiles of the same geometric size. In this evaluation, the ability for an adversary to determine which character was typed into a text editor over SSH tunnelled VNC is evaluated.

5.3.1 Data Gathering Layer

The setup for this network security experiment is as follows. A Docker container representing the *remote desktop* runs the *Xvnc* server as well as the *openssh* server. This container also runs the *Geany* text editor so that a connecting client may be able to type characters which appear on-screen. In addition, there is another Docker container which simply runs an *openssh* client. The complete setup is that from the host computer, a VNC client makes an *unencrypted* connection to this *openssh* client container. This connection is received by the *openssh* client where it is sent over SSH tunnel to the Docker container running *Xvnc*. The

result is that by running *tcpdump* in the *openssh* client container, one may observe *both* the plaintext protocol data from the host *as well as* the SSH tunnelled data as it moves to and from the *remote desktop* container.

As explained in all other side-channel detection examples, a set of labels mapping user interaction events to *adversary-observable* network traffic patterns is required. Taking advantage of the fact that both the plaintext *and* SSH tunnelled versions of the VNC communication are available, the role of the *data gathering layer* in this scenario is to, by parsing the plaintext VNC traffic, determine when the *keypress* and *keyrelease* events occur, which key was pressed/released, and extract the *encrypted* server to client network traffic associated with these events.

5.3.2 Feature Extraction Layer

In order to build the required dataset mapping *GUI interaction* to generated encrypted network traffic, the `get_pcap_ssh_encrypted_sequence` method from the `FeatureExtractionLayer` module is employed. The result from the execution of this method is used to fill a dictionary mapping *characters typed* to byte sizes of generated encrypted network traffic (Listing 5.6).

5.3.3 Machine Learning Layer

The *machine learning layer* for this scenario is built in the same way as the scenarios for the *SSH console* side-channel and the *HTTPS web browsing* side-channel. The dataset of keypresses and associated encrypted network traffic patterns is split into *training* and *testing* sets and a *decision tree classifier* is evaluated to predict how successful an adversary would be at predicting which key was typed over an SSH encrypted VNC session.

5.3.4 Threat Modelling Layer

The *threat modelling layer* for this scenario is simple. The *generated machine learning adversary model* is queried for all letters a through z for the probability of successfully predicting the character typed (Listing 5.7).

5.3.5 Reactive Layer

The *reactive layer* for this scenario simply uses the `render_bar_graph` method from the `bar_graph` sub-module of the `ReactiveLayer` module (Listing 5.8) to draw a bar graph describing the probability of successful keypress prediction by a wire-tapping adversary (Figure 5.3).

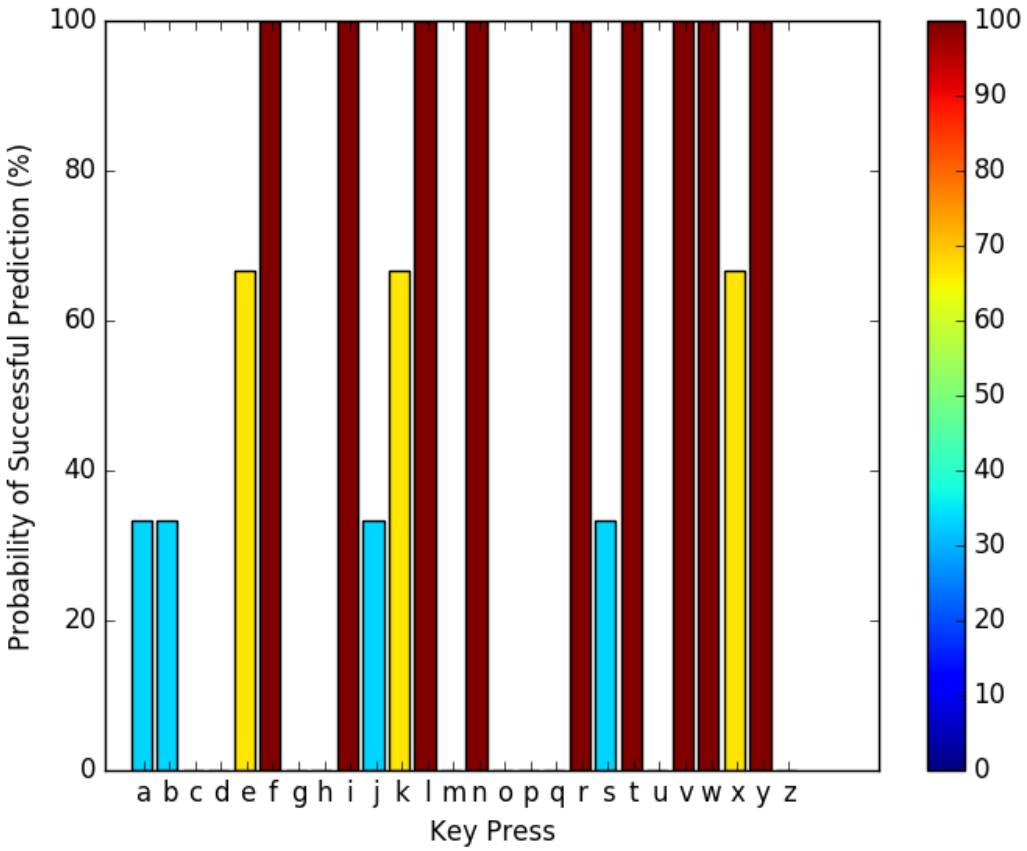

Figure 5.3: The probability of success of a wire-tapping adversary predicting keys typed based on observing encrypted network traffic features.

```
import DataGatheringLayer
import FeatureExtractionLayer

PCAP_EXAMPLE = "vnc_geany36_alphabet.pcap"

"""
Use the FeatureExtractionLayer to get all sequences of sizes of SSH
packet payloads and their corresponding plaintext labels.
"""
PLAIN_SRC = ("172.17.0.1", "*")
PLAIN_DST = ("172.17.0.3", 5900)
CRYPTO_SRC = ("172.17.0.2", "*")
CRYPTO_DST = ("172.17.0.3", "*")

#Label encrypted packets based on plaintext data
labelled_encrypted_events = \
        FeatureExtractionLayer.get_pcap_ssh_encrypted_sequence(
                        PCAP_EXAMPLE,
                        PLAIN_SRC,
                        PLAIN_DST,
                        CRYPTO_SRC,
                        CRYPTO_DST,
                        DataGatheringLayer.vnc_rfb.key_event_tagger)

key_dict = {}

for datum in labelled_encrypted_events:
        keypress = datum[0][1]
        bytes_exchanged = sum(datum[1])
        if keypress not in key_dict:
                key_dict[keypress] = []
        key_dict[keypress].append(bytes_exchanged)

simplified_labels = []
simplified_data = []

for k in "abcdefghijklmnopqrstuvwxyz":
        for n in key_dict[k]:
                simplified_labels.append(k)
                simplified_data.append([n])
```

Listing 5.6: Using the `get_pcap_ssh_encrypted_sequence` method to build a table mapping keypresses to amounts of received encrypted network traffic.

```
import ThreatModellingLayer

adv_performances = {}
for k in "abcdefghijklmnopqrstuvwxyz":
        adv_performances[k] =\
         ThreatModellingLayer.vnc_over_ssh.evaluate_key_type_prediction(
                adversary_model, k)

alphabet = []
perfs = []
for k in "abcdefghijklmnopqrstuvwxyz":
        alphabet.append(k)
        perfs.append(100*adv_performances[k])
```

Listing 5.7: Determining the probability of successful keypress prediction.

```
import ReactiveLayer

#Create a graph of the predictability of each character
ReactiveLayer.bar_graph.render_bar_graph(
        0,
        100,
        alphabet,
        perfs,
        "Key Press",
        "Probability of Successful Prediction (%)",
        "/www/ssh_keypress_predict.png")
```

Listing 5.8: Drawing the graph of character predictability.

5.4 Analysis of Mumble VoIP Traffic

In this evaluation, network traffic from a container running the *Mumble* VoIP server (*murmur*) is analyzed. Mumble uses a client-server architecture with all traffic flows between client and server protected by SSL. During initial investigation of the protocol, it was observed that a *fixed-bitrate* audio codec was used (Figure 5.4) thus demonstrating an awareness of side-channels on the part of the *Mumble* developers as a *variable-bitrate* audio codec could leak information on the content of the conversation through analysis of the instantaneous bitrate [47]. Despite the sensible decision to use *fixed-bitrate* audio codecs there is still an opportunity for side-channels that needs to be considered. Mumble clients only transmit when the user is talking. This could happen *manually* such as requiring the user to press and hold a key while talking or it could happen *automatically* by means such as measuring the raw amplitude level (Figure 5.5). The result of this is that now there are discrete *bursts* of network activity with time durations closely linked to the time durations of speech segments.

No.	Time	Source	Destination	Protocol	Length	Info
125	66.377849	172.19.0.4	172.19.0.2	TLSv1	284	Application Data, Application Data
126	66.377872	172.19.0.2	172.19.0.4	TCP	66	64738 → 55976 [ACK] Seq=1976 Ack=2372 Win=36480 Len=0 TSva
128	66.389040	172.19.0.4	172.19.0.2	TLSv1	284	Application Data, Application Data
129	66.389055	172.19.0.2	172.19.0.4	TCP	66	64738 → 55976 [ACK] Seq=1976 Ack=2590 Win=37632 Len=0 TSva
131	66.409894	172.19.0.4	172.19.0.2	TLSv1	284	Application Data, Application Data
132	66.409909	172.19.0.2	172.19.0.4	TCP	66	64738 → 55976 [ACK] Seq=1976 Ack=2808 Win=38656 Len=0 TSva
134	66.432713	172.19.0.4	172.19.0.2	TLSv1	284	Application Data, Application Data
135	66.432729	172.19.0.2	172.19.0.4	TCP	66	64738 → 55976 [ACK] Seq=1976 Ack=3026 Win=39680 Len=0 TSva
137	66.455359	172.19.0.4	172.19.0.2	TLSv1	284	Application Data, Application Data
138	66.455365	172.19.0.2	172.19.0.4	TCP	66	64738 → 55976 [ACK] Seq=1976 Ack=3244 Win=40832 Len=0 TSva
140	66.466124	172.19.0.4	172.19.0.2	TLSv1	284	Application Data, Application Data
141	66.466130	172.19.0.2	172.19.0.4	TCP	66	64738 → 55976 [ACK] Seq=1976 Ack=3462 Win=41856 Len=0 TSva
143	66.486721	172.19.0.4	172.19.0.2	TLSv1	284	Application Data, Application Data
144	66.486727	172.19.0.2	172.19.0.4	TCP	66	64738 → 55976 [ACK] Seq=1976 Ack=3680 Win=43008 Len=0 TSva
146	66.507360	172.19.0.4	172.19.0.2	TLSv1	284	Application Data, Application Data
147	66.507366	172.19.0.2	172.19.0.4	TCP	66	64738 → 55976 [ACK] Seq=1976 Ack=3898 Win=44032 Len=0 TSva
149	66.528015	172.19.0.4	172.19.0.2	TLSv1	284	Application Data, Application Data
150	66.528022	172.19.0.2	172.19.0.4	TCP	66	64738 → 55976 [ACK] Seq=1976 Ack=4116 Win=45056 Len=0 TSva
152	66.548739	172.19.0.4	172.19.0.2	TLSv1	284	Application Data, Application Data

Figure 5.4: Information flows from the *bot* client to the *murmur* server at a *fixed* bitrate.

In order to evaluate what information could be learned from this type of side-channel, a *bot* client was written in Python using the *pymumble* module to communicate with the *murmur* server. This *bot* client streamed the audio from the *TED Talk How tech companies deceive you into giving up your data and privacy* by *Finn Myrstad* [48]. Streaming occurred

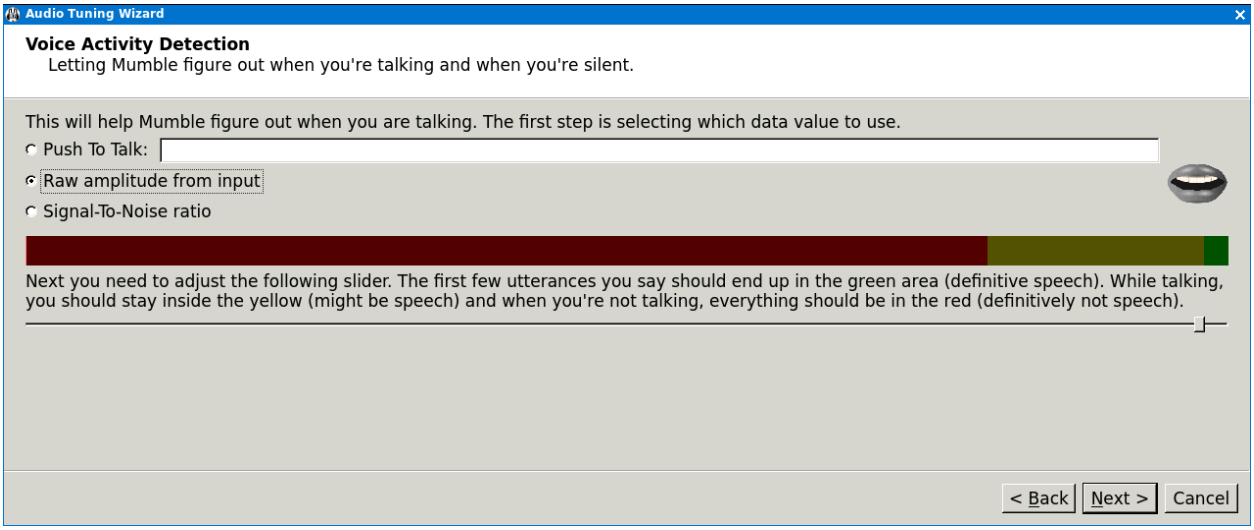

Figure 5.5: The Mumble client can be configured to only transmit when audio *amplitude* exceeds a *threshold* level.

with each packet containing 20ms of audio. In order to simulate the *Mumble* raw amplitude detection setting, a rule is defined that if the *average absolute value* of PCM data points in a packet exceeds 100, the packet is considered to contain voice information and is transmitted, otherwise the packet is assumed to be silence and is not transmitted. With this configuration in place, the *TED Talk* audio was looped for three iterations while network traffic at the *murmur* served was recorded.

5.4.1 Data Gathering Layer

As always, in order to train a machine learning classifier to simulate a side-channel adversary, a label for each traffic pattern is required. In order to label this traffic stream, the subtitles file for the *TED Talk* is used [49]. This subtitles file is parsed and converted to a JSON document denoting at which PCM sample a given subtitle begins and at which sample the given subtitle ends. This JSON document is then used in the mumble *bot* client to transmit UDP packets marking the beginnings and endings of speech sections within the stream of encrypted network traffic.

5.4.2 Feature Extraction Layer

The features used for training the machine learning adversary in this scenario are the bursts of network activity that flow from *bot* client to *murmur* server. To implement this type of detection, the method `packet_timing_density_detector` from the `FeatureExtractionLayer` module is used. Through experimentation, it was determined that the optimal parameters for this function are a *packet time spacing* of $20ms$ with a *tolerance* of $\pm 650\%$. The $20ms$ *spacing* is a reasonable conclusion as, in this test, *Mumble* was configured to contain $20ms$ of audio per network packet. With a *tolerance* of $\pm 650\%$ this implies that any gap larger than $150ms$ is considered to denote the beginning for a new phrase, and is thus also a reasonable conclusion.

After these *clusters* of network activity are extracted, the captured traffic stream is searched to see if a given cluster lies *between* the start and end of a subtitle section. The dataset is then built mapping each *subtitle label* to a possible network traffic cluster pattern (Listing 5.9).

5.4.3 Machine Learning Layer

This dataset is then split into *training* and *testing* subsets where each *unique* subtitle label is used for *training* and the remaining data is used for *testing* (Listing 5.10). A *decision tree classifier* is then built to measure how well an adversary could predict the subtitle of a traffic burst *given* that they were able to learn the initial mapping of traffic patterns to subtitle labels.

5.4.4 Threat Modelling Layer

The *security model* of this system demands that a wire-tapping adversary should not be able to know the corresponding subtitle given a sample of encrypted network traffic. In this evaluation, the probability of successful subtitle prediction by a wire-tapping adversary is

evaluated. This is done through the `evaluate_subtitle_index_prediction` method of the `mumble` sub-module from the `ThreatModellingLayer` module.

5.4.5 Reactive Layer

Similar to the *reactive layer* from the SSH tunnelled VNC example, the *reactive layer* in this scenario also uses the `render_bar_graph` method to draw a bar graph of the *predictability* of each subtitle (Figure 5.6). Due to the large size of the set of subtitles, the subtitle index number is not printed and the border from each bar is removed.

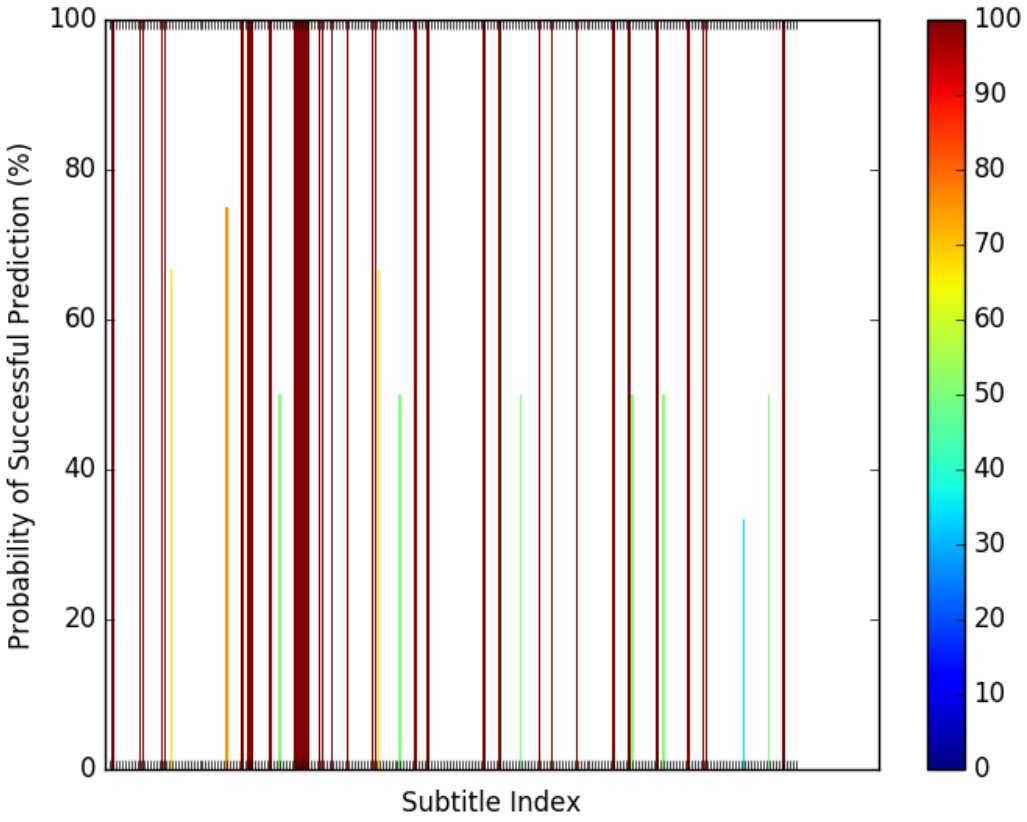

Figure 5.6: By classifying based on traffic burst length, several subtitles can be accurately predicted by a wire-tapping adversary. Tick marks begin at subtitle 1 and end at subtitle 224.

```
import FeatureExtractionLayer
from scapy.all import IP,TCP

PCAP_EXAMPLE = "ted_talk_mumble_packets.pcap"

#Break this file down into time-based clusters of 20ms
def voip_filter_method(pkt):
        if pkt.haslayer(TCP):
                if pkt[IP].src == "172.19.0.4":
                        if pkt[IP].dst == "172.19.0.2":
                                if pkt[TCP].dport == 64738:
                                        return True
        return False

#Get the labels out of the stream
stream_labels = []
for lbl in FeatureExtractionLayer.get_label_times(PCAP_EXAMPLE, 5006):
        stream_labels.append(lbl)

timespans = []
for tsp in FeatureExtractionLayer.labels_to_spans(stream_labels):
        timespans.append(tsp)

ml_data = []
ml_labels = []

#20ms -/+ 650%
for cluster in FeatureExtractionLayer.packet_timing_density_detector(
                                                PCAP_EXAMPLE,
                                                voip_filter_method,
                                                0.02,
                                                6.5):
        ti = cluster[0].time
        tf = cluster[-1].time
        #Get the possible labels based on ti, tf
        for possible_lbl in FeatureExtractionLayer.label_cluster(ti,
                                                        tf,
                                                        timespans):
                ml_data.append(len(cluster))
                ml_labels.append(possible_lbl['subtitle'])
```

Listing 5.9: Extracting the traffic bursts and labels from the capture file to later be used as model training/evaluation features.

```
import MachineLearningLayer

model_training_data = []
model_training_labels = []
model_test_data = []
model_test_labels = []

#Get subtitles up to 223 into training set, everything else in test set
isTraining = True
prevVal = 0
for i in range(len(ml_data)):
        if ml_labels[i] != 223 and prevVal == 223:
                isTraining = False

        if isTraining:
                model_training_data.append([ml_data[i]])
                model_training_labels.append(ml_labels[i])
        else:
                model_test_data.append([ml_data[i]])
                model_test_labels.append(ml_labels[i])

        prevVal = ml_labels[i]

#Create the machine-learning adversary model
adversary_model = MachineLearningLayer.make_decision_tree_model(
                                        model_training_data,
                                        model_training_labels,
                                        model_test_data,
                                        model_test_labels
                                        )
```

Listing 5.10: Using one unique example of the traffic pattern for each subtitle to train a *decision tree classifier*.

5.5 Analysis of CPU Timing when Entering Password

It is well known that a string comparison algorithm which compares two strings character-by-character and returns *false* at the first inequality of characters (Listing 5.11) is vulnerable to a *timing side-channel* attack as a greater equality of leading characters yields longer execution times [50]. Execution time, however, is not always easy to measure. For example, if the process under attack is a *suid* privileged process, an *unprivileged* user may execute it with the UNIX utility `time` in order to learn which code paths are executed based on execution timing information. This is an example of a simple measurement of processor timing. Other *more complicated*, albeit *completely software-based* side-channel attacks involve the measurement of time from an *unprivileged* process to indirectly measure the timing of a *privileged process* through scheduler interactions [51] [52].

Consider now *embedded systems*, where the input/output to/from an adversarial user is limited and command shells, even *unprivileged ones* are unavailable. To cope with this constraint, research has shown that *power side-channels* can be exploited to expose the section of code which the CPU is executing based on its pattern of power consumption [53]. Specifically, *power side-channels* become very effective when a CPU executes a `HALT` instruction as the purpose of this instruction is to place the CPU into a *low power* mode where no instructions are executed until it is woken up and brought back into *operating mode* by an interrupt. Operating systems designers employ this `HALT` instruction functionality so that when the operating system is *idle*, power consumed and heat generated is *minimized* [54].

The result is that the *power* consumed by a processor can be divided into two *modes*, *operational* and *idle*, where the *transition* from *operational* to *idle* occurs on a `HALT` instruction and the *transition* from *idle* to *operational* occurs on an interrupt. Through the measurement of time in the *operational* modes, an adversary may be able to learn which code sections were executed based on their execution times.

Considering the *timing side-channel* associated with the above discussed *string compar-*

```
char check_valid(char* reference_valid, char* check_this)
{
        int i;
        for (i = 0; i < 1000; i++)
        {
                if (reference_valid[i] != check_this[i])
                {
                        return 0;
                }
        }
        return 1;
}
```

Listing 5.11: The algorithm used to compare strings in this evaluation example leaks information based on execution time.

ison algorithm, this evaluation example will conduct a simulation of this algorithm running on an *electronic door lock* based on a minimal *Linux* based operating system. The system is designed so that in order to *unlock* the door, a *1000 character* long string must be entered - such as from the scanning of a barcode. Using simulated *power side-channel analysis*, this evaluation example estimates the true *information entropy* of the required authentication string given the information provided to the adversary through the *power side-channel*.

5.5.1 Data Gathering Layer

In order to gather the data describing the amount of instructions executed between *idle* modes, the *Instrumentable Testbench Virtual Machine* is employed and is configured with a *controllable option* which, when enabled, at each `HALT` instruction, the amount of CPU instructions executed since the previous `HALT` instruction is logged to an internal list data structure. The controlling of this option can be done via UDP packet thus allowing it to be automated by external programs (ie. potential continuous integration tests). When the state of the *controllable option* switches from *on* to *off*, the *maximum* value of the *filtered* set of execution times is logged to a file. In this *data gathering* task, *filtered* refers to only

keeping execution times in the range of 20000 to 40000 instructions or greater than 60000 instructions. These numbers were *empirically* chosen as they are effective at filtering out *idle* Linux system activity (Figure 5.7, Figure 5.8).

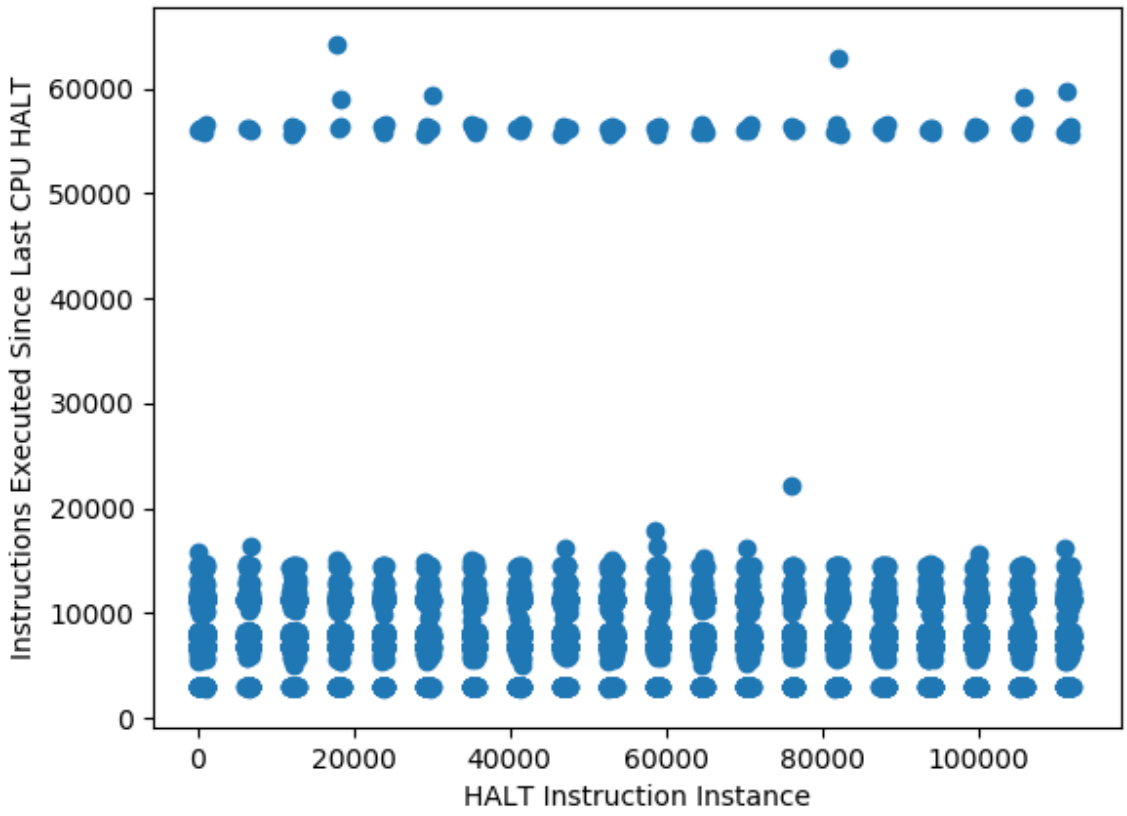

Figure 5.7: When the minimal Linux operating system is idle, lengths of sections of instructions processed between `HALT`s can be clustered in the regions of 0 to 20000 instructions and around 57500 instructions.

After gathering the *execution time data feature* for each *string verification*, the *labels* are already known from the order which the tests were conducted. The *data features* and *labels* are then organized into separate lists to be later used at the *machine learning layer* for training the modelled adversary (Listing 5.13).

The source code for capturing the amount of CPU instructions executed between `HALT` instructions is shown in Listing 5.12.

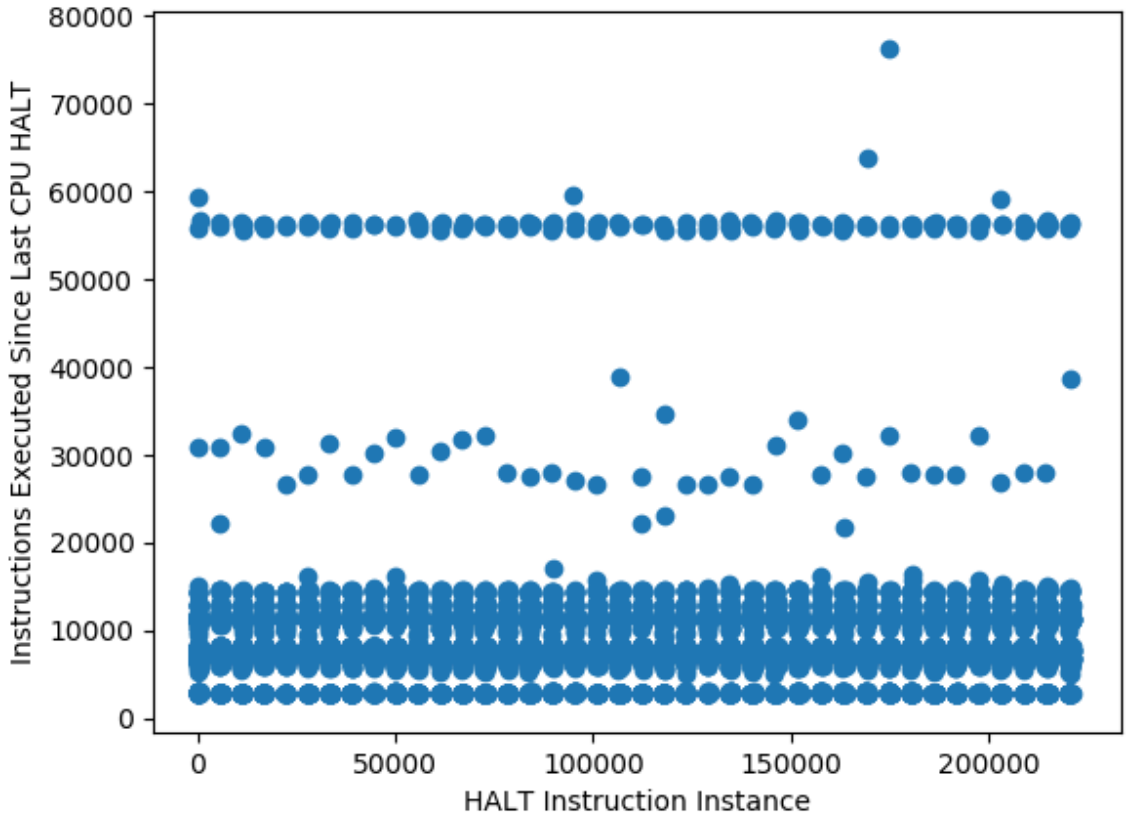

Figure 5.8: When the minimal Linux operating system is executing a CPU consuming userspace process, instruction segments of lengths that lie outside the *idle* cluster occur.

5.5.2 Feature Extraction Layer

As all necessary *feature extraction* occurs in the previously described *Data Gathering Layer* (Listing 5.12), no feature extraction code is present in this *scenario* file. After the filtered lengths of instruction sequences have been generated by the instrumented virtual machine in the *Data Gathering Layer*, this data can be immediately passed to the *Machine Learning Layer*.

5.5.3 Machine Learning Layer

The *machine learning layer* in this *side-channel* attack scenario follows a standard form of splitting the dataset into 50% *training* and 50% *testing* and building a *decision tree classifier* to simulate the adversary.

5.5.4 Threat Modelling Layer

The *threat modelling layer* in this *side-channel* attack scenario is the first example in this thesis to use the `calculate_string_entropy` method. Although the implementation of this method is complicated, its use is simple as it only requires the *trained* machine learning adversary, the amount of symbols in the *private* string's *alphabet* and the *length* of the *private* string. In just *one* line of code, the proposed framework can return an estimate of the true entropy of a private system variable given the system's *modelled* side-channels (Listing 5.14).

5.5.5 Reactive Layer

The *reactive layer* of this side-channel attack scenario calculates the *ideal* entropy of the private system variable based on its length and then employs the `render_entropy_target` method to visualize this difference in information entropy (Listing 5.15, Figure 5.9).

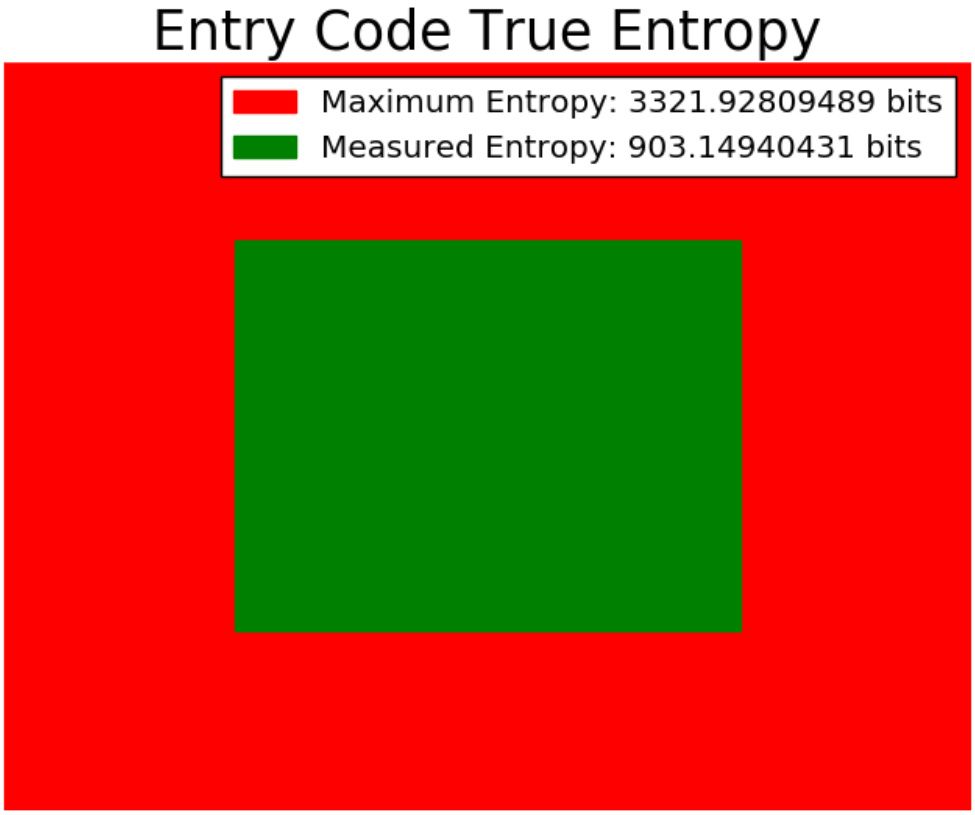

Figure 5.9: The rendered info-graphic shows that the entropy of the authentication code is greatly reduced through side-channels as the *green* area is much smaller than the *red* area.

```
import BochsFaultInjection, RemoteControllers
f_halt_log = open("filtered_halts.log", 'w')
my_machine = BochsFaultInjection.x86_64_Machine()
...
main_hdd = BochsFaultInjection.VirtualHardDrive("ubuntu_base.img")
...
#Create the Remote instrumentation controller
remote_log_controller = RemoteControllers.UDPController(
                '127.0.0.1', 10777, {'log': 0})

#Create a method for catching the HLT instruction
log_block_instance = prev_shouldLog = 0
ins_counts = []
def halt_debug(ins_execd):
        global log_block_instance, prev_shouldLog, ins_counts
        shouldLog = remote_log_controller.get_control_var('log')
        if shouldLog == 1:
                if prev_shouldLog != shouldLog:
                        log_block_instance += 1
                ins_counts.append(ins_execd)
        elif shouldLog == 0:
                if prev_shouldLog != shouldLog:
                        total_inst = ins_counts[:]
                        ins_counts = processing_gaps = []
                        for i in range(0, len(total_inst)):
                                if i == 0:
                                        continue
                                processing_gaps.append(
                                total_inst[i] - total_inst[i-1]
                                )
                        filt_processing_gaps = [-1]
                        for pg in processing_gaps:
                                if pg in range(20000, 40000) or pg>60000:
                                        filt_processing_gaps.append(pg)
                        f_halt_log.write(str(max(filt_processing_gaps))\
                                + "\n")
                        f_halt_log.flush()
        prev_shouldLog = shouldLog
#Set this callback method
my_machine.set_halt_callback(halt_debug)
my_machine.start()
```

Listing 5.12: This Python script, when executed, runs a virtual machine running a minimal Linux operating system and logs the `HALT` events of interest to a file.

```
"""
In the Data Gathering Layer, we will simply open the halt logging file
and create a 'labels' list and a 'data' list.
"""

CPU_INSTRUMENTATION_LOGFILE = "filtered_halts.log"
PASSWORD_CORRECTNESS_ATTEMPTS = []
for i in range(4):
        for j in [0,10,20,30,40,50,60,70,80,90,100]:
                PASSWORD_CORRECTNESS_ATTEMPTS.append(j)


cpu_inst_labels = []
cpu_inst_data = []

f_cpuinst = open(CPU_INSTRUMENTATION_LOGFILE, 'r')
line = f_cpuinst.readline()
cursor_x = 0
while True:
        if not line:
                break
        if line == "\n":
                break

        inst_count = int(line.rstrip())
        if inst_count == -1:
                #There was an issue with the logging process,
                #skip this sample.
                line = f_cpuinst.readline()
                cursor_x += 1
                continue

        cpu_inst_labels.append(PASSWORD_CORRECTNESS_ATTEMPTS[cursor_x])
        cpu_inst_data.append([inst_count])

        cursor_x += 1
        line = f_cpuinst.readline()

f_cpuinst.close()
```

Listing 5.13: In the *data gathering layer* for this side-channel attack scenario, the log file generated from Listing 5.12 is used for populating the `cpu_inst_labels` and `cpu_inst_data` lists which are later used for training the machine learning model adversary.

```
import ThreatModellingLayer

"""
In the Threat Modelling Layer we will measure the expected amount of
trials to crack the password under the given side-channels.
"""

measured_entropy =\
 ThreatModellingLayer.string_compare.calculate_string_entropy(
        adversary_model,
        10,
        1000
        )
```

Listing 5.14: The `calculate_string_entropy` method is used to estimate the true entropy of a private system variable given its side-channels.

```
import ReactiveLayer
import math

"""
In the Reactive Layer we will render an image depicting the 'true'
entropy of the password entry.
"""

max_pw_entropy = math.log(10 ** 1000, 2)

#Render!
ReactiveLayer.entropy_target.render_entropy_target(
                max_pw_entropy,
                measured_entropy,
                "Entry Code True Entropy",
                "/www/passcode_entropy.png"
                )
```

Listing 5.15: The *reactive layer* in this side-channel attack scenario creates a visual report on the estimated true entropy of the required authentication code data.

5.6 Summary

This chapter has presented the *evaluation* of the side-channel detection framework and the evaluation results have shown the presence of private information leaking side-channels in popular system configurations. In addition, source code and detailed descriptions of each test case have been provided thus demonstrating the interworking of framework layers. The following chapter concludes the thesis and presents future work.

Chapter 6

Conclusion and Future Work

In *Chapter 1* of this thesis, the scope of the problem and proposed solution has been laid out. Specifically, the *introduction* section of this thesis has explained how the removal of *private* information leaking side-channels from a computer system is a highly non-trivial task and the accidental creation of these side-channels is much more difficult to prevent compared to simpler types of vulnerabilities such as SQL or OS command injections.

Chapter 2 has presented the *related work* from this thesis' literature review. The discussion of the *related work* has included both research describing patterns common to side-channels found in *production-ready* systems by security researchers, *as well as*, methodologies used by researchers for *automated* or *semi-automated* side-channel detection.

Chapter 3 forms the core of this thesis presenting the *design* of a five-layer framework employing *machine learning* algorithms to assist in the creation and evaluation of hypothetical side-channel attack scenarios and the measurement of their results against a system's *threat model*. The *design* emphasizes *modularity* so that large parts of code used to demonstrate the exploitation of one side-channel vulnerability can be easily adapted and used for the simulation of another side-channel vulnerability. This quality of *modularity* helps to solve one of the discussed problems from the *related work* literature review as the reproducibility of side-channel information leaks as well as the ability to share exploit code among security

researchers are greatly improved.

Chapter 4 describes the *implementation* of the proposed *framework*. Specifically, it has outlined the *software packages* that were selected for the construction of the framework while placing an emphasis on the *ability* to extend the framework in future work as a motivation for the choice of packages made.

Chapter 5 presented an *in-depth* evaluation of the *implementation* of the presented framework. Specifically, *Chapter 5* has presented detailed side-channel attack scenarios for; *analyzing SSH traffic for console access of a remote server*, *analyzing TLS protected web browsing traffic*, *analyzing VNC remote desktop traffic tunnelled over SSH transport*, *analyzing encrypted VoIP traffic from the Mumble system*, *analyzing power consumption of a simulated embedded Linux system running a password checking algorithm.*

The main contributions of this thesis are the *design*, *implementation*, and *evaluation* of a side-channel detection framework. The *evaluation* of this framework has shown that it is indeed effective at detecting *critical* side-channel information leaks from common software system configurations, thus providing an additional research contribution of exposing the security flaws in these *often thought to be secure* configurations. In spite of these promising results, there remains many goals which if met, would further enhance the *effectiveness* of the framework. For example, all evaluations conducted in this thesis have employed *software-in-the-loop (SIL)* testing. For the side-channels that are the result of *network activity*, this is highly appropriate as the generation of network packets is a *completely software dependant task* thus capturing network packets with *tcpdump* from the point-of-view of a security researcher is *identical* to a side-channel attacker capturing network packets with a *wiretap*. For side-channels involving *unintended hardware interactions*, *SIL* testing *may* not be as effective as required.

To solve this issue, for *future work*, *hardware-in-the-loop (HIL)* testing could be employed thus gathering more accurate side-channel data at the cost of less *deterministic* tests and greater *financial cost* of equipment [55]. To compromise, *HIL* solutions could be employed

to generate software models of side-channels detected in *real world* hardware solutions thus creating more accurate *SIL* tests. Furthermore, these generated software models could be shared among researchers thus allowing them to greater explore potential side-channel information leaks in their applications at a much lower costs. Not only could the *sharing* of statistical information be useful for more accurate *hardware simulation*, for *future work*, a library of *datasets* for applications, their *internal* behaviours, and their associated generated *side-channel cues* could be created. This library would assist both *system integrators* and *end users* with the verification of their specific security requirements. For example, a *dataset* could be released describing network traffic patterns generated for various interactions with a web application. A *user* or *system integrator* could then use this dataset to verify that certain *private* web application interactions could not be detected if they occurred on a cryptographically protected *private* network such as a VPN or Tor [56]. To summarize, for *future work*, the inclusion of more *data mining* could extend the *effectiveness* of the proposed framework.

In conclusion, this thesis has presented the *design*, *implementation* and *evaluation* of a framework for checking software systems for side-channel information leaks. Through the evaluation of the implemented framework, it has been shown that common software configurations *assumed* to be *secure* in reality can leak *private* information through side-channels if improperly used. Through the *collection* and *sharing* of more data on software and hardware performance, the framework could be further improved, in *future work*, to support the detection of more types of *side-channels*.

Bibliography

[1] M. Stampar, "sqlmap - under the hood," in *PHDays 2013*, Moscow, May 2013. [Online]. Available: https://www.slideshare.net/stamparm/ph-days-2013miroslavstamparsqlmapunderthehood.

[2] Y. Zhou and D. Feng, "Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing." *IACR Cryptology ePrint Archive*, vol. 2005, p. 388, 2005. [Online]. Available: http://dblp.uni-trier.de/db/journals/iacr/iacr2005.html#ZhouF05.

[3] R. Amirtharajan and J. B. B. Rayappan, "Steganography-time to time: A review," *Research Journal of Information Technology*, vol. 5, pp. 53–66, 02 2013.

[4] M. Lescisin and Q. Mahmoud, "Tools for active and passive network side-channel detection for web applications," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/lescisin.

[5] A. Holmes, S. Desai, and A. Nahapetian, "Luxleak: Capturing computing activity using smart device ambient light sensors," in *Proceedings of the 2Nd Workshop on Experiences in the Design and Implementation of Smart Objects*, ser. SmartObjects '16. New York, NY, USA: ACM, 2016, pp. 47–52. [Online]. Available: http://doi.acm.org/10.1145/2980147.2980150.

[6] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461.

[7] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 388–397. [Online]. Available: http://dl.acm.org/citation.cfm?id=646764.703989.

[8] T. W. Kim, T. H. Kim, and S. Hong, "Breaking korea tansit card with side-channel analysis attack - unauthorized recharging," in *18th Annual BlackHat Asia Conference*, Singapore, 2017. [Online]. Available: https://pdfs.semanticscholar.org/a353/7e75e46a82bcfe222edd0a3ec9b0f0758ab0.pdf.

[9] N. Apthorpe, D. Reisman, S. Sundaresan, A. Narayanan, and N. Feamster, "Spying on the smart home: Privacy attacks and defenses on encrypted iot traffic," *CoRR*, vol. abs/1708.05044, 2017. [Online]. Available: http://arxiv.org/abs/1708.05044.

[10] National Security Agency, "Tempest: A signal problem," September 2007.

[11] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 465–488, Firstquarter 2018.

[12] M. Li, Y. Meng, J. Liu, H. Zhu, X. Liang, Y. Liu, and N. Ruan, "When csi meets public wifi: Inferring your mobile phone password via wifi signals," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1068–1079. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978397.

[13] D. Genkin, M. Pattani, R. Schuster, and E. Tromer, "Synesthesia: Detecting screen content via remote acoustic side channels," *CoRR*, vol. abs/1809.02629, 2018. [Online]. Available: http://arxiv.org/abs/1809.02629.

[14] Symantec Corporation, "A technical review of caching technologies," 2017. [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/white-papers/caching-technologies-en.pdf.

[15] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[16] T. Hornby, "Side-channel attacks on everyday applications: Distinguishing inputs with flush+reload," in *19th Annual BlackHat USA Conference*, Las Vegas, 2016. [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Hornby-Side-Channel-Attacks-On-Everyday-Applications-wp.pdf.

[17] G. E. Blelloch, "Introduction to data compression," 2013. [Online]. Available: https://www.cs.cmu.edu/~guyb/realworld/compression.pdf.

[18] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009, pp. 248–255.

[19] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen, "Sidebuster: Automated detection and quantification of side-channel leaks in web application development," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 595–606. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866374.

[20] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 263–274. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046737.

[21] C. Sung, B. Paulsen, and C. Wang, "Canal: A cache timing analysis framework via llvm transformation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 904–907. [Online]. Available: http://doi.acm.org/10.1145/3238147.3240485.

[22] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[23] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan, "Orange: Data mining toolbox in python," *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013. [Online]. Available: http://jmlr.org/papers/v14/demsar13a.html.

[24] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278.

[25] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson, "I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP

'11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 147–161. [Online]. Available: https://doi.org/10.1109/SP.2011.23.

[26] F. Picariello, S. Rapuano, and U. Villano, "Evaluation of power consumption of workstation computers using benchmarking," in *Proceedings of the 12th IMEKO TC10 Workshop on Technical Diagnostics*, Florence, Italy, 2013. [Online]. Available: https://pdfs.semanticscholar.org/e10d/12d52e7ca6e90bbd56ec1fdd96cd3eca0003.pdf.

[27] T. Richardson and J. Levine, "The Remote Framebuffer Protocol," Internet Reqests for Comments, RFC Editor, RFC 6143, March 2011. [Online]. Available: https://tools.ietf.org/pdf/rfc6143.pdf.

[28] OWASP, "Top 10-2017 a3-sensitive data exposure," Jan. 2018. [Online]. Available: https://www.owasp.org/index.php/Top_10-2017_A3-Sensitive_Data_Exposure.

[29] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on ssh," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. Berkeley, CA, USA: USENIX Association, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251327.1251352.

[30] M. F. Triola, "Bayes' theorem," in *Elementary Statistics 11th Edition*. Addison-Wesley, Jan. 2009.

[31] D. Niyonkuru, "Incident management and chatops at shopify," in *Proceedings of the 2017 Site Reliability Engineering Conference (SREcon17 Europe)*. Dublin: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/srecon17europe/program/presentation/niyonkuru.

[32] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[33] J. Schipp, J. Dopheide, and A. Slagell, "Islet: An isolated, scalable, & lightweight environment for training," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, ser. XSEDE '15. New York, NY, USA: ACM, 2015, pp. 17:1–17:6. [Online]. Available: http://doi.acm.org/10.1145/2792745.2792762.

[34] Red Hat, Inc., "Ansible in depth," 2017. [Online]. Available: https://www.ansible.com/hubfs/pdfs/Ansible-InDepth-WhitePaper.pdf.

[35] N. Naik, "Migrating from virtualization to dockerization in the cloud: Simulation and evaluation of distributed systems," in *2016 IEEE 10th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA)*, Oct 2016, pp. 1–8.

[36] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472: O'Reilly Media, Inc., 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp.

[37] P. Biondi, "Packet generation and network based attacks with scapy," in *CanSecWest 2005*, 2005. [Online]. Available: http://www.secdev.org/conf/scapy_csw05.pdf.

[38] K. P. Lawton, "Bochs: A portable pc emulator for unix/x," *Linux J.*, vol. 1996, no. 29es, Sep. 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=326350.326357.

[39] T. Hornby, "Bochs hacking guide," 2014. [Online]. Available: https://defuse.ca/bochs-hacking-guide.htm.

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2078195.

[41] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science Engineering*, vol. 13, no. 2, pp. 22–30, March 2011.

[42] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[43] G. Goller and G. Sigl, "Side channel attacks on smartphones and embedded devices using standard radio equipment," in *Revised Selected Papers of the 6th International Workshop on Constructive Side-Channel Analysis and Secure Design - Volume 9064*, ser. COSADE 2015. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 255–270. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21476-4_17.

[44] C. Truong, L. Oudre, and N. Vayatis, "ruptures: change point detection in Python," *ArXiv e-prints*, Jan. 2018.

[45] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[46] Wikipedia Authors, "Wikipedia main page," Sep. 2018. [Online]. Available: https://web.archive.org/web/20180917052759/https://en.wikipedia.org/wiki/Main_Page.

[47] S. Colley and D. Chell, "Practical attacks against encrypted voip communications," in *HITBSECCONF2013*, Malaysia, 2013. [Online]. Available: https://www.mdsec.co.uk/2013/10/practical-attacks-against-encrypted-voip-communications/.

[48] F. Myrstad, "How tech companies deceive you into giving up your data and privacy," Sep. 2018. [Online]. Available: https://www.ted.com/talks/finn_myrstad_how_tech_companies_deceive_you_into_giving_up_your_data_and_privacy?language=en.

[49] B. Greene, "How tech companies deceive you into giving up your data and privacy (subtitles)," Oct. 2018. [Online]. Available: https://amara.org/en/videos/egZvro5sSZt5/en/2300045/.

[50] D. Mayer and J. Sandin, "Time trial: Racing towards practical remote timing attacks," in *Blackhat USA 2014*. Las Vegas, NV: Matasano Security Research, 2014. [Online]. Available: https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/TimeTrial.pdf.

[51] A. Tannous, J. Trostle, M. Hassan, S. E. McLaughlin, and T. Jaeger, "New side channels targeted at passwords," in *2008 Annual Computer Security Applications Conference (ACSAC)*, Dec 2008, pp. 45–54.

[52] P. Vila and B. Kopf, "Loophole: Timing attacks on shared event loops in chrome," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 849–864. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/vila.

[53] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 92–101. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486801.

[54] P. Barry and P. Crowley, *Modern Embedded Computing: Designing Connected, Pervasive, Media-Rich Systems*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[55] C. O'Flynn and Z. D. Chen, "Chipwhisperer: An opensource platform for hardware embedded security research," in *In: Constructive Side-Channel Analysis And Secure Design - COSADE*, 2015.

[56] M. Lescisin and Q. Mahmoud, "Dataset for web traffic security analysis," in *44th Annual Conference of the IEEE Industrial Electronics Society*. Washington, DC: IEEE Industrial Electronics Society, 2018.