University of California
Santa Barbara

Detection, Quantification, and Mitigation of Network Side Channels

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

İsmet Burak Kadron

Committee in charge:

Professor Tevfik Bultan, Chair
Professor Giovanni Vigna
Professor Yu-Xiang Wang

September 2022

The Dissertation of İsmet Burak Kadron is approved.

Professor Giovanni Vigna

Professor Yu-Xiang Wang

Professor Tevfik Bultan, Committee Chair

September 2022

Detection, Quantification, and Mitigation of Network Side Channels

Acknowledgements

I would like to express my deepest thanks to my Ph.D. advisor, Tevfik Bultan, whose encouragement and mentorship have been invaluable to me. Throughout my years in the Verification Lab at UCSB, Tevfik has been exceptionally supportive and generous with his advice.

I am thankful to my committee members, Giovanni Vigna and Yu-Xiang Wang, for providing me with feedback, support and encouragement as I reached milestones in my Ph.D.

My research could not have happened without collaborations with great, talented researchers. I am especially grateful to have closely collaborated with Nicolàs Rosner and Chaofan Shou at UCSB. I am also very grateful to my collaborators outside UCSB. Collaborating with Corina Păsăreanu and Divya Gopinath at CMU and NASA Ames was especially helpful and formative to me.

I will miss my time spent in the Verification Lab and I am thankful to the current and former members, Nicolàs, Nestan, Miroslav, Isaac, Baki, Lucas, Will, Seemanta, Yilmaz, Mara, Albert, Chaofan, Emily, Laboni, Shafiuzzaman.

I am grateful to be a member of the UCSB Computer Science and dancing communities. Organizing coffee hours, barbecues and meeting so many people has been a great experience. I will cherish my chats with Karen and running cooking club events with Samantha and Genevieve was very helpful to keep my sanity during the pandemic.

Finally, I would like to thank my parents, Emine and Bahadır, and my brother Emre for their unending support. In addition, I would like to thank all my friends for their friendship and support over the course of my journey: Mehmet, Metehan, Ahmet, Tarık, Emre, Utku, Mert, Zeki, Alex, Megan, and Hawkins. I am especially grateful for Mehmet Emre's friendship over the course of 10+ years from my Boğaziçi days.

Curriculum Vitæ

İsmet Burak Kadron

Education

2022 (Expected)	Ph.D. in Computer Science, University of California, Santa Barbara, CA, USA.
2014	B.Sc. in Computer Engineering, Boğaziçi University, Istanbul, Turkey.

Publications

İsmet Burak Kadron, Chaofan Shou, Emily O'Mahony, Yılmaz Vural, Tevfik Bultan, *Targeted Black-Box Side-Channel Mitigation for IoT.* Submitted.

İsmet Burak Kadron, Yannic Noller, Rohan Padhye, Tevfik Bultan, Corina S. Păsăreanu, Koushik Sen, *HUGS: Human-Guided Software Testing.* Submitted.

Mara Downing, William Eiers, Brian Ozawa Burns, Erin DeLong, Anushka Lodha, **İsmet Burak Kadron**, Tevfik Bultan, *Symbolic Quantitative Verification of Quantized Neural Networks.* Submitted.

İsmet Burak Kadron, Tevfik Bultan, *TSA: A Tool to Detect and Quantify Network Side-Channels.* Tool Demonstrations Track of the 30th ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE 2022).

İsmet Burak Kadron, Divya Gopinath, Corina S. Păsăreanu, Huafeng Yu, *Case Study: Analysis of Autonomous Center line Tracking Neural Networks.* In the proceedings of 13th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2021).

İsmet Burak Kadron, Nicolàs Rosner, Tevfik Bultan, *Feedback-Driven Side-Channel Analysis for Networked Applications.* In the proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020).

Seemanta Saha, William Eiers, **İsmet Burak Kadron**, Tevfik Bultan, *Incremental Attack Synthesis.* In ACM SIGSOFT Software Engineering Notes 44(4): 16 (2019). Proceedings of the Java Pathfinder Workshop 2019 (JPF 2019).

Divya Gopinath, Mengshi Zhang, Kaiyuan Wang, **İsmet Burak Kadron**, Corina S. Păsăreanu, Sarfraz Khurshid, *Symbolic Execution for Importance Analysis and Adversarial Generation in Neural Networks.* In the proceedings of 30th International Symposium on Software Reliability Engineering (ISSRE 2019).

Nicolàs Rosner, **İsmet Burak Kadron**, Lucas Bang, Tevfik Bultan, *Profit: Detecting and Quantifying Side Channels in Networked Applications.* In the proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS 2019).

Seemanta Saha, **İsmet Burak Kadron**, William Eiers, Lucas Bang, Tevfik Bultan, *Attack Synthesis for Strings using Meta-Heuristics.* In ACM SIGSOFT Software Engineering Notes 43(4): 56 (2018). Proceedings of the Java Pathfinder Workshop (JPF 2018).

Abstract

Detection, Quantification, and Mitigation of Network Side Channels

by

İsmet Burak Kadron

Modern software systems such as web clients and Internet of Things (IoT) devices regularly access and transmit private and sensitive data such as location information or user actions. Although these systems use secure and encrypted communications to transmit this information, the information can be recovered by observing the side effects of the communication such as packet sizes, timings and source and destination information which is public to any eavesdropper. These side effects can be obfuscated by delaying packet timings, padding packet contents or injecting dummy packets. These obfuscations also impact the quality of transmission, therefore a balance between user privacy and network overhead is needed.

In this dissertation, we provide methods for (1) detecting and quantifying network side-channel information leakages, (2) input generation for automating analysis of network side-channels, (3) automating side-channel mitigation with user constraints. Firstly, we present how the network side-channels can be detected and quantified by identifying relevant features with trace analysis. Our approach quantifies the information leakage of each feature using Shannon entropy and probability estimation methods, and provides a ranking of features based on the amount of leakage. Secondly, we discuss how the detection and quantification can be improved by dynamically generating inputs based on user provided mutators and seed inputs. Our method determines which features are affecting the information leakage and picks the mutators that influence those features or the secret. Thirdly, we present approaches on measuring the amount of information leakage using upper and lower bounds on the estimates. We also present techniques that synthesize side-channel attacks using classifiers and provide upper bounds on classifier accuracy.

Then, we present a search-based method to generate mitigation strategies to information leakages based on user constraints to balance network overhead and leakage amount. Our approach iterates over the top ranking features and tries to reduce the information leakage of each feature by searching a padding and delaying strategy that minimizes an objective function based on the amount of leakage and network overhead, stopping when no further improvement is found. Lastly, we present the tool we developed which unifies the described approaches in a single workflow. For all approaches, we demonstrate their effectiveness on a set of experimental benchmarks.

Contents

List of Figures

List of Tables

List of Procedures

Chapter 1

Introduction

Our world's professional, commercial, governmental, and personal activities are quickly migrating to networked software systems. Standalone systems are an artifact of the past: most modern applications are network enabled. As computer systems become increasingly interconnected and more information circulates over the network, information leaks with tangible and disturbing consequences on our daily lives have become a recurrent section in the news. Effects of information leakage from computer systems range from influencing the US presidential elections (e.g., the Clinton and Podesta emails published by Wikileaks [1] in 2016) to revealing personal and financial information of billions of people (e.g., the Equifax [2] and Yahoo! [3] leaks announced in 2016 and 2017). Especially in consumer applications, networked systems such as web services, mobile applications, and Internet of Things devices allow users to perform their personal banking operations, control lights and locks in their house, and many other things. While these services are helpful to users, they also carry the risk of being vulnerable to malicious actors. Any information leakage, such as location of the user, their application usage information, such as when they turn on their lights or coffee machine, compromises the privacy of the user. Hence, detection of information leaks in networked applications has become a significant and urgent problem in software development.

In side-channel information leakages, private information can be extracted by a passive attacker who observes and analyzes visible side effects of the computation. This type of information leakages are becoming more common as the side effects such as power usage, network usage and execution time are optimized to suit different situations. Well-known side channel attacks include those based on power consumption [4], electromagnetic radiation [5], cache timing [6], and CPU-level branch prediction and race conditions, such as Spectre [7] and Meltdown [8] attacks.

To mitigate information leakage on network traffic, most top-100 online services are now using SSL/TLS encryption, and its adoption by smaller websites and services is growing at a fast pace [9]. This is a positive step toward avoiding trivial leaks, however encryption can also provide a false sense of security. This kind of encryption only hides the content of TCP/IP packet payloads. There is still a plethora of visible metadata (such as packet size, timing, direction, flags, etc.) that can be obtained from message headers and searched for patterns that may be exploitable as side channels. Side-channel analysis of encrypted network traffic has been used, for example, to gain knowledge about user keystrokes during SSH connections [10], to identify medical conditions of a patient from the encrypted traffic generated by a healthcare website [11], to identify which app, among a known set of fingerprinted apps, is being used by a mobile phone user [12], and to learn about sleeping habits of users by monitoring IoT sensor traffic [13].

Detecting these types of information leakages require novel analysis tools and methods. There has been prior work on white-box side-channel analysis techniques [14–17] but they are difficult to apply to network applications. White-box tools require access to source code, which is typically not public for mobile applications or commercial IoT devices. Moreover, these applications do not lend themselves to white-box analysis due to their multi-component nature, consisting of client and server components, which could all be written using different languages and frameworks. Black-box methods like network traffic analysis tools are more suitable for these applications because they do not require source code access, and they can be applied to heterogeneous, multi-component systems. Black-box techniques only require access to the system under test in order to run the system multiple times to gather data and analyze the system's behavior using the data gathered by profiling [18–20].

This dissertation focuses on the problem of detection, quantification and mitigation of network side-channels. This dissertation is concerned with understanding;

- How do side channels arise in network traffic?
- How to measure the amount of information leaked via network side channels?
- How to automate the network side-channel analysis problem?
- How to mitigate network side channels?

Side-channel Analysis and Quantification. To address the first point of how the side channels arise, in Chapter 2, we present a black-box side-channel analysis technique, Profit, which detects and quantifies side-channels in networked applications [18]. Profit takes a profiling-input suite and runs the system with the provided inputs to produce a set of network packet traces of varying length. These packet traces give rise to an enormous feature space that includes sizes and timing of all packets and their aggregations. In order to identify the features that leak information and to quantify the amount of information leaked, Profit uses techniques such as multiple sequence alignment developed for bio-informatics to align packets of network traces and extracts phases where the packet sequences are similar. This alignment helps us identify features that leak information in smaller subtraces. To address the second point of measuring the side-channels, Profit uses information-theoretic metrics, such as Shannon entropy, to quantify the amount of information leaked by network observations, based on the features identified during phase detection and feature selection, and the automatically inferred probability distributions for features. Profit presents the user with a ranking of the features that are the most worth looking into, sorted by the amount of information leakage.

Automating Analysis with Mutation Based Input Generation. One difficulty in using Profit is profiling. The users need to profile the system with different inputs extensively and repeatedly to cover all possible behaviors of the system, capture the encrypted network traces and label the data with the information leakage source the user wants to check. An example of this is a smart light bulb device in which the user would want to check if the certain user actions (e.g. turning the light on or off) are leaking over the network traces. To find the leakage, the user would need to turn on and off the lightbulb at different luminosities, different times if they want an extensive input suite.

To address the problem of automating the analysis, in Chapter 3, we present an approach for side-channel analysis using a mutation based input generation, called AutoFeed [21]. Instead of using a large set of user-provided inputs (which requires either manually writing the inputs or writing a generator for random inputs), AutoFeed uses a small set of seed inputs and mutators provided by the user. These mutators can be used to generate a larger input set. To explore a variety of behaviors that correlate with the secret information, we need to choose mutators that modify the fields that are related to the secret. We perform this by testing each mutator on the seed set and adjusting the weights of the mutators based on its ability to provide the inputs that generate different traces or secret values compared to the original inputs.

Upper and Lower Bound Estimates for Leakage. Measuring the amount of information leakage accurately is important in determining the impact of the network side-channel on the user privacy. Measuring the information leakage for individual features may not reveal the total amount of information leakage obtained by all features. There may be cases where individual features do not fully correlate with the secret information but combined features do. One problem is that increasing the number of features increases the time complexity of the leakage computation exponentially. Therefore, we need to use approximate methods that can compute the information leakage in a timely manner. To do so, we explore methods that give bounds on the amount of information leakage using neural networks and other statistical methods. In Chapter 4, we describe techniques for obtaining bounds on information leakage for network side channels.

Side-Channel Leakage Mitigation. As we mentioned before, network side-channels hurt user's privacy with the vulnerabilities in smart home systems, smartphone applications, websites because of the network traffic metadata such as packet sizes, timings. Prevention of these information leakages is important and these information leakages can be prevented by obfuscating the correlation between the traffic metadata and the secret information. This requires modifying the traffic by padding packet contents to change their sizes, delaying packets to change their timings and injecting dummy packets which can change overall timing and confuse attackers. One critical problem is that balancing these modifications impact the quality of service of the network applications. Delaying packets or increasing bandwidth for a video streaming service may disrupt the quality of streaming and cause the application to lose users. Therefore, a mitigation method should achieve a balance between the user's privacy and quality of service.

To address the problem of mitigating side channel information leakages, in Chapter 5, we present Shark!, a technique for synthesizing mitigation strategies. Shark! works by collecting encrypted network traces and labeling them with the secret value of the corresponding trace (which can be user actions or a device state such as a motion sensor's status at the time of the trace capture). Shark! analyzes the traces by extracting features such as packet sizes and timings, and quantifies the information leakage. Using the information leakage quantification, Shark! prioritizes which features to target and iteratively develop a mitigation strategy with respect to a tunable objective function balancing the information leakage reduction and overhead. The tunable objective function can be customized by the user by changing the parameter that controls the trade-off between information leakage and mitigation overhead. This enables Shark! to synthesize a set of Pareto optimal [22] mitigation strategies corresponding to different trade-offs between privacy and performance.

1.1 Contributions

This thesis contributes the following:

- A black-box dynamic network analysis technique to detect and quantify side channels on encrypted network traffic.
- A mutation-based input generation technique to improve automation in testing and analysis of networked applications.
- A mitigation strategy synthesis method that finds the best strategy based on privacy and quality of service constraints.
- A tool which combines the aforementioned methods into a single workflow that researchers and developers can use.

These contributions address the problem of detection, quantification and mitigation of network side-channel information leakages and understanding how side channels arise in network traffic, how to measure the amount of information leaked, how to automate this problem, and how can side channels be obfuscated or mitigated.

1.2 Dissertation Outline

The rest of the dissertation is structured as follows.

In Chapter 2, we discuss the work on network side-channel detection and quantification, Profit. we cover the details on the system model, feature extraction using trace alignment, various quantification methods and go over the experimental evaluation using DARPA STAC benchmarks.

In Chapter 3, we cover the work on test input generation for network side channels, AutoFeed, explain how input generation works to achieve a feedback driven side-channel analysis framework, go over experimental results and implementation details on DARPA STAC benchmarks.

In Chapter 4, we cover the work on finding bounds on information leakages using statistical methods, describe how the information leakage attack can be demonstrated using classifiers and describe our experimental results on IoT benchmarks.

In Chapter 5, we cover the work on finding a mitigation strategy that balances the trade-off between information leakage and network overhead. We describe the search-based approach that develops packet padding, delaying and injection strategies targeting the most leaking features and synthesizes the mitigation strategy. We go over the experimental evaluation results on multiple IoT benchmarks.

In Chapter 6, we describe the tool for detecting and quantifying side-channel information leakages, TSA (**T**ool for **S**ide-channel **A**nalysis) which unifies the previous approaches. We describe the architecture and usage of TSA, and give case studies demonstrating the usefulness of the tool.

In Chapter 7, we summarize the prior works in the fields of software side-channel analysis, input generation and side-channel mitigation.

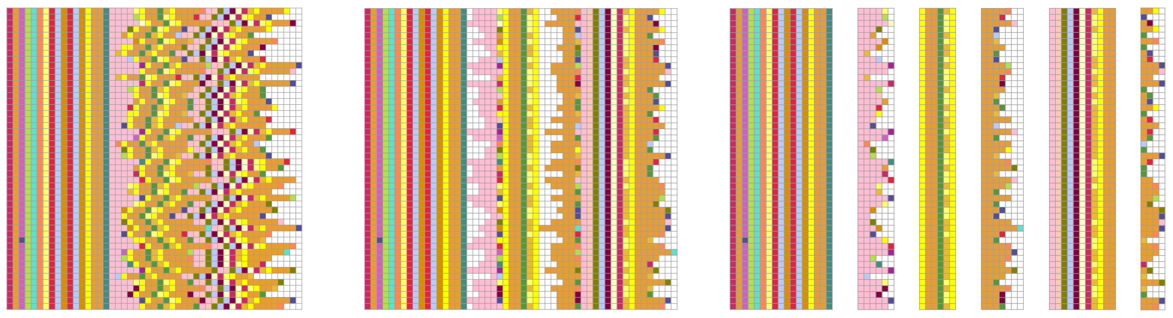
In Chapter 8, we present our final remarks, cover the possible future directions and conclude the dissertation.

Chapter 2

Detection and Quantification of Network Side Channels

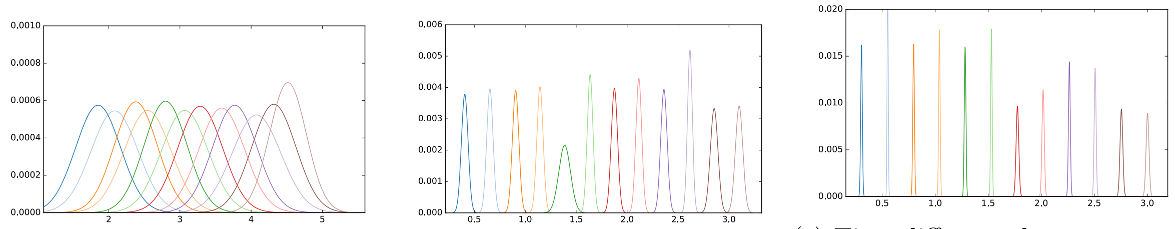
Identification of network side channels for a software system requires searching for correlations between sensitive information that the system accesses (i.e., secret inputs to the software system) and the outputs of the system that are observable over the network. Network packet traces are a complex form of output. Each trace contains a large, variable number of observables, resulting in an intractable number of potential features to investigate for information leakage. Selecting the features to analyze, and quantifying the amount of information leaked from different features are challenging problems. In this chapter, we present a tool called Profit that, given a software application and a profiling input suite, determines whether and how much information leakage occurs through a network side-channel for a particular secret of interest [18].

Given a profiling-input suite, Profit uses black-box network profiling to produce a set of network packet traces of varying length. These packet traces give rise to an enormous feature space that includes sizes and timing of all packets and their aggregations. In order to identify the features that leak information and to quantify the amount of information leaked, Profit uses a three-step approach:



(a) Traces as captured. (b) After alignment. (c) After splitting into phases.

Figure 2.1: Trace alignment and phase detection (50 traces shown) for GabFeed. Colors represent different packet sizes.



(a) Total duration of the whole trace. (b) Total duration of the fifth phase. (c) Time difference between the third and fourth packets within the fifth phase.

Figure 2.2: Probability densities for different feature values for GabFeed. Colors represent different secrets. X-axis is time (sec).

1. *Alignment and Phase Detection:* We apply techniques developed for gene alignment to align packets of network traces. Then, by studying the variations across traces, we identify phases of application behavior as either constant-length patterns or variable-length exchanges between such patterns.

2. *Feature Extraction and Selection:* We define a feature space that covers observations about individual packets as well as observations about sequences of packets (such as the time difference between two packets, or the total size over all packets). Trace alignment and phase detection enable us to identify new features that we would not consider otherwise, and to do so in a modular way, by focusing on features of each phase separately. We also extract features from the unmodified, full-length original traces.

3. *Information Leakage Quantification:* We use information-theoretic metrics, such as Shannon entropy, to quantify the amount of information leaked by network observations, based on the features identified during phase detection and feature selection, and the automatically inferred probability distributions for features. We present the user with a ranking of the features that are most worth looking into, sorted by the amount of information leakage.

When combined, these three steps provide a black-box approach for detecting information leakage from applications that is due to network communication. We experimentally evaluate Profit using benchmark applications from the DARPA Space/Time Analysis for Cybersecurity (STAC) program [23]. The STAC benchmark is developed by DARPA to evaluate the effectiveness of side-channel detection techniques, and consists of a variety of realistic networked applications that often contain side-channel vulnerabilities. Our experiments with the DARPA STAC benchmark show that Profit is able to automatically identify features associated with the side-channel vulnerabilities in these applications and quantify the amount of information leaked by each feature, providing crucial insight about the existence and severity of side-channel vulnerabilities.

The rest of the chapter is organized as follows. In Section 2.1 we give an overview of our approach. In Section 2.2 we discuss the system model we use. In Section 2.3 we present the trace alignment and phase detection techniques. In Section 2.4 we present the feature extraction and leakage quantification techniques. In Section 2.5 we discuss the experimental evaluation of our approach using the DARPA STAC benchmark. In Section 2.6 we discuss the limitations of our approach. In Section 2.7, we provide a summary of the chapter.

2.1 Motivation and Overview

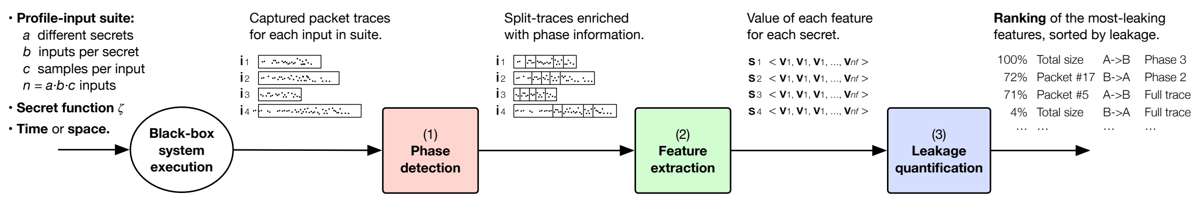

• Profile-input suite:
a different secrets
b inputs per secret
c samples per input
n = a·b·c inputs
• Secret function ζ
• Time or space.
Black-box system execution
Captured packet traces for each input in suite.
(1) Phase detection
Split-traces enriched with phase information.
(2) Feature extraction
Value of each feature for each secret.
(3) Leakage quantification
Ranking of the most-leaking features, sorted by leakage.
100% Total size A->B Phase 3
72% Packet #17 B->A Phase 2
71% Packet #5 A->B Full trace
4% Total size B->A Full trace

Figure 2.3: Profit workflow.

Let us consider GabFeed, an application from the DARPA STAC benchmark, as a motivating example. GabFeed is a Web-based forum. Users can post messages, search the posted messages, consult a database on special issues, and engage in direct chat. Besides the typical user login form, it also offers a challenge-response mechanism for the user to confirm the server's identity. This mechanism has a timing side channel—the small delay between two of the network packets spawned by this action is proportional to the number of 1s in the binary representation of the server's private key.

Consider the following interaction: A user performs a search, then checks the identity of the server before performing another more sensitive search. This is repeated many times, for profiling purposes, using different server private keys with varying numbers of 1s. Suppose that we captured the network traffic for many different inputs and painstakingly inspected it using WireShark [24]. Even if we knew where to look, it would still take nontrivial amount of effort to confirm that those two specific packets indeed leak sensitive information. If we did *not* know where to look, attempting this manually would be a truly daunting endeavor.

An automated tool that can assist in such a search would need to examine a vast feature space—not just the size of each packet, its flags, its direction, but also all possible time differences (deltas) and sums over all possible subsets of packets. Since this is infeasible to do for network traces generated by realistic applications, a well-chosen feature space needs to be selected for consideration.

Figure 2.1a shows the network traffic captured by Profit for the GabFeed application for 50 repeated interactions, using many different server private keys with 12 different numbers of 1s in the key, and different search queries. Each row represents a complete interaction (a *trace*) as a sequence of packet sizes. Colors represent packet sizes. To keep variations visible, the palette is not a gradient.

In this example, the fact that both search operations introduce variable amounts of space and time utilization before and after the leaky event makes the crucial feature harder to characterize. In fact, even *naming* the feature in terms of the captured traffic is not trivial (e.g., it is not "the n-th packet" for any consistent value of n).

Figures 2.1b and 2.1c show the same 50 traces after being globally aligned and then separated into phases, respectively, by Profit. This process enables Profit to synthesize the crucial feature that successfully captures this side channel. Figure 2.2 shows the probability density functions (for each value of the secret, i.e., for each number of 1s in the private key) for a few of the numerous features that were considered by Profit during the leakage quantification step.

Without alignment or phase detection, the best (i.e., most-leaking) feature that Profit reported was the time difference between the first and the last packets in each whole trace—that is, the duration of each trace. Note that since there are 12 different values of the secret (number of 1s in the key), there is $\log_2 12 = 3.58$ bits of secret information. Profit quantified the leakage of this feature as roughly 40% of the secret information (1.44 of 3.58 bits). Figure 2.2a shows the probability density functions inferred by Profit. Each curve represents one possible value of the secret. Intuitively, the leakage of 40% is much less than 100% because of the significant overlap between the distributions, yet well above 0% because there is some degree of certainty (note that the first and last curves, for instance, are almost completely non-overlapping).

With phase knowledge, Profit can consider more refined features, like the time difference between the first and last packets *of the fifth phase*. Figure 2.2b shows the probability densities for this feature, for which Profit computed 99% leakage (3.56 of 3.58 bits). Note that the total duration of phase 5 also includes some noise from other packets in the phase, and this entails some minor overlap.

As it turns out, the top-ranking feature reported by Profit was the time difference *between packets #3 and #4 of the fifth phase.* As illustrated in Figure 2.2c, this even more specific feature has maximal separation between the distributions of each secret's probability given an observation. Not surprisingly, it is the only feature that yields 100% leakage (3.58 of 3.58 bits) for this example.

Figure 2.3 shows the three main steps of our approach implemented in our tool Profit. Given a profiling suite, we first generate network traces that correspond to each input value. We run each input value multiple times in order to capture variations due to noise. Next, we align network traces and divide them to phases after alignment. After alignment and phase detection, each network trace is divided to a fixed number of sub-traces where each sub-trace corresponds to a phase. Next, using our feature library we identify the set of features for each phase. We, then quantify the amount of information leaked via each feature in terms of entropy using the automatically inferred probability distributions for features. At the end, Profit produces a ranked list of features, sorted with respect to the amount of information leaked via that feature. Profit also reports the amount of information leaked via each feature in terms of number of bits using Shannon entropy.

2.2 System Model

We present a simple formalization of the system model and definitions for some of the concepts used in our approach.

Inputs and secrets

We target networked applications, such as client-server and peer-to-peer systems, that communicate through an encrypted network channel (typically TCP encrypted using SSL/TLS). Our target systems often require complex, structured inputs. For a particular system of choice, let the *input domain* $\mathbb{I}$ be the set of all valid inputs, and let $\zeta : \mathbb{I} \longrightarrow \mathbb{S}$ be a function which, given an input, projects the *secret*—a piece of confidential information that the user wants to make sure that the system does not leak. We will call ζ the *secret function* (i.e., the secret-projecting function), and $\mathbb{S}$ the *secret domain* (i.e., the domain of the secret).

Packets

A *packet* is an abstraction of a network packet. Real-world packets contain many details, including nested payloads and headers with many fields and options. We assume that payloads are encrypted, and attacks trying to break the encryption is outside the scope of this work. We limit our abstraction of packets to a core subset of metadata from the highest-level header that is particularly relevant for side-channel analysis: the size of the encrypted payload in bytes (p.size), the time at which the packet was captured (p.time), and the source and destination addresses (p.src, p.dst) of the packet. We represent each packet p as a tuple that consists of packet meta-data:

$$p = (p.\text{time}, p.\text{size}, p.\text{src}, p.\text{dst}, p.\text{sport}, p.\text{dport})$$

Traces

Running a certain input $i \in \mathbb{I}$ through the system while capturing network traffic yields a *trace*, which is a sequence of packets $t = \langle p_1, p_2, \ldots, p_{|t|} \rangle$. Let $\mathbb{T}$ be the set of all possible traces.

Input set and secret set

Generally, it is not be feasible to run all possible inputs exhaustively through the system. Therefore, the user typically needs to select a subset of $\mathbb{I}$ that she wants to profile, i.e., a profiling input suite. Let the *input set* $\mathbf{I} \subseteq \mathbb{I}$ denote this set of distinct inputs that will be fed into the system during the analysis.

As explained above, each input $i \in \mathbf{I}$ has an associated secret value $\zeta(i)$. By choosing a set of inputs, the user is also choosing a set of secrets. Let the *secret set* (i.e., the set of secrets) $\mathbf{S} \subseteq \mathbb{S}$ be the set of distinct secrets fed into the system during the analysis.

Input list, secret list, captured trace list

Due to system nondeterminism (e.g., network noise, randomized padding), two runs with the same input $i \in \mathbf{I}$ may yield different traces. The user may find it desirable to run each input multiple times. We thus introduce *input lists*, which may include multiple appearances of each input.

When conducting a Profit analysis, the user generates a list of n inputs $\langle\, i_1, i_2, \ldots, i_n \rangle$, which implies a list of n secrets $\langle\, s_{(1)}, s_{(2)}, \ldots, s_{(n)} \rangle$ that can be obtained via $\zeta(i_j)$. Running all the inputs through the system while capturing network traffic yields a list of traces $\langle\, t_{(1)}, t_{(2)}, \ldots, t_{(n)} \rangle$. We will call these lists the *input list*, the *secret list*, and the *captured trace list*, respectively.

Features

A *feature* is a function $f : \mathbb{T} \longrightarrow \mathbb{R}$ that projects some measurable aspect of a network trace. Some examples of possible features are: the size of the first packet in the trace, the time of the last packet in the trace, the maximum of all sizes of odd-numbered packets in the trace, etc. There is an infinite number of possible features, ranging from very simple to arbitrarily complex ones.

Profiles

By running a Profit analysis, a *profile* of the system is obtained, which maps each feature name to the profile for that feature. The profile of the system for a feature f is the list of $(\zeta(i_j), f(t_j))$ tuples for $j \in [1 \ldots n]$, i.e., $\langle\, (s_1, f(t_1)), (s_2, f(t_2)), \ldots, (s_n, f(t_n)) \rangle$. In other words, the system profile for a feature f associates the secret value of each trace with the value of f for that trace. To summarize the results of the profiling, we can represent the trace list obtained from profiling as

$$T = (t_{(1)}, t_{(2)}, ..., t_{(|T|)})$$

and the corresponding secrets as a vector

$$y = (y_{(1)}, y_{(2)}, ..., y_{(|T|)}).$$

Direction-induced subtraces

If $t \in \mathbb{T}$ is a trace, let $t^{\uparrow}$ and $t^{\downarrow}$ be the traces induced by keeping only the packets from t whose attributes p.src and p.dst are consistent with the specified direction, respectively. For instance, suppose $t = \langle\, p_1, p_2, p_3, p_4 \rangle$ where p_1 and p_4 were sent from client to server, and p_2 and p_3 were sent from server to client. Then $t^{\uparrow} = \langle\, p_1, p_4 \rangle$ and $t^{\downarrow} = \langle\, p_2, p_3 \rangle$. For peer-to-peer systems, the left side denotes a designated peer that runs on the client machine, and the right side denotes all other peers.

Split traces

A *trace-splitting function* ϕ is a function that, given a trace $t \in \mathbb{T}$, splits t into subtraces (which are themselves traces) whose concatenation is the original t. A *split trace* is a sequence of traces obtained by splitting a trace.

2.3 Alignment and Phase Detection

In this section we describe our heuristics for trace alignment and phase detection. The former leverages well-known tools from molecular biology, and the latter is based on the output of the former.

2.3.1 Trace alignment

Given a list of captured traces $\langle t_1, t_2, \ldots, t_n \rangle$, where each t_i may have a different length, we would like to detect *stable* patterns that appear in nearly identical form across nearly all of the t_i, and then use them to identify the *variable* parts in between which, despite varying significantly across traces, could be semantically related in a meaningful way. This is essentially *multiple sequence alignment* (MSA), a well-studied problem in computational biology [25] where sequences of nucleic acids need to be aligned in a similar fashion. Many crucial analyses in biology (e.g., determining the evolutionary history of a family of proteins) depend on MSA. However, obtaining an optimal alignment is an NP-hard problem [26, 27]. Many heuristic approaches exist, typically based on progressive methods [28] or iterative refinement [29, 30]. Some popular heuristic toolkits that yield a good compromise between accuracy and execution time are the CLUSTAL [31] family, T-COFFEE [32], and MAFFT [33]. They are often limited to strings over small alphabets, give each character a specific biological meaning, and rely heavily on precomputed tables for common character combinations from the biology domain. We use MAFFT, which offers a generic mode with a large alphabet and no special meaning for each character.

We align traces based on their packet size sequences, i.e., for $t = \langle p_1, p_2, \ldots, p_{|t|} \rangle$ we consider $\langle p_1.\text{size}, p_2.\text{size}, \ldots, p_{|t|}.\text{size} \rangle$. We also incorporate some information about packet direction into the sequence of sizes by encoding the direction of each packet into the sign of its size. Considering packet timestamps could also provide useful insight for alignment, but we found it difficult to leverage size and time information simultaneously in a consistent way. For the purpose of alignment, and for our benchmark, sequences of (directed) packet sizes proved to be a far more useful characterization than sequences of timestamps. Nevertheless, note that a size-based alignment can serve as a source of new features not just in space but also in time, as we saw in Section 2.1.

Recall Figure 2.1 from Section 2.1, which shows 50 traces captured from GabFeed before alignment, after alignment, and after phase splitting. Colors represent packet sizes. White represents the absence of a packet. To improve readability of slight differences in packet sizes, the palette is intentionally not a gradient.

The alignment tool yields a list of sequences of packet sizes, but each sequence may contain gaps (shown in white). Gaps are inserted so as to try and maximize the alignment of patterns that are recurrent across traces. As a consequence, stable patterns become aligned columns, and variable patterns can emerge which are visibly related across traces, but were hard to detect before alignment.

Figures 2.4 and 2.5 are analogous to Figures 2.1a and 2.1b, respectively, but show a much smaller set of shorter sequences.

2	5	8	8	9	-4	-3	1	1	1	
0	5	8	7	-4	-3	1	1			
2	5	8	8	8	6	-4	-3	1	1	1
2	3	8	8	4	-4	-6	1	1	1	1

Figure 2.4: An example of unaligned sequences of values.

2	5	8	8	9	–	-4	-3	1	1	1	–
0	5	8	7	–	–	-4	-3	1	1	–	–
2	5	8	8	8	6	-4	-3	1	1	1	–
2	3	8	8	4	–	-4	-6	1	1	1	1

Figure 2.5: An example of aligned sequences of values (with inserted gaps).

2.3.2 Phase detection

As exemplified by Figure 2.1 (b), thanks to the inserted gaps, the aligned sequences present a new horizontal axis that is better suited for splitting the traces into meaningful subtraces. This eases the detection of stable regions, which we will call *stable phases*, and as a consequence, of the variable regions that appear before, in between or after them, which we will call *variable phases*. Note that the word *phase* applies to both kinds of regions.

We now need a heuristic method to find stable phases and select cut-points along the horizontal axis of the matrix. Let M be an aligned matrix with n rows and m columns. Let C_j be the j-th column, and $\#G_j$ the number of gaps in it. The *density* of the j-th column is the ratio C_j/n, and its *diversity* is the variance of the $(n - \#G_j)$ values in C_j that are not gaps. We characterize stable regions using two thresholds: the *maximum diversity* (ψ) that a column may have in order to be part of a stable phase, and the *minimum width* (ω), in columns, that may constitute a stable phase.

Hence, a stable phase is a maximally wide run of adjacent columns that are fully dense and that satisfy both thresholds: (i) the run is at least ω columns wide, and (ii) each column within the run has at most ψ diversity. Using this characterization, we synthesize a regular pattern that can parse all sequences of values. The pattern is akin to a regular expression, but with arbitrary integer values instead of characters. For the simple example shown in Figure 2.5, assuming $\omega = 3$ and $\psi = 0.25$, the synthesized pattern would be

$$(\mathsf{int}*)((2|0)(5|3)(8)(8|7))(\mathsf{int}*)((-4)(-3|-6)(1)(1)(\mathsf{int}*))$$

where int stands for "any integer" and $*$ is the Kleene star.

The pattern demands that the stable parts be present, accounts for some amount of diversity in them, and allows for freedom before, after, and in between the stable parts. In general, assuming $k > 0$ stable regions are found, we build a pattern of the form

$$(V_1)(S_1)(V_2)(S_2)\dots(V_k)(S_k)(V_{k+1})$$

Each S_i represents a stable region. For the i-th stable region with length l, $S_i = d_1 d_2 \dots d_l$, where each of the d_j is either a constant integer (if position j within S_i always had the exact same value for all traces), or a union of integers $d_j = (x_1|x_2|\dots|x_r)$ if that position exhibited r different values within the allowed threshold. Each V_i represents a variable region and consists of a free pattern (any sequence of integers). All regions are named and then used to extract the corresponding groups. Thus, the synthesized expression indeed becomes a parser for sequences of directed packet sizes.

If the number of available captured traces is so large that the MSA tool would take too long to find an alignment, we can still apply the tool to a reasonably large random subset of the traces. We then detect phases as explained above, and use the synthesized expression to parse the rest of the traces. Some traces could fail to parse if their stable parts include extraneous values that were not present in any of the aligned traces. If the traces that fail to parse are less than 1% of the total number of traces, we consider them outliers and ignore them. If they exceed 1%, we add them to the initial subset and realign. For all the examples in our benchmark, using a subset of at least 500 traces, we have never encountered a case where more than 1% of the traces have failed to parse.

2.4 Leakage Quantification

Once the traces have been separated into phases, we employ a set of feature extraction functions. For any particular feature, we use Shannon entropy to estimate the amount of information an attacker can gain by making side-channel observations about that feature in network traces. We find the features that leak information about a benign user's secret values, and rank them to identify the most informative trace features.

2.4.1 Feature extraction

Feature extraction is commonly used in leakage quantification and machine learning to extract information from data [12]. In our approach, we process network traces and subtraces obtained through phase alignment to extract features of interest. Each trace t is converted to three subtraces $t^{\uparrow}$, $t^{\downarrow}$ and $t^{\updownarrow}$ according to the direction of packets as explained in Section 2.2. We define a feature set $\mathbb{F} = \{f^1, f^2, \ldots, f^n\}$ such that each feature function f^j extracts a statistic from a packet or all packets from a subtrace.

We define aggregate features, which compute the sum of packet sizes and sum of timing differences between the first and last packets in a subtrace. We define fine-grained packet-level features, consisting of the size of packet p_i and timing differences between consecutive packets p_i and p_{i+1}. To ensure that we have same number of features for each sampled trace, we align each subtrace left and remove packets that are not fully aligned when computing per-packet features. Aggregate features are not affected by this change and use the entire subtrace. The features are summarized in Table 5.1. Applying a feature function f^j to each packet series obtained from traces results in a feature profile $P^i = \langle\, (s_1, v_1^j), (s_2, v_2^j), \ldots, (s_n, v_n^j) \rangle$, which is used to compute information leakage.

Table 2.1: Definition of network trace features.

Feature Function	**Definition**	**Description**
$f^{\text{sum-size}}(t)$	$\sum_{p \in t} p.\text{size}$	Sum of sizes of packets in sub-trace t.
$f^{\text{size}}(\langle\, p_1, \ldots, p_n \rangle, i)$	$p_i.\text{size}$	Size of packet i.
$f^{\text{total-time}}(\langle\, p_1, \ldots, p_n \rangle, i)$	$p_n.\text{time} - p_1.\text{time}$	Total time duration of packets in subtrace.
$f^{\Delta\text{time}}(\langle\, p_1, \ldots, p_n \rangle, i)$	$p_{i+1}.\text{time} - p_i.\text{time}$	Time difference between packets i and $i+1$.

2.4.2 Leakage Quantification

In our threat model, an attacker who is observing the network communication can record a network trace, extract features from that trace, and make an inference about the value of an unknown secret. Our goal in this section is to describe how to measure the strength of this inference process. The ultimate goal is to compare and rank individual features in terms of their usefulness in determining the value of the secret. Here, we fix our attention on the relationship between secrets and a single particular feature of interest, f^j, and so we omit the superscript j for the current discussion and refer simply to f as the feature of interest, and v_i as the i^{th} value of feature f in the given feature profile.

Quantitative information flow Before observing a run of the system, an outside observer has some amount of *initial uncertainty* about the value of the secret. Benign users of the system perform interactions and, meanwhile, an attacker observes the network traces and computes the value of feature f. In our scenario, observing a trace feature results in some amount of *information gain.* In other words, measuring f reduces an observer's *remaining uncertainty* about the secret s. Our goal is to measure the strength of *flow* of information from s to f, which is called the *mutual information* between the feature and the secret. This intuitive concept can be formalized in the language of quantitative information flow (QIF) using information theory [34]. Specifically, we make use of *Shannon's information entropy* which can be considered a measurement of uncertainty [35, 36].

Given a random variable S which can take values in $\mathbb{S}$ with probability function $p(s)$, the *information entropy* of S, denoted $\mathcal{H}(S)$, which we interpret as the observer's *initial uncertainty*, is given by

$$\mathcal{H}(S) = -\sum_{s \in \mathbb{S}} p(s) \log_2 p(s) \tag{2.1}$$

Given another random variable, V, denoting the value of the feature of interest, and a conditional distribution for the probability of a secret given the observed feature value, $p(s|v)$, the *conditional entropy of S given V*, which we interpret as the observer's remaining uncertainty about S, is

$$\mathcal{H}(S|V) = -\sum_{v\in\mathbb{V}} p(v) \sum_{s\in\mathbb{S}} p(s|v) \log_2 p(s|v) \tag{2.2}$$

Given these two definitions, we can compute the expected amount of information gained about S by observing V. The *mutual information* between V and S, denoted $\mathcal{I}(S;V)$ is defined as the difference between the initial entropy of S and the conditional entropy of S given V:

$$\mathcal{I}(S;V) = \mathcal{H}(S) - \mathcal{H}(S|V) \tag{2.3}$$

Probability estimation via profile samples The preceding discussion assumes that the probabilistic relationships between the secret and the feature values are known, i.e. $p(s|v)$. However, since we do not know this relationship in advance, we estimate the conditional probability distribution using the samples generated via profiling.

We begin with a generic discussion of estimating probability distributions from a finite sample set. Let $\mathbf{V}$ be a sample space, V be a random variable that ranges over $\mathbf{V}$, v represent a particular element of $\mathbf{V}$, and $\mathbf{v} = \langle v_1, \ldots, v_n \rangle$ be a finite list of n random samples from $\mathbf{V}$. We estimate the probability of any $v \in \mathbf{V}$ in two ways. Each method relies on a choice of "resolution" parameter, which we make explicit in the following descriptions. The reader may refer to Figure 2.6.

Histogram estimation. We choose a discretization which partitions the sample set $\mathbf{v}$ into m intervals or "bins" where c_i is the count of the samples in bin i. The bins are represented by intervals of length $\Delta v = m/(\max \mathbf{v} - \min \mathbf{v})$. Then for any v, $p(v)$ is estimated by the number of samples that are contained in the same interval as v divided by the total number of samples. The resolution parameter is m and the probability estimator for v which falls in bin i is given by

$\hat{p}(v) = c_i/n$. This estimation of probability is straightforward and commonly used. However, our experiments indicate that, due to the huge search space, our sampling is extremely sparse. Hence, histogram-based probability estimation fails to generalize well to predict the probability of unseen samples.

Gaussian estimation. We can estimate the probability of any $v \in V$ by assuming the sample set comes from a Gaussian distribution. We compute the mean, μ, and standard deviation σ from the set of samples $\mathbf{v}$. We then have an estimate $\hat{p}(v)$ assuming v comes from the normal distribution $N(\mu, \sigma)$. This allows us to more smoothly interpolate the probability of feature values for any v that was not observed during profiling.

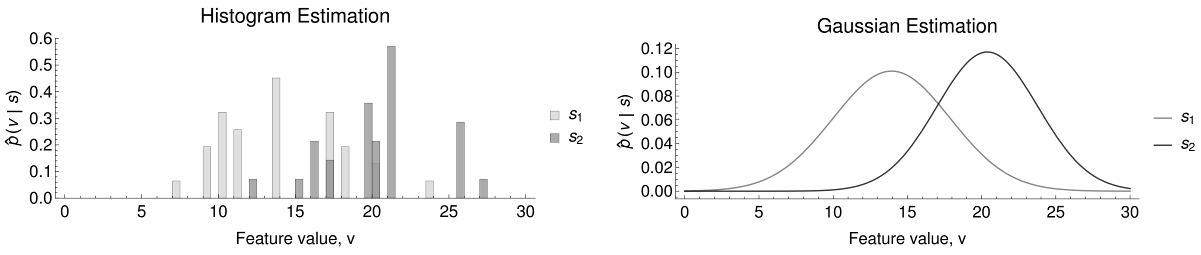


Figure 2.6: Estimating a probability distribution from samples using histograms or Gaussian estimates.

Information gain estimation via profile. We make use of the profile for the current feature of interest f to estimate the expected information gain. We consider a profile P that consists of n pairs of secrets and feature values, $P = \langle (s_1, v_1), (s_2, v_2), \ldots, (s_n, v_n) \rangle$.

For any particular secret $s \in \mathbf{S}$ let $\mathbf{v}_s = \langle v_i : s_i = s \rangle$ be the list of feature value samples that correspond to s. We use $\mathbf{v}_s$ to estimate the probability distribution of the feature value given the secret, $\hat{p}(v|s)$, using either the histogram- or Gaussian-based method. We then compute the probability of a secret value given a feature value, $\hat{p}(s|v)$, using a straightforward application of Bayes' rule. We assume a uniform probability distribution for $p(s)$ and using $\hat{p}(s|v)$, we apply equations 2.1, 2.2, and 2.3, to compute $\hat{\mathcal{I}}(S, V)$, the estimated information gain (leakage) for the secret given the current feature of interest.

Example. Consider a scenario in which we have two possible equally likely secrets, s_1 and s_2. Thus, we have 1 bit of secret information. After conducting profiling for a feature f, we can compute the estimate for the probability of the feature values given the secret values $\hat{p}(v|s_1)$ and $\hat{p}(v|s_2)$ using either histogram-based estimation or Gaussian estimation as depicted in Figure 2.6.

Using histogram-based estimation with the bin-width of $\Delta x = 0.5$ as shown, we observe that the only sample collisions occur at $v = 17$ and $v = 20.5$. Since we observe very few collisions this way, we expect that histogram-based estimation will tell us that there is a high degree of information leakage since most observable feature values correspond to distinct secrets. Indeed, the estimated information gain is 0.8145 bits out of 1 bit.

On the other hand, we have sparsely sampled the feature value space, and if we were able to perform more sampling, we would "fill in" the gaps in the histogram. Hence, using Gaussian distributions to interpolate the density, as show in Figure 2.6, we see that we are much better able to capture the probability of observable feature value collisions. Using the Gaussian probability estimates, we compute that the expected information leakage is 0.4150 bits out of 1 bit, much less than when estimating with the histogram method. We say that the histogram overfits the sampled data. Estimating probabilities from a sparse set of features without overfitting is addressed in multiple works [37–40]. Our experimental evaluation (Section 6.2) indicates that Gaussian fitting works well for estimating entropy in network traffic features.

2.5 Experimental Evaluation

In this section we present the experimental evaluation of Profit on the DARPA STAC benchmark.

2.5.1 DARPA STAC Systems and Vulnerabilities

The applications in our benchmark are from the DARPA Space/Time Analysis for Cybersecurity program [41], which seeks to push the state of the art in both side-channel and algorithmic-complexity vulnerability detection. Algorithmic complexity attacks are beyond the scope of this work; we focus on STAC's side-channel-related applications. These STAC applications [23] include a collection of realistic Java systems, many of which contain side-channel leaks in time or in space, and certain secrets of interest. Some of the systems come in multiple variants, some of which may leak more than others, or have a particular vulnerability added or removed. All the systems are network-based (web-based, client-server, peer-to-peer), and most of the vulnerabilities are based on profiling network traffic and eavesdropping. We have omitted some applications whose side channels are based on other media, such as interception of file I/O, or whose vulnerabilities are exclusively about cryptography.

Airplan is a Web-based client-server system for airlines. It allows uploading, editing, and analyzing flight routes by metrics like cost, flight time, passenger and crew capacities. One secret of interest is the *number of cities* in a route map uploaded by a user; the challenge is to guess this using a side channel in space. Airplan 2 has a vulnerability by which the cells of the table shown on the *View passenger capacity matrix* page are padded with spaces to a fixed width. Thus, the HTML code for the table looks neatly laid out. This is easily overlooked by the end-user, as multiple spaces are rendered as one space by Web browsers, but it does influence the number of bytes transmitted. Thus, the download size of this particular page becomes proportional to the number of cities squared. In Airplan 5, the HTML cell padding is randomized rather than fixed, which dilutes the leakage but does not eliminate it. Airplan 3 does not pad the cells, and is thus much more resilient to this kind of attack; there is still a correlation, but it's a much weaker one.

Another secret of interest in Airplan is the *strong connectivity* of a route map uploaded by an airline. Both Airplan 3 and Airplan 4 have a *Get properties* page that shows various attributes of a route map. Airplan 3 has a vulnerability that causes a slight variation in the byte size of this page depending on whether the route map in question, viewed as a graph, is strongly connected. Airplan 4 does not have this vulnerability. The fault can be exploited to fully leak the secret in the former, while the latter does not leak at all.

Bidpal is a peer-to-peer system that allows users to buy and sell items via a single-round, highest-bidder-wins auction with secret bids. It allows users to create auctions, bid on an auction, find auctions, etc. The secret of interest is the *value of the secret bid* placed by a user. Bidpal 2 contains a timing vulnerability whereby a certain loop is executed a number of times proportional to the maximum possible bid, and a counter is increased; after the counter exceeds the victim power plant's offered amount, a different action is performed per iteration which takes slightly longer. Thus, the total execution time of the loop correlates with the secret.

GabFeed, as explained in Section 2.1, is a Web-based forum where users can post messages, search posted messages, consult special issues, and chat. Users can log in, but may also confirm the server's identity through a challenge-response form. In GabFeed 2, this mechanism is affected by a timing vulnerability in a `modPow()` method, where a branch is only taken when the i-th bit of the server's private key is 1. Thus, a small delay between two network packets in the challenge-response authentication is proportional to the number of 1s in the binary representation of the private key. In GabFeed 1, the `modPow()` method is securely implemented and the vulnerability is not present.

SnapBuddy is a Web application for image sharing; it allows users to upload photos from different locations, share them with friends, and find out who is online nearby. The secret is the physical location of the victim user. During the execution of the *Change user location* operation, a few network messages are sent, including one whose size correlates with the destination location. By careful manual inspection one can confirm that each one of the 294 known locations has a unique associated message size, thus providing a unique signature for each location. However, the crucial message may impact the size of one, two, three, or up to four adjacent packets depending on its total size. Thus, one should pay attention to the sum of those packets.

PowerBroker is a peer-to-peer system used by power suppliers to exchange power. Power plants with excess supply try to sell power, whereas those with a shortfall try to purchase it. The secret of interest is the value offered by one of the participating power plants. PowerBroker 1 has a vulnerability in time whereby a certain loop is executed a number of times that is proportional to the amount of the price, in dollars, offered for the power. This induces a time execution difference that ends up affecting network traces. In PowerBroker 2 and PowerBroker 4, this loop is always executed a constant number of times, which removes the vulnerability. In addition to this, in PowerBroker 2 as in Bidpal 2, the behavior of the program changes when loop counter reaches the bid. However unlike Bidpal 2, this change in behavior does not impact the time taken for a loop iteration so the program remains non-vulnerable.

TourPlanner is a client-server system that, given a list of places that the user would like to visit, calculates a tour plan that is optimal with respect to certain travel costs. It is essentially a variation of the traveling salesman problem. The secret of interest is *the user-given list of places*. The TourPlanner system has a subtle timing vulnerability. The computation can take a while, so the server sends periodic progress-report packets to the client. Their precise timing exposes the duration of certain internal stages of the computation. There are five consecutive packets of which the four time-deltas in between (i.e., the time differences between each packet and the following one) are particularly relevant. Each of these deltas, by itself, leaks just a little information about the secret. Their sum leaks more information than each of them separately. And when interpreted as a vector in $\mathbb{R}^4$, they constitute a signature for the secret list of places with a high level of leakage.

2.5.2 Experimental setup

Profiling-input suite generation

In many real-world contexts, one can leverage existing input suites and/or existing input generators that might be available for the system.

If no input suite or input-generating script is available, we will need to generate inputs to run the system. Generating complex structured inputs for black-box execution of a system is a nontrivial task, and its full automation is beyond the scope of this work.

Manually designing a profiling-input suite generator compels us to consider the following goals:

1. Secret domain coverage: We want to exercise the system for many different secrets, i.e., choose a secret set $\mathbf{S}$ that is reasonably representative of the secret domain $\mathbb{S}$.

2. Input domain coverage: We want to choose an input set $\mathbf{I}$ that is reasonably representative of the input domain $\mathbb{I}$. Typically, for each secret $s \in \mathbf{S}$ we may need many different inputs $i \in \mathbb{I}$ such that $\zeta(i) = s$. Since such inputs may differ from each other in various different ways, we may want to sweep several dimensions to capture a representative subset.

3. Sampling for noise resilience: We want to run each input $i \in \mathbf{I}$ multiple times so that system noise can be modeled and accounted for, especially if we know or suspect that the system may have a strong degree of nondeterminism.

4. Cost of execution: The product of the above can spawn a large set of inputs. Depending on the system, executing them may be costly. Cost and coverage is a classic trade-off.

For all the experiments presented in this work, the inputs were created by generalizing the example interaction scripts that were included with the documentation of each system. Based on the available scripts and documentation, we identified the main degrees of freedom and strived to sweep each of those dimensions as uniformly as possible, all while keeping the total execution time of the Cartesian product within our resource availability.

The size of our input suites varies from one application to another because some applications take up to two orders of magnitude longer than others to execute each interaction. The number of parameters also varied from one application to another because different applications' inputs involve different orthogonal degrees of freedom. Whenever multiple applications were executed for the same secret, we used the same input suite for all of the applications.

Trace alignment parameters

When aligning biological sequences, MSA tools are sensitive to parameter tuning. Aligning network packet sizes, however, seems to be an easier task. Our data often contains arbitrarily long unalignable regions, so we set MAFFT to the mode recommended for that purpose by its developers, and left all other parameters untouched at their default values.

Phase detection parameters

We used $\omega = 3$ (minimum stable phase width), $\psi = 0.25$ (maximum stable column diversity), and a maximum size of up to 1000 traces for the subset that we sent to MAFFT for alignment. For the input suites that consist of less than 1000 traces (see Table 2.2), external alignment sufficed. For input suites with more than 1000 traces, the traces that did not parse due to anomalies (see Section 2.3.2) were always less than 1%.

2.5.3 Experimental results

In this section, we are going to discuss our results and explain our findings on DARPA STAC benchmark.

Confirmation of limitations of the histogram approach

Figure 2.7 shows the leakage results over three Airplan applications with both Gaussian and histogram-based estimation with various bin sizes. In this figure, we can see that Gaussian estimation is estimating 100%, 25% and 79% for three Airplan applications. The leakage results of histogram estimation vary with different the bin sizes with overfitting in lowest bin size and underfitting in largest bin size. Assuming the feature is sampled from a Gaussian distribution, if we fix the bin size according to one application, it either overestimates or underestimates the leakage for other applications.

Example of Profit output

Table 2.4 shows the results we get from Profit where we obtain leakage for each extracted feature ranked according to leakage percentage in decreasing order.

Results for vulnerable applications

Table 2.2 summarizes the results returned by Profit for applications with a known side-channel vulnerability. For each application we show the secret leaked by the known vulnerability and the type of the vulnerability (in space or in time). We also report the number of distinct secrets, distinct inputs, and executions per input that were used during profiling. On the right side, we show the results returned by Profit. The *Best feature* column shows, among the features that were present in Profit's output ranking, the one that most specifically and closely captures the leakage induced by the known vulnerability. The *Rank* column indicates the position within Profit's output ranking in which said feature appeared. The *Leak* column shows the percentage of information leakage computed by Profit.

In 6 out of 7 cases, the best feature that most closely leads to the vulnerability appeared at the very top of Profit's ranking. In all cases, it appeared within the top-five. In all cases where the vulnerability fully leaks the secret, Profit computed a leakage of 95% or more, except in the cases of Bidpal, PowerBroker 1, and TourPlanner. For Bidpal and PowerBroker 1, a larger number of samples per input would be needed in order to compensate for the noise, but this was hard to obtain because both applications take several minutes per execution. In the case of TourPlanner, where each sample takes very little time, Profit actually identified all four relevant time-deltas, which appeared within the top-10 with leakages of about 14% to 16% each. As mentioned in Section 2.5.1, an even higher leakage (by no means 100%, but probably above 50%) can be achieved by considering all four deltas together as a multi-dimensional feature, but, as explained in Section 2.6, this is beyond the abilities of the current version of Profit. Remarkably, although it only handles one feature at a time, Profit correctly inferred that the *sum* of the four deltas (i.e., the total duration of the phase that isolated them) yielded a greater leakage than any of the four separately, and reported that feature at the top of the ranking. It is also worth noting, when looking at the *Best feature* column, that the phase detection mechanism allowed Profit to be very specific about the location of the features listed at the top of its rankings. Even in cases where the data was insufficient to reach a fully accurate quantification of the leakage, Profit was able to point the user to the right features.

Results for groups of vulnerable and non-vulnerable applications

Table 2.3 summarizes the results returned by Profit for each group of applications associated with a particular side-channel vulnerability. Each group begins with the vulnerable application that was shown in Table 2.2, followed by other applications in which the vulnerability has been mitigated or eliminated. For each application we show the secret leaked by the known vulnerability and the type of the vulnerability (in space or in time). We also show whether the vulnerability is present or not. On the right side, we show the results returned by Profit. For the first application of each group, the *Best feature* and *Leak* columns show the same values as Table 2.2. For the other applications in the group, the *Best feature* column shows the same feature and the *Leak* column shows the percentage of information leakage computed by Profit for

Application	Secret	# Secs.	# Unique inputs	# Runs per input	Type	Best feature	Rank	Leak
Airplan 2	Number of cities	13	500	5	Space	Sum ↓ phase 4	1	100%
Airplan 3	Strong connectivity	2	500	5	Space	Pkt 10 ↓ phase 3	1	100%
SnapBuddy 1	Location of user	294	294	10	Space	Sum ↑ phase 2	1	95%
Bidpal	Secret bid value	49	49	4	Time	Δ 19-20 ↓ full trace	1	59%
GabFeed 1	No. of 1s in key	12	60	5	Time	Δ 4-5 ↓ phase 2	1	100%
PowerBroker 1	Price offered	49	49	4	Time	Δ 9-10 ↑ full trace	4	60%
TourPlanner	Places to visit	250	250	20	Time	Total time ↓ phase 3	1	30%

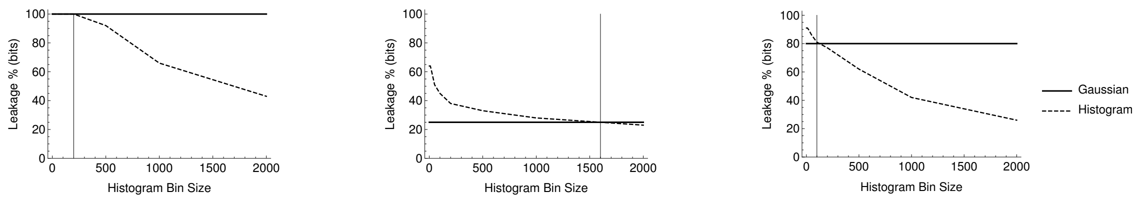
Table 2.2: Profit results on vulnerable applications.

Application	Secret	Type	Vulnerable?	Best feature	Leak	Top feature	Leak
Airplan 2	Number of cities	Space	Yes	Sum ↓ phase 4	100%	Sum ↓ phase 4	100%
Airplan 5	Number of cities	Space	Partially	Sum ↓ phase 4	79%	Sum ↓ phase 4	79%
Airplan 3	Number of cities	Space	No	Sum ↓ phase 4	25%	Packet 20 ↓ full trace	36%
Airplan 3	Strong connectivity	Space	Yes	Packet 10 ↓ phase 3	100%	Packet 10 ↓ phase 3	100%
Airplan 4	Strong connectivity	Space	No	Packet 10 ↓ phase 3	0%	Packet 1 ↑ phase 2	4%
Bidpal 2	Secret bid value	Time	Yes	Δ 19-20 ↓ full trace	59%	Δ 19-20 ↓ full trace	59%
Bidpal 1	Secret bid value	Time	No	Δ 19-20 ↓ full trace	9%	Δ 16-17 ↑ full trace	19%
GabFeed 1	No. of 1s in key	Time	Yes	Δ 6-7 ↓ full trace	100%	Δ 6-7 ↓ full trace	100%
GabFeed 5	No. of 1s in key	Space	No	Δ 6-7 ↓ full trace	24%	Δ 6-7 ↓ full trace	24%
GabFeed 2	No. of 1s in key	Time	No	Δ 6-7 ↓ full trace	19%	Δ 11-12 ↕ full trace	20%
PowerBroker 1	Price offered	Time	Yes	Δ 9-10 ↑ full trace	60%	Total time ↕ full trace	60%
PowerBroker 2	Price offered	Time	No	Δ 9-10 ↑ full trace	13%	Total time ↕ full trace	13%
PowerBroker 4	Price offered	Time	No	Δ 9-10 ↑ full trace	9%	Δ 16-17 ↑ full trace	18%

Table 2.3: Profit results on vulnerable vs. non-vulnerable apps.

that feature. Finally, for all applications, the *Top feature* column shows the feature that appears at the top of Profit's ranking (or the most specific one, in the event of a tie between features that subsume each other).

For all application groups we can see that, as the vulnerability is mitigated or removed, the leakage computed by Profit decreases significantly and in the correct relative proportion. While we have no firm guarantee that the computed leakages are exact (since, as stated in Section 2.6, they depend on the input suite), we can observe that they are always consistent with the known facts about the different DARPA STAC applications and their present and absent vulnerabilities. Lastly, in 8 out of 13 cases, the top feature reported by Profit is indeed the best feature, and in all other cases, the top feature reported was not significantly higher (in rank or in leakage) than the best one.



(a) AIRPLAN 2 (b) AIRPLAN 3 (c) AIRPLAN 5

Figure 2.7: Information leakage comparison of Gaussian and histogram-based entropy estimation for changing bin sizes for three versions of the AIRPLAN application.

Rank	Feature	Dir.	Subtrace	Leak (%)	Leak (bits)
1	Total size	↓	Phase 4	79%	2.94 of 3.70
1	Total size	↕	Phase 4	79%	2.94 of 3.70
1	Total size	↓	Full trace	79%	2.94 of 3.70
4	Packet 20 size	↓	Full trace	59%	2.16 of 3.70
5	Packet 27 size	↓	Full trace	56%	2.10 of 3.70
6	Packet 24 size	↓	Full trace	53%	1.97 of 3.70
6	Packet 28 size	↓	Full trace	53%	1.97 of 3.70
8	Packet 21 size	↓	Full trace	50%	1.86 of 3.70

Table 2.4: Feature ranking returned by Profit for AIRPLAN 2.

2.6 Limitations

Quality of the profiling-input suite

The most important limitation of our approach to keep in mind is that the quality of the leakage quantifications computed by Profit depends on the quality of the profiling-input suite. Our ability to accurately quantify leakage is strongly linked to our ability to accurately estimate the likelihood of collisions between observations from different secrets. Ideally, we would like to increase the size and diversity of our input set $\mathbf{I}$ to be as close as possible to the input domain $\mathbb{I}$, so that the probability distribution of collisions would approach the one that we would see if we could afford to execute all of $\mathbb{I}$. If $\mathbf{I}$ is so small that it hardly ever causes any of those collisions, leakage could be overestimated. On the other hand, if the suite is too large, it may be unfeasible to execute it due to resource constraints.

Normal distribution of feature values

We assume that, for a given feature, and for each secret, the probability of the feature given the secret follows an approximately normal distribution. We thus model the probability density function for each secret with a Gaussian curve. If the user expects different distributions, or a chi-squared test reveals that these distributions are far from being normal, one may want to model the probability density functions using a different kind of distribution.

One-dimensionality of features

The feature space that we consider in this work is intentionally limited to one-dimensional features. We compute the leakage for many features, but consider them one at a time. As exemplified by the TourPlanner vulnerability explained in Section 2.5.1, when several features are combined in just the right way, they can leak more than each one of them separately (or than all of them combined in a trivial way). Quantifying the joint leakage of combined features is simple when one can assume that all the features are independent, but in this context, that is almost never the case. Quantifying the joint leakage (that is, the correlation with the secret) of multiple features that are partially correlated between themselves is a complex matter, which is beyond the scope of this article and we will address in future work. Nevertheless, in many cases Profit will still report partial leakage for one or more of the combinable features, which can at least point the user in the right direction (see Section 2.5.3).

2.7 Chapter Summary

In this chapter, we presented Profit. Profit combines network trace alignment, phase detection, feature selection, feature probability distribution estimation and entropy computation to quantify the amount of information leakage that is due to network traffic. Our experimental evaluation on DARPA STAC benchmark demonstrates that Profit is able to identify the features that leak information for the vulnerable applications. Moreover, Profit is able to correctly order the amount of leakage in different variants of the same application.

Chapter 3

Test Input Generation for Network Side Channel Analysis

In this chapter, we present AutoFeed, a tool for feedback-driven black-box profiling of software systems that detects and quantifies side-channel leakage automatically. [21] The user provides some seed inputs for the target system, and a set of mutators which, given a valid input, return another one. The user chooses a *secret of interest*—some aspect of the input that they consider sensitive, whose leakage they want to detect and quantify. AutoFeed then repeatedly executes the target system, generates new inputs, captures network traffic, and adjusts input generation and system execution strategies based on the feedback it obtains by analyzing captured traffic.

Modern systems use encryption. AutoFeed analyzes side channels in network traffic—the visible aspects of traffic that eavesdroppers can easily capture despite encryption, such as the size, timing, and direction of network packets. AutoFeed extracts meaningful features from these visible characteristics, and uses conditional entropy to find features that maximize information gain about the secret of interest. For example, it may find that the time elapsed between certain packets leaks some amount of information about the secret. The final output from AutoFeed is an automatically generated *ranking* of the top n most-leaking features, sorted by how much information they each leak about the secret of interest.

There has been prior work on quantifying leakage in network traces in particular [18, 42] and in program traces in general [20, 43]. However, all of them rely on manually generated input suites and do not address the problem of the quality of the input suite. AutoFeed automates the manual effort of providing inputs. Instead, the user writes mutators to explore the input space. An automated feedback loop progressively generates and runs more inputs and improves accuracy of leakage estimation. AutoFeed also automates the assessment of usefulness of different mutators and the stop criterion that determines when the leakage estimation and the output feature ranking become stable, avoiding diminishing returns of computational effort. Compared to prior work, AutoFeed enhances the degree of automation significantly, reduces the amount of wasted profiling effort, and improves the reliability of the results.

My contribution in this chapter is to present a feedback-driven, black-box technique to detect and quantify side channels using mutator-based input generation, statistical modeling of the observed data, and a stop criterion to detect convergence of leakage estimation. AutoFeed runs incrementally, generates more inputs as needed, caches inputs to avoid repetitions, and stops running when its iterative leakage quantification stabilizes. In particular we present:

1. An automated search mechanism to determine crucial hyperparameter values dynamically based on feedback, in order to estimate probability distributions for modeling the observed data, and a comparative study of techniques to model the observed data using histograms, Gaussian distributions, and kernel density estimation (KDE).

2. A mechanism to focus input space exploration on dimensions that provide more information about the leakage. AutoFeed lets users model the input space using mutators, and relate them to different dimensions of the input space. AutoFeed automatically explores each dimension and assigns weights to the mutators. The effort invested in exploring each dimension is proportional to how much each dimension fosters changes in leakage estimation.

3. An automated stop criterion that halts the input generation process once the leakage estimate stabilizes. This allows convergence of the leakage estimation to a value close to the ground truth independent of the starting input set.

4. Experimental evaluation of the effectiveness of AutoFeed on handcrafted examples with known quantitative ground truths and on the DARPA Space/Time Analysis for Cybersecurity (STAC) benchmark [23], which consists of realistic-sized software systems (Web, client-server, and peer-to-peer) developed by DARPA, in both controlled, low latency and less controlled, high latency network conditions in order to evaluate side-channel vulnerability detection techniques.

The rest of the chapter is organized as follows. In Section 3.1 we provide motivation and an overview of our approach. In Section 3.2 we describe the core techniques and heuristics we developed for AutoFeed. In Section 3.3 we present an experimental evaluation of AutoFeed. In Section 3.4 we provide a summary of this chapter.

3.1 Motivation and Overview

Generating a set of profiling inputs to quantify information leakage presents unique challenges. The problem is quite different from generating an input suite for testing. In traditional testing, the goal is to find inputs that violate assertions or crash the system. In side-channel profiling, the goal is to characterize the relationship between a certain *secret* (i.e., some private or sensitive variable) and the *publicly observable output* of the system, such as the timing and sizes of encrypted network packets. Many new issues arise. We do not know how inputs and outputs are related. We do not know how outputs and secrets are related. Each observable feature may reveal very little or very much about a secret. For each secret, there is an immense space of output features that could leak information about it—the timing of a particular network packet, the time elapsed between two packets, the size of a packet, the sum of sizes of a subset of the packets, etc. Given an observable output feature, it is hard to figure out how its value relates to the value of the secret.

Challenge: Foster collisions. Suppose a secret is picked. Given a set of inputs, each with a different value of the secret, suppose we run each input through the target system. If the set is small, we will find some feature (say, the time of a certain network packet) that takes a unique value for each secret value. Based on such observations, one might be misled into concluding that the feature fully leaks the value of the secret. But the actual leakage could be much lower, or even none, because: (1) If we generate more inputs, we may observe the same value of the feature for two inputs with different secrets. We call these *collisions.* (2) If we run the same input twice, due to system *noise,* we may see different feature values for the exact same input. These two phenomena, collisions and noise, create complex relationships between secrets and features.

It is desirable to find inputs that foster collisions between secrets in each of the system's observable output features. Imagine that we probe a medical system to see how much information it leaks about a patient's age when a patient's record is accessed by medical staff through the network. If we profile the system with a small sample (e.g., fetch 10 patient records), we may observe that the size of a certain packet changes with the age of the patient. But the size of the packet could have taken a unique value for each of the 10 executions by coincidence. A collision occurs when we fetch the records of two patients with different ages (say, 18 and 57) for which that packet has the same size (say, 215 bytes). This introduces uncertainty: an eavesdropper that captures an interaction with a 215-byte packet cannot tell if the patient is 18 or 57 years old. Thus, the feature does not *fully* leak the secret. As we fetch more records, if the observed collision rate progressively approaches the actual rate, our quantification of information leakage will progressively approach the actual amount of information leaked. An input set with an overly low collision rate (w.r.t. the full input space) will result in overestimating the leakage. Finding inputs that foster collisions in the most-leaking features improves the estimation.

Now consider time features, such as the time elapsed between the third and fourth packets of each interaction. We want to quantify how much information this feature leaks about patient age. Since time is continuous, the probability of seeing the exact same value for two inputs is zero. Even if we run *the exact same input* multiple times, we will see slightly different values. This is due to system noise, such as variance in network latency. By running each input multiple times, we can model the noise as a probability distribution. Collisions occur when distributions overlap. The greater the overlap, the more uncertainty about the secret value, resulting in a lower estimation of the amount of information leaked.

Note that the relationships between inputs, outputs and secrets are arbitrary: they depend on the behavior of the target system. The same is true of noise, and software pseudo-randomness can add arbitrary extra noise. Hence, there are no general rules to build an adequate black-box input suite before the analysis. Statically crafted input suites can always lead to incorrect results.

Challenge: Explore a vast input space. To compute the exact amount of information leaked by a system about a secret when performing an action, we would need to execute the action for every input, which is generally not feasible. How many inputs we are willing to execute depends on how long it takes to run each one and how long we are willing to wait for the analysis to complete.

System inputs can be complex and may include structured data. For example, the AIRPLAN system from the DARPA STAC benchmark (see Section 3.3.3) takes as input an arbitrary graph of airports and flight routes, and each edge is decorated with six different weights. Inputs to DARPA's RAILYARD system involve different kinds of train cars, different types and quantities of cargo, crew members, train routes, stops, schedules, and more. The possibilities are endless and depend on the system. Each output feature can be affected by *any part* of the input—including those that are related to the secret of interest, and those that are not.

Prior work [18, 20, 42, 43] requires the user to provide the full input suite *before the analysis* *begins.* Thus, the user must sample the input space in some way that covers all its dimensions adequately. But, even for one feature, the user cannot know in advance which input dimensions will foster changes in the leakage estimation of that feature. To make things worse, there is an enormous space of observable features. If the user tries to be conservative and cover all bases, combinatorial explosion results in a prohibitive number of inputs. If the user tries to reduce the input set to keep the analysis time feasible, leakage estimation results may be incorrect.

Challenge: Quantify the leakage. Extracting probability distributions from observable features is nontrivial. Histograms can overfit the data and lead to false positives. Gaussian fitting can over-abstract the data: if it is not normally distributed, the model will be wrong. For example, when a feature is multi-modal, Gaussian fitting will produce a unimodal approximation, and false overlaps will underestimate the leakage. Kernel density estimation [44] offers greater flexibility and is well-suited for a wide variety of data, but heavily depends on the *window size* or bandwidth; too small a value leads to similar problems as with histograms, whereas too large a value can lead to similar problems as with Gaussian fitting.

Overview of our approach. As said above, crafting an input suite before the analysis is tedious and risky. Different input suites can lead to different leakage quantification results. AutoFeed offers a mutation-based mechanism to specify the space of valid inputs. It automatically generates new inputs on demand using a feedback loop. Manual user effort is limited to writing the mutators, providing a small set of initial seeds, and choosing the secret of interest. (The mutators, once written, can be reused for many secrets.) AutoFeed automates everything else. It iteratively mutates inputs and periodically quantifies the leakage to update its belief about which features leak the greatest amount of information about that secret. In doing so, it automates both the exploration of the output feature space and the exploration of the input space. Since the user cannot know which mutators will be most important for a secret, AutoFeed measures the effect of different mutators on leakage estimation, and weighs them accordingly (see 3.2.3) in a feedback-driven way. By focusing the computational effort on those mutators that have greater effect on the leakage estimation of the most promising features, the input space is explored efficiently.

Quantification computation is nontrivial. We conducted a comparative analysis of different approaches (see 3.3.5). AutoFeed automatically discovers the distribution of observed data using KDE and automatically finds a suitable bandwidth parameter (see 3.2.4).

By enforcing a stop criterion, AutoFeed automates evaluating whether the leakage estimation is stable enough (see 3.2.5). This reduces the risks associated with having to manually decide when to stop. It also allows AutoFeed to run analyses batches unattended.

3.2 Feedback-driven Side-Channel Analysis

In this section we provide some basic definitions and explain the main algorithms and heuristics used in AutoFeed.

3.2.1 System Model

Assume that a software system, use case, and secret of interest are selected by the user. We reuse the following definitions from the system model in Section 2.2 and recap it in this section. The *input domain* $\mathbb{I}$ is the set of all valid inputs for the use case. The *secret domain* $\mathbb{S}$ is the set of all values that the secret of interest can take. Given an input, the *secret function* $\zeta : \mathbb{I} \longrightarrow \mathbb{S}$ projects its secret value. Running every input in $\mathbb{I}$ is usually not feasible: the *input set* $\mathbf{I} \subseteq \mathbb{I}$ is the set of distinct inputs that are executed during an analysis. Since the secret is a function of the input, by choosing a set of inputs, we are also choosing a set of secrets. The *secret set* $\mathbf{S} \subseteq \mathbb{S}$ is the set of distinct secrets that appear in some input during an AutoFeed analysis. Assuming a generalized $\zeta : \mathcal{P}(\mathbb{I}) \longrightarrow \mathcal{P}(\mathbb{S})$, we can say that $\zeta(\mathbf{I}) = \mathbf{S}$. A *packet* is an abstraction of a real network packet. We assume packets are encrypted. Decrypting them is beyond the scope of this work. We consider side-channel characteristics of each packet: its size, time, and direction in which it flows. Each time we execute an input $i \in \mathbb{I}$ through the system, we capture a *network trace*, which is a sequence of packets. We also add the following definitions. A *seed* is an input $i \in \mathbb{I}$ provided by the user. A *mutator* is a function $m : \mathbb{I} \longrightarrow \mathbb{I} \cup \{\mathsf{None}\}$ that, given a valid input, returns another valid input, or None if the input cannot be mutated by m. For instance, in our AIRPLAN example, the RemoveFlight mutator removes one of the direct flights between two airports. This mutator cannot be applied to a map in which all direct flights have been removed. Lastly, an *initial set* of inputs $D \subseteq \mathbb{I}$ is a set of inputs obtained by applying some amount of random mutation to the seeds.

Procedure 1 AUTOFEED(App, I, M, RPI, C) Given an application App, an initial set of inputs I, a set of mutators M, a repetition per input value RPI, and a time budget per iteration C, AUTOFEED quantifies the leakage using a feedback loop.

```
1: $Traces \leftarrow$ EXECUTE($App, I, RPI$)
2: $N \leftarrow C/($AVGTIME($Traces$) $\times RPI)$ ▷ Calculate the number of inputs ($N$) to generate per iteration, given $C$ seconds of time budget per iteration
3: $I' \leftarrow$ MUTATE($I, M, N, \vec{W}_{uniform}$) ▷ Generate new inputs using mutators where each partition of mutators has equal weight
4: $Traces' \leftarrow$ EXECUTE($App, I', RPI$)    ▷ Generate corresponding traces
5: $I \leftarrow I \cup I'$
6: $Traces \leftarrow Traces \cup Traces'$
7: $Leak' \leftarrow$ QUANTIFYLEAKAGE($Traces$)
8: $\langle \vec{W} \rangle \leftarrow$ GETWEIGHTS($App, I, M, N$)    ▷ Compute the weights for the mutators
9: repeat    ▷ Main loop for feedback-driven exploration
10:     $Leak \leftarrow Leak'$
11:     $I' \leftarrow$ MUTATE($I, M, N, \vec{W}$)    ▷ Generate new inputs using mutators
12:     $Traces' \leftarrow$ EXECUTE($App, I', RPI$)    ▷ Generate corresponding traces
13:     $I \leftarrow I \cup I'$
14:     $Traces \leftarrow Traces \cup Traces'$
15:     $Leak' \leftarrow$ QUANTIFYLEAKAGE($Traces$)
16: until $|Leak' - Leak| < \epsilon$    ▷ Stop criterion check convergence of leakage value
17: return $Leak'$
```

3.2.2 AutoFeed Workflow

The high level algorithm demonstrating the workflow of the AutoFeed tool is shown in Procedure 1. AutoFeed requires the following inputs from the user: an application to run App, initial seed inputs I, a set of mutators M, value for repetitions per input RPI, and a time budget per iteration C. First, AutoFeed executes the App with the initial seed inputs to generate

Procedure 2 GetWeights(App, I, M, N) Given an application App, a set of inputs I, a set of mutators M and a partition, and number of inputs to generate N, GetWeights computes weights for subsets of mutators.

```
1: $Traces \leftarrow$ Execute($App, I, RPI$)                    ▷ Generate corresponding traces
2: $F \leftarrow$ ExtractFeatures($Traces$)                      ▷ Extract features over traces of original inputs
3: for each subset $M_i$ of $M$ do                               ▷ where the $M_i$ are a partition of $M$
4:     $I_i \leftarrow$ Mutate($I, M_i, N$)                      ▷ Generate inputs using a subset of mutators
5:     $Traces' \leftarrow$ Execute($App, I_i, RPI$)
6:     $F' \leftarrow$ ExtractFeatures($Traces'$)  ▷ Extract features over traces of mutated inputs to estimate weight of $M_i$
7:     $\vec{W}[i] \leftarrow \sum_j \mid F'_j - F_j \mid /(F_{max} - F_{min}) + [Sec'_j \neq Sec_j]$   ▷ Weight of the current subset of mutators is proportional to number of mutated inputs with a different feature value or secret
8: $W_{sum} \leftarrow \sum_i W[i]$
9: for each $\vec{W}[i]$ do                                      ▷ Normalize the mutator weights
10:     $\vec{W}[i] \leftarrow \vec{W}[i]/W_{sum}$
11: return $\langle \vec{W} \rangle$
```

Procedure 3 Mutate($I, M, N, \vec{W}$) Given a set of inputs I, a set of mutators M, number of inputs to generate N, and mutator weights $\vec{W}$, Mutate generates new unique inputs using the mutators.

```
1: $I_{new} \leftarrow \emptyset$
2: while $|I_{new}| < N \wedge |I| > 0$ do
3:     $i \leftarrow$ RandomSelect($I$)                          ▷ Select a random input
4:     $M' \leftarrow M$
5:     $done \leftarrow false$
6:     while $M' \neq \emptyset \wedge \neg done$ do
7:         $m \leftarrow$ RandomSelect($M', \vec{W}$)            ▷ From set $M'$ according to weights $\vec{W}$
8:         $i_{new} \leftarrow m(i)$
9:         if $i_{new} \in I \vee i_{new} =$ None then
10:             $M' \leftarrow M' - \{m\}$                       ▷ If a mutator does not create a new input, drop it
11:         else
12:             $I_{new} \leftarrow I_{new} \cup \{i_{new}\}$
13:             $done \leftarrow true$
14:     if $\neg done$ then
15:         $I \leftarrow I - \{i\}$                             ▷ If no mutator yields a new input, drop it
16: return $I_{new}$
```

an initial set of traces. Based on these initial traces, it calculates the number of inputs to generate per iteration (N) that corresponds to the given input time budget per iteration (C). Then, it applies the mutators on the seed inputs to get new inputs, executes the *App* on these inputs, and uses the traces obtained from these executions to obtain an initial estimation of the information leakage. Using the initial leakage results, AutoFeed uses heuristics to compute weights for mutators, where the weight of each mutator corresponds to the likelihood of applying that mutator during input generation. After these initialization steps, AutoFeed starts executing its main loop for feedback-driven exploration of the input state space for obtaining an accurate estimation of information leakage. In each loop iteration, AutoFeed uses mutators to generate new inputs, executes the *App* on new inputs to generate corresponding traces, and updates the leakage estimation using all the traces captured so far. When the change in the leakage estimate falls below a small value (ϵ), AutoFeed terminates execution and reports the computed leakage.

In the main workflow of the AutoFeed tool shown in Procedure 1 we use two other procedures that we discuss below: GetWeights, and Mutate. For the sake of readability and clarity of presentation, we present all these procedures from the perspective of a single feature (the top feature) corresponding to the feature that leaks the most amount of information. In actual implementation of AutoFeed, a large set of features are taken into account and their leakage is estimated until termination. After the initial input generation step and initial leakage estimation, only for GetWeights top k features are selected as we believe mutators that discover more behaviors on those features will impact the leakage results.

3.2.3 Assigning Weights to Subsets of Mutators

AutoFeed uses the user-provided mutators to generate new inputs and explore the input space. It is not possible to know in advance which mutators would be more effective in exploration of the information leakage. Some mutators may generate new secret values which may help our analysis by improving the information leakage estimation. Some mutators may generate inputs with the same secret value but different feature values which can again help our analysis by improving the information leakage estimation. On the other hand, some mutators may generate inputs that do not provide any new insight to the relationship between the secret and the observable features. For example, some mutators may change the input without modifying the secret or any of the observable features. Such mutators will not help our analysis in improving the information leakage estimation. AutoFeed evaluates the influence of mutators on the leakage estimation based on changes in top feature or secret and computes weights for mutators which are proportional to their likelihood of changing secret value or perturbing feature values. These weights are then used to bias the random selection of the mutators where each mutator is selected with a probability that is proportional to its weight. Hence, the mutators that influence the leakage estimation less are chosen less frequently and the mutators that influence the leakage estimation significantly are chosen more frequently.

To do this analysis, the user groups the mutators into subsets. We call these subsets of mutators *dimensions*. Mutators can be grouped by the attribute they are modifying. For instance, in the Railyard system, mutators that add/remove stops from the train schedule are one dimension, whereas those that add/remove personnel from the train crew are another dimension. Mutators can also be grouped by the magnitude of the change that they cause on the input: if a mutator increases an input field by 1, and another mutator increases it by 1000, we may want them to be weighted separately.

We assume that the user provides a partition of the set of mutators, so that each mutator belongs to a single dimension, and each dimension is a subset of the set of mutators. To assess the impact of each subset of mutators on the leakage estimation, for each subset, we generate and run inputs generated only using mutators in that particular subset. Using the traces of these runs and previous traces, we quantify the leakage and record the amount of change in the leakage between this step and the previous step. After we do this test for each subset of mutators, we weigh each subset proportionally to the amount of change in leakage we recorded for that subset of mutators. Psuedocode for this process is given in Procedure 2.

3.2.4 Leakage Quantification

This section describes how QuantifyLeakage function in Procedure 1 works. To quantify information leakage, we start from a set of captured traces, each one labeled with the secret value associated with that trace, and we align packets using markers inserted at runtime which denote different stages of the interaction. We then extract the related packet based features (such as packet timing and size) and aggregated features (such as total duration, total size, etc.) obtained using alignment. After obtaining the features, we can estimate the probability distribution of features per secret using multiple methods and compute the mutual information between the secret and feature using the estimated probability distribution for each feature. We use *Shannon entropy* [45] to calculate the mutual information $I(\mathbf{S};\mathbf{V})$ and it is derived as

$$I(\mathbf{S};\mathbf{V}) = -\sum_{s\in\mathbf{S}} p(s)\log_2 p(s) - \left(-\sum_{v\in\mathbf{V}} p(v)\sum_{s\in\mathbf{S}} p(s|v)\log_2 p(s|v)\right)$$

where $\mathbf{S}$ and $\mathbf{V}$ are the sets of secret and feature values and we estimate $p(v|s)$ for each secret, $p(s)$ is assumed to be uniform and $p(s|v)$ and $p(v)$ are estimated using Bayes' rule. The first term represents the initial amount of information about the secret. The second represents the remaining uncertainty after observing the feature. The difference is the amount of information gained by observing that feature. For more details, see [18, 45].

The simplest way of estimating the shape of the data distribution is by modeling it as a histogram. This method puts the data in discrete bins where the ratio of elements determine the probability. One problem with this method is that its results are dependent on the bin size and determining an ideal bin size is difficult. If we conservatively choose the smallest bin size we can, then collisions will go undetected unless a huge number of samples is used.

Another method is modeling the data distribution as a Gaussian distribution where the mean $\hat{\mu}$ and standard deviation $\hat{\sigma}$ of the data is obtained and $\hat{p}(x)$ is estimated as $N(x; \hat{\mu}, \hat{\sigma})$. This method extrapolates well but is based on the strong assumption that the data is normally distributed. This assumption may fail if the data is generated from a more complex distribution. Whenever the assumption fails, the data is underfitted: spurious collisions arise, and the information leakage tends to be understated.

Another way of estimating probability distributions is using *kernel density estimation* (KDE) [44]. Using KDE, we can estimate the distribution of data without assuming a specific distribution. Unlike a histogram, our estimation is smooth, which helps us model continuous data better and extrapolate to unseen data more easily. If we want to estimate $p(x)$, the kernel density estimator $\hat{p}(x)$ is

$$\hat{p}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

where n is number of samples, h is the positive bandwidth parameter and K is a non-negative function called *kernel*. There are various kernel functions: uniform, triangular, Gaussian, Epanechnikov, etc. The bandwidth h affects our estimation greatly. If it is too small, it overfits the data we have; if it is too large, it underfits the data.

In this work, we have used two methods for bandwidth selection. First selection method is the optimal bandwidth if the underlying distribution is Gaussian in which bandwidth $h = 1.06\hat{\sigma}n^{-1/5}$, where $\hat{\sigma}$ is the standard deviation of the data [46]. Second method is more general and instead of assuming any underlying distribution, we use statistical cross-validation techniques to select the ideal bandwidth. We use grid search which is used for hyper-parameter optimization by training using a set of candidate parameters on a model (KDE in this case) and evaluating each trained model. The evaluation metric is obtained using *repeated k-fold cross-validation* where the data is split into k equal subsets and for a single subset, we use the other $k - 1$ subsets to train the model by estimating KDE using only the other subsets. The selected subset is tested to obtain the likelihood of this subset on the model. If the likelihood is high, that means KDE with this particular bandwidth does not overfit the data points and generalizes to unseen data as we test likelihood with a separate subset from training subsets. This process is repeated for all k subsets, for multiple splits of the data and the results (likelihood) are averaged. The model which gets us a higher likelihood on the test sets is the best performing model as it means it fits the data well. We select the ideal bandwidth as the bandwidth of the best performing model [47–49]. This method has some variance in bandwidth selection as it depends on the dataset and the particular splitting but variance can be reduced with the repetitions. [50] We set k to be 5 in our experiments, with 3 repetitions.

For comparison purposes, we have also included in the experiments a version of KDE in which the bandwidth is fixed. As for kernel selection, we use Epanechnikov kernel in our implementation which is optimal in minimizing mean square error. [51]

3.2.5 Stop Criterion

As AutoFeed's main loop runs, the leakage estimation can converge if newly generated inputs no longer discover new behaviors. If the estimation stops changing, this can mean that the exploration of the input space has saturated and we may finish the analysis and print the leakage estimation ranking. To detect this condition, we check the change in information leakage of the top leaking feature and finish the analysis if it is smaller than a predetermined ϵ for a long enough period of time. Accuracy of leakage estimation depends on the accuracy of probability estimation $\hat{p}(x)$ and as number of inputs N increases, accuracy of $\hat{p}(x)$ will also increase.

In Procedure 1 we show the pseudocode for this process. Note that this pseudocode is a simplified version: in the actual AutoFeed implementation, we terminate the analysis only if leakage estimation for the top k features converges, and we assume that leakage estimation converges if it changes less than ϵ for at least n consecutive iterations (rather than the last iteration as in Procedure 1), where k and n are adjustable parameters. To simplify the presentation, in Procedure 1 we show a version where $k = 1$ and $n = 2$, but the values of k and n are adjustable in our implementation.

3.3 Experimental Evaluation and Implementation

In this section, we describe the implementation and experimental evaluation details for AutoFeed. We first experimentally evaluate AutoFeed using five example functions which have interesting input/output relationships. We also evaluate AutoFeed using software systems from the DARPA Space/Time Analysis for Cybersecurity (STAC) program [41], which are publicly available [23]. The STAC systems are multi-component systems (Web, client-server, peer-to-peer) that communicate over TCP streams encrypted with TLS/SSL, developed by DARPA to evaluate side-channel vulnerability detection techniques. The applications in our benchmark are a superset of those used in [18]. We added Railyard because we were interested in modeling its highly structured input format with the mutator-based approach.

3.3.1 Implementation

AutoFeed is written in Python. We use the trace-capturing library from [18], which relies on *scapy* [52] for packet sniffing. AutoFeed uses *scikit-learn* [53] and *numpy* [54] for probability estimation and leakage quantification, *matplotlib* [55] for plotting, and the Python *docker* [56] library for container orchestration. We ran AutoFeed on the DARPA STAC Reference Platform, which comprises three Intel NUC computers (see Section 3.3.4). Users can run AutoFeed on any number of computers and networks, including localhost. For API design, readers can refer to Section 6.1.2 which describes the API in depth for the tool, TSA.

3.3.2 Example Functions

We define five example functions which take an input and produce a single feature in order to evaluate the contributions discussed in Section 3.2.3 and 3.2.4. Examples 1–4 have the input format $(s, x, y, a, b, c, d, e, f, g, h)$ where s, the secret value, is between 1–16, and the other fields are between 1–100. Example 5 takes a list of strings as input and the secret value is the length of the list limited to a maximum length of 15. Since the secret has 16 possible values in all cases, the total amount of information that could possibly be leaked is $\log_2 16 = 4.00$ bits. Code for Examples 1–5 can be seen in Listing 3.1. Feature value of Example 1 is distributed uniformly between 0.5 and 1.0. Since there is no correlation between secret and feature, this example leaks 0 bits. In Example 2, there's a bijection between feature values and secrets. Thus, this example fully leaks 4.00 bits. Example 3 is multimodal, where the distribution changes according to value of x. When x is even, there is perfect correlation between secret and feature values. When x is odd, there is no correlation. We use Shannon entropy, an average measure of leakage, and this example leaks 2.00 out of 4.00 bits. Feature of Example 4 depends on fields x and y but those fields are not related to secret, thus this example leaks 0 bits. Feature of Example 5 depends on both number and length of list elements, thus there are some collisions. It leaks 2.21 bits of information.

Listing 3.1: Code for the example functions

```
def f1(s,x,y,a,b,c,d,e,f,g,h):
    return randomfloat(0.5,1.0)
def f2(s,x,y,a,b,c,d,e,f,g,h):
    return random(1,50)*20 + s
def f3(s,x,y,a,b,c,d,e,f,g,h):
    if x%2 == 0: return s*10
    else: return y+1000
def f4(s,x,y,a,b,c,d,e,f,g,h):
    return x+y
def f5(list1):
    return len(str(list1))
```

3.3.3 STAC Systems

Airplan (265 classes, 1,483 methods) is the airline system from our Section 3.1 example. Users can upload, edit, and analyze flight routes by metrics like cost, flight time, passenger and crew capacities. Our secret of interest is the *number of airports* in a route map uploaded by a user. Bidpal (251 classes, 2,960 methods) is a peer-to-peer system where peers buy and sell items via a single-round auction with secret bids. Users can create auctions, search auctions, and place bids. The secret of interest is the *secret bid* placed by a user. GabFeed (115 classes, 409 methods) is a Web-based forum. Users can create posts, search existing posts, and engage in chat. Our secret of interest is the *Hamming weight* (i.e., number of ones) of the server's private key. SnapBuddy (338 classes, 2,561 methods) is a Web application for image sharing. Users can upload photos from different locations, share them with their friends, and find out who is online by geographical proximity. Our secret of interest is *the location* of a user (victim). PowerBroker (315 classes, 3,445 methods) is a peer-to-peer system used by electricity companies to buy and sell power. Plants with excess power try to sell it, and plants that need power try to buy it. The secret of interest is the *value offered* by one of the plants (victim). TourPlanner (321 classes, 2,742 methods) is a client-server tour optimizer—a variation of the traveling salesman problem. Given a list of cities that the user wants to visit, it computes a tour with optimal travel costs. The secret of interest is *the set of places* that the user (victim) wants to visit. Railyard (28 classes, 60 methods) is a system to manage a train station. The station manager can build trains by adding different kinds of cars, different types and quantities of cargo, adding personnel to the train, and adding stops to the train's schedule. The secret of interest is *the set of types of cargo* that are on the train when it departs from the station.

Table 3.1: Mutators used (each line is a different dimension).

Airplan	
AddAirport, RemoveAirport	Add/remove one airport.
AddFlight, RemoveFlight	Add/remove one direct flight.
IncrDensity, DecrDensity	Increase/decrease flight density by 20%.
IncrWeight, DecrWeight	Increase/decrease one weight value by 1.
BoostWeights, DeboostWeights	Multiply/divide all weights by 10.
Railyard	
AddCar, RemoveCar	Add/remove a train car.
AddCargo, RemoveCargo	Add/remove a piece of cargo.
AddCrew, RemoveCrew	Add/remove one crew member.
AddStop, RemoveStop	Add/remove one train stop.
ChangeStops	Change all stops with new ones.
ChangeCrew	Change all crew with new ones.
GabFeed	
AddOne, RemoveOne	Add/remove one 1 to the key.
AddFive, RemoveFive	Add/remove five 1s to the key.
ShuffleOnes	Shuffle the 1s in the key.
TourPlanner	
ReplaceOneCity	Replace one city with a different one.
ShuffleCities	Shuffle the order of the five cities.
Bidpal	
IncrBid, DecrBid	Increase/decrease bid by $10.
PowerBroker	
IncrOffer, DecrOffer	Increase/decrease the offer by $10.
SnapBuddy	
PickLocation	Pick a known location from the list.

3.3.4 Experimental Setup

We used the DARPA STAC Reference Platform [18], with 3 Intel NUCs (server, client, and eavesdropper) connected by an Ethernet switch with low noise (latency: 0.22 ms min, 0.31 ms avg, 0.57 ms max). As for AutoFeed parameters, we set RPI to 20 for all programs when looking for timing side channels, and 5 for Airplan 3 and SnapBuddy as there was non-determinism in their behavior. Time budget per iteration C is set to 5 minutes. We set the histogram bin size to 1 for space side channels, and 10^{-5} for time. For KDE with fixed bandwidth, we set the bandwidth to 0.1 for space side channels, and 10^{-5} for time. For grid search, we search over 10 parameters from the aforementioned fixed bandwidth for space/time to the maximum range of the relevant feature. We also include the standard deviation bandwidth in the search. For mutation weighing, we assign weights using GetWeights, considering top-5 features. The mutators for DARPA STAC systems are described in Table 3.1. The mutators are manually written to modify the secret and various aspects of the input with the hope that some of the mutators will affect the observables. The secret of interest for each app in DARPA STAC systems is determined by DARPA. For the stop criterion, we set the value of ϵ to a 0.5% difference and checked that, for the top feature, the leakage estimation stayed within that difference for 3 consecutive iterations.

We also used two leakage quantification tools, Leakiest [20] and F-BLEAU [43], for comparison. Leakiest computes mutual information between each feature and secret using histogram and KDE assuming Gaussian distribution for quantification and hypothesis testing. F-BLEAU computes min-entropy, which provides a lower bound on Shannon entropy, using a nearest neighbor based approach on all features.

3.3.5 Experimental Results

Leakage method comparison. For the five example functions, we started with 16 seed inputs and ran 250 iterations, obtaining 100 data points per iteration. Results are shown in Figure 3.1. In Example 1, where observables are continuous and uniform, Gaussian and KDE with std.dev. bandwidth converge easily. Histogram and KDE with fixed bandwidth converge very slowly. KDE with parameter search converges to the ground truth as fast as Gaussian and KDE-StdDev. In Example 2, Gaussian and KDE-StdDev wrongly converge to zero leakage: they assume a Gaussian distribution, but this feature is multi-modal. Histogram and KDE-Fixed converge to the correct result right away thanks to small bin size and bandwidth parameters. KDE-ParamSearch initially gets the wrong result but converges to the correct one when enough data is obtained. In Example 3, because the feature distribution is bimodal, Gaussian and KDE-StdDev yield incorrect results. Histogram and KDE-Fixed converge to a value near the actual leakage, but very slowly. KDE-ParamSearch converges much faster to the correct result, unlike the other methods. In Examples 4–5, all methods perform similarly.

In all five cases, when an assumption fails, the method yields a wrong result or takes too long. Using KDE-ParamSearch, our results do not overfit the data like Histogram or KDE-Fixed, and they do not underfit like Gaussian and KDE-StdDev. With this approach, we are able to select the best bandwidth value that maximizes likelihood of data and we are able to converge to the correct leakage value.

For STAC applications, using a small set of seeds (<75), we are able to distinguish if a vulnerability is present, mitigated, or absent; weigh mutators automatically, and stop iterating when the leakage values for top features stabilize. See Table 3.2.

For Airplan, Railyard and SnapBuddy, AutoFeed converges quickly and vulnerable cases are found to leak 100%, whereas in cases where leakage is mitigated or absent, lower leakage results are found. For all cases except Railyard, the L_{KDE-PS} result is greater than the lower bound estimated by F-BLEAU. F-BLEAU estimates leakage for multi-dimensional feature vectors and it may have found a correlation between 2 features that AutoFeed is not able to detect since AutoFeed analyzes each feature separately.

For GabFeed, PowerBroker, Bidpal and TourPlanner, AutoFeed converges in 8 to 20 iterations and the leakage results for leaky versions have higher leakage than for non-leaky versions. For PowerBroker, Bidpal and GabFeed cases, especially in non-leaky cases, Histogram and KDE-Fixed overestimate the leakage. For PowerBroker 1 and TourPlanner, L_{KDE-PS} is lower than L_{Gauss} and the reason is that candidate bandwidth values have values greater than standard deviation and in these cases, a bandwidth value greater than std.dev. was selected as the ideal bandwidth, resulting in a lower leakage estimation. PowerBroker 4's results show L_{Gauss} is overestimating the leakage but std.dev. is actually 100 times lower than our fixed bandwidth, resulting in $L_{KDE-Fixed}$ overfitting on the data but the fixed methods still overestimate the leakage, reporting 100% leakage on other features.

Comparing Leakiest to KDE-ParamSearch, Leakiest sometimes underestimates the leakage (SnapBuddy) and it is unable to produce a result when the number of samples is too low (GabFeed). Leakage quantification took between 45 minutes and 5.5 hours on almost all applications and exact runtime per application can be seen on Table 3.2. Only TourPlanner takes more than a day to analyze in total. The reason is size of the secret domain of TourPlanner is much greater than other applications, at least 6 times more, and the parameter search is done to estimate $p(x|s)$ for each secret value s in the secret domain $\mathbf{S}$, making the runtime proportional with size of the secret domain.

In summary, KDE-ParamSearch, with a stop criterion, converges to a leakage value between Histogram and Gaussian, in most cases greater than the lower bound identified by F-BLEAU, and handles all data distributions automatically.

Mutator weighing comparison. To test the effectiveness of assigning weights to mutators, we ran all five examples starting from the same seed set, once with mutation weighing, once without mutation weighing, and estimated the leakage using KDE-ParamSearch. The goal is to see if selecting useful mutators gets the leakage results closer to the ground truth. Results are shown in Figure 3.2. First four cases had 62 mutators to change the secret and other variables. The fifth example has 110 mutators to change the input list: add/remove elements, shuffle characters, replace words, shuffle list, etc.

For Example 1, leakage difference between two runs is minimal because the observable value does not depend on the input. For Examples 2–5, the run with mutation weighing is able to converge faster because it gives more weight to mutators that change the parameters like s, x, y that affect the observables.

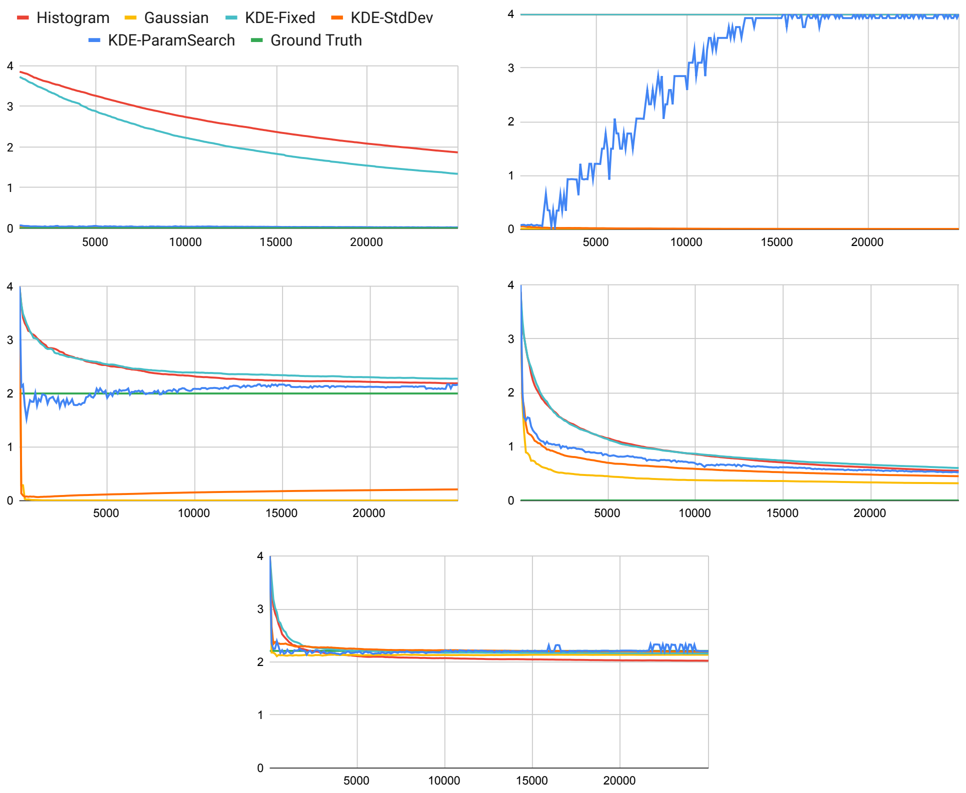


Figure 3.1: Information leakage results for Examples 1–5 using Gaussian, Histogram and KDE. X-axis shows number of data points. Y-axis shows leakage in bits. Ground truth for the Examples 1–5 are 0 bits, 4.00 bits, 2.00 bits, 0 bits and 2.21 bits respectively.

We ran a similar test on some STAC apps with complex inputs like RAILYARD and there is some difference between leakage results with and without mutator weighing (for top feature, 28% without weighing, 22% with weighing) but without the ground truth, it is impossible to evaluate if our approach improved the leakage estimation on the STAC apps.

Automated input set generation. Results in Figure 3.1 also show one of the key advantages. Consider the leakage values computed on Example 2. A tool that relies on manually constructed input sets cannot differentiate the input set with 10000 inputs from the one with size 20000. However, the leakage values for these input sets are very different. Based on its feedback-driven iterative approach, AutoFeed is able to converge to an accurate leakage estimation automatically starting from the same input set.

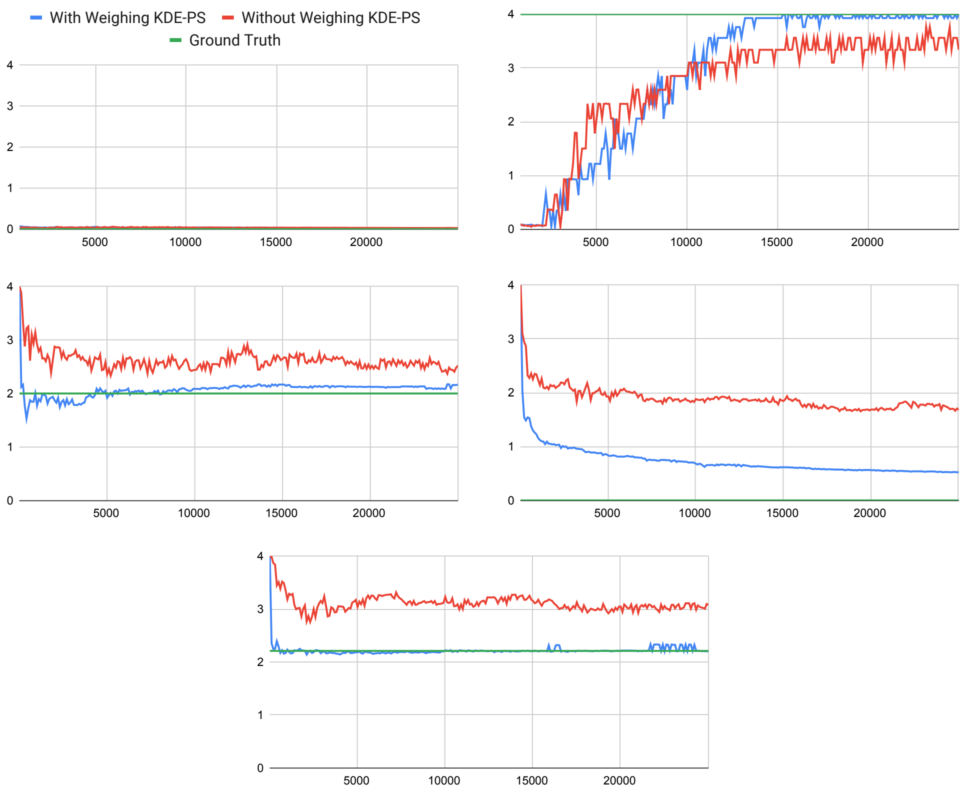


Figure 3.2: Information leakage results for Examples 1–5 with and without mutator weighing, using KDE-ParamSearch. X-axis shows number of data points. Y-axis is leakage in bits. Ground truth for the Examples 1–5 are 0 bits, 4.00 bits, 2.00 bits, 0 bits and 2.21 bits respectively.

Leakage results for different noise levels. To demonstrate that AutoFeed also produces meaningful results on a noisy network environment, we simulate the same experiments as if the servers are on three different locations. We measured the latency of three servers, one in US West Coast (Google servers, latency: 3.43 ms avg, 0.08 ms std.dev), one in US East Coast (Wikimedia servers, latency: 74.64 ms avg, 3.20 ms std.dev), and one in Russia (VK servers, latency: 220.52 ms avg, 2.38 ms std.dev). We used these latency values to add Gaussian timing noise to the obtained packets and simulate a noisy network environment. These simulations only affect the cases where we look for timing side channels. The results are in Table 3.3. We expect the leakages to drop because of extra collisions created by noisy environments. For all cases, the leakages drop when compared to the original experiments as we predicted. Some cases like

Table 3.2: Leakage results using AutoFeed with different probability estimation methods. L_{Gauss} and L_{Hist} are Gaussian-based and histogram-based estimations respectively. $L_{Leakiest}$ is Leakiest-based estimation. $L_{KDE-Fix}$, L_{KDE-SD}, L_{KDE-PS} are using KDE with a fixed bandwidth, standard deviation based bandwidth and parameter search based bandwidth respectively. $L^*_{F-BLEAU}$ is min-entropy results using the F-BLEAU tool. Top Feature is the top feature when run with L_{KDE-PS}. Runtime describes total analysis runtime in minutes.

Programs	Type	Vulnerability	Top Feature-AutoFeed	L_{Gauss}	L_{Hist}	$L_{Leakiest}$	$L^*_{F-BLEAU}$	L_{KDE-SD}	$L_{KDE-Fix}$	L_{KDE-PS}	Iter.	Runtime
AIRPLAN 2	Space	Present	Σ Sizes Phase 4 $\downarrow$	100%	100%	99%	90%	100%	100%	100%	3	76 min.
AIRPLAN 5	Space	Mitigated	Σ Sizes Phase 4 $\downarrow$	89%	94%	74%	82%	88%	93%	89%	4	114 min.
AIRPLAN 3	Space	Absent	Size Pkt 20 $\downarrow$	46%	33%	21%	20%	45%	35%	47%	6	161 min.
RAILYARD	Space	Absent	Size Pkt 2 $\updownarrow$	22%	27%	21%	27%	20%	22%	22%	12	202 min.
SNAPBUDDY	Space	Present	Σ Sizes Full Trace $\uparrow$	100%	100%	47%	100%	100%	100%	100%	3	47 min.
GABFEED 1	Time	Present	Δ Pkt 12-13 $\updownarrow$	99%	100%	N/A	60%	100%	100%	98%	8	108 min.
GABFEED 2	Time	Absent	Δ Pkt 11-12 $\updownarrow$	29%	71%	N/A	24%	31%	66%	31%	19	297 min.
GABFEED 5	Time	Absent	Δ Pkt 11-12 $\updownarrow$	29%	65%	N/A	21%	31%	61%	32%	15	240 min.
POWERBROKER 1	Time	Present	Δ Pkt 9-10 $\uparrow$	43%	100%	42%	53%	45%	100%	39%	20	313 min.
POWERBROKER 2	Time	Absent	Δ Pkt 43-44 $\updownarrow$	11%	32%	3%	18%	10%	24%	15%	18	263 min.
POWERBROKER 4	Time	Absent	Δ Pkt 28-29 $\updownarrow$	22%	9%	9%	32%	26%	9%	25%	16	220 min.
BIDPAL 2	Time	Present	Δ Pkt 28-29 $\updownarrow$	22%	100%	33%	32%	23%	100%	23%	17	217 min.
BIDPAL 1	Time	Absent	Δ Pkt 35-36 $\updownarrow$	2%	39%	3%	15%	3%	35%	14%	10	137 min.
TOURPLANNER	Time	Present	Δ Pkt 12-13 $\updownarrow$	60%	70%	62%	42%	62%	67%	60%	12	2057 min.

Table 3.3: Leakage results using L_{KDE-PS} for four different noise conditions.

Programs	Type	Vulnerability	Top Feature-AutoFeed	STAC Platform	US-West	US-East	Russia
GABFEED 1	Time	Present	Δ Pkt 12-13 $\updownarrow$	98%	97%	96%	96%
GABFEED 2	Time	Absent	Δ Pkt 11-12 $\updownarrow$	31%	29%	9%	8%
GABFEED 5	Time	Absent	Δ Pkt 11-12 $\updownarrow$	32%	23%	8%	7%
POWERBROKER 1	Time	Present	Δ Pkt 9-10 $\uparrow$	39%	39%	37%	36%
POWERBROKER 2	Time	Absent	Δ Pkt 43-44 $\updownarrow$	15%	14%	3%	3%
POWERBROKER 4	Time	Absent	Δ Pkt 28-29 $\updownarrow$	25%	20%	9%	8%
BIDPAL 2	Time	Present	Δ Pkt 28-29 $\updownarrow$	23%	23%	22%	23%
BIDPAL 1	Time	Absent	Δ Pkt 35-36 $\updownarrow$	14%	9%	8%	3%
TOURPLANNER	Time	Present	Δ Pkt 12-13 $\updownarrow$	60%	50%	5%	5%

GABFEED 1 are affected less than others. We believe this is because there is not much overlap between the distributions and the separation is greater than the level of noise.

3.4 Chapter Summary

In this chapter, we presented AutoFeed, a black-box tool to detect and quantify side-channel information leakage in networked software systems. AutoFeed significantly reduces the manual effort required by prior black-box side-channel analysis approaches by providing a feedback-driven automated process for input space exploration and information leakage estimation. Given a set of input mutators and a small number of seed inputs, AutoFeed iteratively mutates inputs and periodically updates its leakage estimations to identify the features that leak most information about the secret. AutoFeed measures the effect of different mutator subsets on leakage, and assigns weights to prioritize mutators that produce more changes in the leakage estimation. AutoFeed uses kernel density estimation and an automated search mechanism to determine crucial hyperparameter values, in order to estimate probability distributions for modeling the observed data. It uses a stop criterion to detect convergence of the leakage estimation and terminate the analysis. The experimental evaluation on the benchmarks shows that AutoFeed is effective in automatically detecting and quantifying information leaks.

Chapter 4

Estimating Bounds for Information Leakage

When analyzing side-channel information leakages on network traces, accurate measurement of the information leakage is important in determining the impact of the network side-channel on the user privacy. In previous chapters, our measurements on the information leakage were computed on single features to answer questions such as "Does the size of 1st packet correlate with the secret information?". However, there may be cases where that measurement may not reveal the total information amount. Let's say we have a case where each packet does not reveal any information but the total of 1st and 3rd packets correlate with the secret value. In that case, measuring information leakages accurately requires examining combined features.

Computing mutual information over multiple features brings its own challenges. One of the challenges is estimating the probability distribution of the combined feature set. This becomes very difficult as the features also correlate within each other. If the features are independent, that makes the estimation easier but that does not hold as the network traffic features also correlate within each other. Another challenge is that increasing the number of features increases the time complexity of the computation exponentially. One solution is an approximate method that computes the information leakage in a timely manner. To find this approximate method, we present various techniques that provide information leakage bounds using feature reduction and critic neural networks.

One other issue is demonstrating the information leakage. In the wild, the attacker observes a network trace, uses the previously computed side-channel analysis results and examines the feature distribution to find the secret information. This attacker works exactly like a classifier and we can demonstrate the attacker's working conditions by training a classifier which takes a network trace and returns the class value (secret information. This classifier learns the relation between the features and the secret information and it can be used to demonstrate the information leakage as a working example that takes a network trace. Performance of the trained classifiers depend on various parameters such as classifier architecture, batch sizes, number of features selected, etc. To find the maximum performance we can achieve using classifiers, we demonstrate methods to find upper bounds on the classifier performance using Bayes' error rate.

In this chapter, we are going to describe methods to measure information leakage with multiple features, describe how to compute information leakage bounds and describe how to demonstrate leakage by using classifiers. In Section 4.1, we describe the challenges in extending the previous work to multiple features and explore various methods to find the bounds on the amount of information over multiple features. In Section 4.2, we describe how we demonstrate the side-channel attack using classifiers. We also describe our explorations on how to compute the maximum possible classifier performance using Bayes' error. In Section 4.3, we describe our experimental evaluation and results on computing the information leakage over multiple dimensions, and training and testing classifiers. In Section 4.4, we conclude and summarize the chapter.

4.1 Measuring Information over Multiple Features

As we described in the previous section, when analyzing network traces, we need to estimate the information leakage over multiple features to improve our leakage estimate. In the previous sections, we have performed this estimation over single features which let us obtain an initial estimate. This estimation may not reflect the total amount of information leakage, especially in the case where combinations of features leak information. For example, if the secret information is sum of two integer feature values, then each feature by itself will not be correlated with the secret information but when combined, there will be full leakage. In the following parts, we describe how to compute information leakage and its bounds over multiple features.

4.1.1 Statistical Estimation of Mutual Information over Multiple Features

As described in system model in Section 2.2, our feature functions can be applied to a trace list T to obtain a feature value list $F_i = \langle f_i(t_{(1)}), f_i(t_{(2)}), \ldots, f_i(t_{(|T|)})\rangle$. If we want to combine these features to have n dimensional features instead of single features, we can have an n dimensional feature value list $F_{1,2,\ldots,n} = \langle (f_1(t_{(1)}), f_2(t_{(1)}), \ldots, f_n(t_{(1)})), \ldots, (f_1(t_{(|T|)}), f_2(t_{(|T|)}), \ldots, f_n(t_{(|T|)}))\rangle$. Each list also has a corresponding secret value list $s = \langle s_1, s_2, \ldots, s_{|T|}\rangle$. Extending the mutual information computation described in the previous chapters, we can compute the mutual information between the n dimensional feature set $\mathbf{F_{1,2,\ldots,n}}$ and the secret value set $\mathbf{S}$. To summarize, we mapped $f_i(t)$ to x_i instead not to mention the trace every time. Given these information, the mutual information between the n dimensional feature set $\mathbf{F_{1,2,\ldots,n}}$ and the secret value set $\mathbf{S}$ is shown below.

$$\mathcal{H}(S|F_{1,2,\ldots,n}) = -\sum_{\langle x_1,\ldots,x_n\rangle \in \mathbb{F}_{1,2,\ldots,n}} p(x_1,\ldots,x_n) \sum_{s\in\mathbb{S}} p(s|x_1,\ldots,x_n)\log_2 p(s|x_1,\ldots,x_n) \quad (4.1)$$

Computing the formula of conditional entropy for multiple dimensions requires estimating $p(x_1, \ldots, x_n)$ and $p(s|x_1, \ldots, x_n)$. Estimating these distributions may be difficult as we would need to have a large amount of samples for different combinations of values for each dimension which we may not have. Increasing the number of dimensions increases the amount of data needed exponentially in the general case as the variable space becomes sparse. [57]. Computing the sum over the multiple dimensions increases the complexity as well. With N dimensions, the time complexity of the computation is $O(k^N)$ which makes the estimation infeasible as we increase the number of features.

One naive solution to solve the probability estimation problem is to assume the features are independent. This makes the problem easier to solve where instead of estimating this N dimensional distribution, we can compute the distributions for each dimension separately and combine them based on Bayes' rule. If the random variables x_i and x_j are independent, probability estimation becomes a much easier problem as $p(x_i, x_j) = p(x_i)p(x_j)$. For conditional distributions such as $p(x_i, x_j|s)$, it can be separated as $p(x_i|s)p(x_j|s)$. This assumption does not hold on the network trace data, therefore we would need to obtain estimates using another method. Independent features can be obtained using feature reduction techniques such as Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) but feature reduction can reduce the correlation between trace features and secret information as well. To address the issue of estimation, we describe how to estimate the probability distribution using neural networks in the following subsection.

4.1.2 Neural Estimation of Mutual Information

Another way to compute the mutual information is to use neural networks and train them on the captured trace data to measure the correlation between features and the secret value. There have been various prior works which estimate the mutual information over complex data for improving machine learning models [58–60]. These methods provide a lower bound to the information leakage using critic networks. To compute the bounds on mutual information, we train a critic network $N(x, y)$ that takes a feature vector x and the secret value y and returns a score based on their correlation. The resulting critic network $N(x, y)$ can be used to replace the conditional distribution $p(y|x)$, therefore we are ultimately trying to learn $p(y|x)$ using a neural network to compute the entropy.

For training, we start with an initial network N with random edge weights. After each training step with a batch of input output pairs, we compute the mutual information lower bounds which we will describe in the next section. The resulting mutual information is used as a negative loss function to train the next set of parameters and prior work shows that minimizing mutual information loss maximizes the lower bound value. [58] In the following paragraphs, we describe the lower bound computation methods we used over network traces.

Interpolated Lower Bounds

Here, we define the mutual information compactly as the expected value of the log difference between the conditional distribution $p(y|x)$ between the secret value y and feature vector x and the secret distribution $p(y)$ for the secret value y. The equation below describes the expected value definition and how it can be converted to our previous definition.

$$\begin{aligned} I(X;Y) &= \mathbb{E}_{p(x,y)}\left[\log_2 \frac{p(y|x)}{p(y)}\right] = \mathbb{E}_{p(x,y)}\left[\log_2 p(y|x) - \log_2 p(y)\right] \\ &= \left(-\mathbb{E}_{p(x,y)}\left[\log_2 p(y)\right]\right) - \left(-\mathbb{E}_{p(x,y)}\left[\log_2 p(y|x)\right]\right) \end{aligned} \tag{4.2}$$

This equation requires the conditional distribution $p(y|x)$ and as we mentioned before, we plan to substitute the conditional distribution $p(y|x)$ with a critic network $N(x, y)$ to estimate a lower bound on the information leakage. Assuming that we do not know the distribution $p(y|x)$, we can use the training dataset to train $N(x, y)$ to learn the substitution for $p(y|x)$. We also assume that we can train the critic network N successfully. One difficulty with this estimation is that it is highly dependent on the critic network's performance. The parameters such as input and hidden dimension size, batch size, number of iterations for training, etc. all affect the results. In our experiments, we explore different parameter values to obtain a well-trained critic network. Using these assumptions, we use various different bound estimation methods to estimate the lower bound of the information leakage such as InfoNCE, NWJ, and interpolated lower bounds [60].

The first lower bound, I_{InfoNCE}, uses the samples and the network to compute the average information between samples x_i, y_i, comparing them to secret values of other samples y_j. The bottom part of fraction approximates $p(y)$ while upper part approximates $p(y|x)$ using $N(x, y)$. As we use samples from the trace dataset, this requires a large and varied sample over the dataset to measure more accurate values. This is a low variance, high bias estimate and it may underfit the underlying distribution in times. [58]

$$I_{\text{InfoNCE}}(X;Y) = \mathbb{E}\left[\frac{1}{K}\sum_{i=1}^{K}\log_2 \frac{2^{N(x_i,y_i)}}{\frac{1}{K}\sum_{j=1}^{K} 2^{N(x_i,y_j)}}\right] \tag{4.3}$$

The second lower bound, I_α, is a modified version of the previous lower bound technique. This bound is developed so that the parameter α sets a tradeoff between two estimations of $p(y)$ and it can be set to different values to achieve a balance between the two estimations. [60].

$$\begin{aligned} I_\alpha(X;Y) &= \mathbb{E}\left[\frac{1}{K}\sum_{i=1}^{K}\log_2 \frac{2^{N(x_i,y_i)}}{\alpha m(y; x_{1:K}) + (1-\alpha)q(y)}\right] \\ m(y; x_{1:K}) &= \frac{1}{K}\sum_{j=1}^{K} 2^{N(x_i,y_j)} \end{aligned} \tag{4.4}$$

One difficulty with this estimation is that it is highly dependent on the critic network's performance. The parameters such as input and hidden dimension size, batch size, number of iterations for training, etc. all affect the results. To address this issue, we explore different parameter values to obtain the best performing result in our experimental evaluation.

4.2 Training Classifiers and Bounding Classifier Performance

The previous methods can provide information leakage estimations to find the amount of information leaked but these estimations are based on assumptions. As we mentioned, most of our information leakage computation is on single features which may not reflect the full amount of leakage. Our mutual information estimation depends on being able to capture the probability distribution of features using Gaussian estimation or KDE which may not hold. Our multi-dimensional information estimation depends on random sampling. Mutual information bounds using neural networks depend on the parameters and performance of the critic network. If the network cannot learn the correlation, we have no lower bound other than min-entropy.

Instead of providing a number to represent the correlation between inputs x and secret value y, we can use machine learning techniques to learn the correlation with classifiers. The aim of using classifiers is to have a lightweight way of predicting user actions (classes) by observing traces where the classifier algorithms are finding patterns and correlations between features and classes. Using a set of the feature vector x and the secret information y pairs, we can train a classifier N that learns the the correlation between these pairs and provides function, $N(x) = y$ which takes the feature vector x and returns the related secret. After training, we can use a trained classifier N which takes a feature vector x and returns the predicted action $y \in \mathbf{Y}$.

4.2.1 Classifiers

There are various classifier models that try to learn the relation between the features extracted from the trace and the secret information. In this section, we describe the various classifiers and their training procedures which we used in our work.

Random Forests

Random forest is a classifier that learns the classification tasks using a set of learners and aggregates the result. A set of decision trees are trained using the dataset, and the majority result is used as the result of this classifier. Decision trees are trees where each node is a condition on a feature, and branches denote the true or false condition. The leaf nodes are classes, so for each input, a path can be extracted, and the class in the leaf node of the extracted path will be the prediction for that input.

For training, n decision trees are trained separately with different techniques to modify the training dataset for diversity in the resulting decision trees. There are multiple ways to train decision trees such as ID3 [61] and C4.5 [62] among others. We used the CART algorithm to train the decision trees as it is a more general form of decision trees while being similar to the publicly available state-of-the-art C4.5 algorithm with small differences for generalization. A simplified algorithm can be seen in Procedure 4 [62–64].

The advantages of the random forests algorithm are its explainability, feature selection, and avoidance of overfitting to the training data. We can use the decision conditions in decision trees to point out which packets and features are leaking information. It also has inbuilt feature selection where features leaking more information will be used for decision rules. With the majority voting, it prevents a single classifier dominating the results and overfitting to the training data.

To train each tree, a technique called bootstrapping is used where for each decision tree, the training samples are randomly selected with replacement over the original dataset. Then a random feature is selected, and a condition over the selected feature is generated so that the condition splits each class into its own leaf in the end. The condition is selected using a heuristic called Gini impurity which is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in

Procedure 4 CART(X, $\vec{y}$) Given the list of inputs $X = \vec{x}_{(1)}, \vec{x}_{(2)}, ...$ where $\vec{x}_{(i)}$ is a vector $(x_1, x_2, ..., x_n)_{(i)}$ with size n and the corresponding labels $\vec{y}$, CART returns a binary decision tree.

```
1: if $\forall i, j : \vec{y}_i = \vec{y}_j$ then
2:     return LeafNode($\vec{y}_i$)    ▷ If all inputs are of the same class, create a leaf node denoting that class
3: for $i = 1...n$ do    ▷ For each feature dimension
4:     $(I_i, p_i) \leftarrow$ ComputeGini($X, \vec{y}, i$)    ▷ Compute the Gini impurity for splitting which shows the probability of guessing error of inputs for a split.
5: $i^* \leftarrow \arg\min_i I_i$
6: $p^* \leftarrow p_{i^*}$    ▷ Select the split with the lowest Gini impurity
7: $N \leftarrow$ Node($p^*$)    ▷ Create node with the split for the tree.
8: $N.right \leftarrow$ CART($X[p^*], \vec{y}[p^*]$)
9: $N.left \leftarrow$ CART($X[\neg p^*], \vec{y}[\neg p^*]$)    ▷ Create left and right child nodes for this node using the dataset partitions where $p^*$ is true or false
10: return $N$
```

the subset.

k-Nearest Neighbors

k-nearest neighbor algorithm is a fairly simple classifier where the class of the input is determined only by the class of inputs from the training data that are near it. To predict the class for an input vector $\vec{x}$, the algorithm finds k inputs in the training set that are nearest to the input using the L_2 norm as a distance metric. The predicted class for the input is the majority class of the k nearest inputs. No training procedure is needed as the algorithm just uses the training set to predict inputs [65]. The advantages of k-NN is its low training time. It should perform well in cases where the number of features is lower because as the number of features increase, the feature space becomes sparser and the distance between data points increases.

Fully connected deep neural networks

Deep neural networks consist of multiple consecutive layers of neurons, where each neuron is a node containing a floating point number and each neuron in a layer is connected to the neurons in the next layer with a weighted edge. The neuron value in the next layer is the weighted sum of the all the connected neuron values in the previous layer passed through an activation function [66].

Deep neural network architectures consist of an input layer, a sequence of hidden layers, and an output layer. The input layer denotes the feature vector we extracted from the network traces. It is connected to a sequence of hidden layers, each connected to the next layer, and ends with the output layer where each neuron corresponds to a class which is the user action or device state. These correspond to the secret values we are trying to predict. The prediction for a particular input is the class which has the highest value of corresponding neuron.

The fully connected neural network can be defined as below where feature vector $\vec{x} = (x_1, x_2, ...)$ is passed through N connected layers, with matrix multiplications and activation functions applied at each layer as a function with input vector as the input where consecutive matrix multiplications and activation functions are applied alternately as

$$\begin{aligned}
h_1 &= f_1(W_1\vec{x} + b_1) \\
h_n &= f_n(W_n h_{n-1} + b_n) \\
y &= \arg\max f_N(W_N h_{N-1} + b_N) \\
F(\vec{x}) &= \arg\max f_n(W_n \dots f_2(W_2 \cdot f_1(W_1 \cdot \vec{x})))
\end{aligned}$$

where h_i represents the ith hidden layer, y represents the output class, W_i represents the edge weights between the $i-1$th and ith layers, b_i represents the biases to be added to the ith layer, and f_i represents the activation function, which is rectified linear unit (ReLU) for $i = 1, 2 \dots N-1$ and logistic sigmoid for $i = N$ [67, 68].

The weighted edges and neuron biases of a network are typically initialized randomly. To train a neural network, labeled input data must be passed through the layers as described. Then, the weights and biases are adjusted to minimize the prediction error, and ideally the neural network improves its predictions with training iteratively.

Neural networks are very flexible and can be applied to a wide variety of tasks. Their shortcomings are that they requires a large amount of dataset as they learn iteratively and they can overfit or underfit to the training set, skewing the accuracy results.

4.2.2 Bounding Classifier Accuracy using Bayes Error Rate

Classifier's ability to learn the correlations depends on the parameters of the classifier. If a classifier can guess the secret 75% of the time, we do not know if the classifier can be improved. To provide some context to the accuracy results, we compute an upper bound to classifier accuracy. We use Bayes error rate which gives the lowest possible error rate for data distributions to compute the accuracy bounds as described in the theorem below [69]. As we explained in the previous sections, obtaining the conditional distribution and computing the entropy is difficult in multiple dimensions. Therefore we describe a theorem on accuracy bounds on single features which can be computed easier.

Theorem 1 *In a classification task with N classes assuming prior class distribution $p(y)$ is known for all values $y \in \mathbf{Y}$, assuming a Bayes classifier F over a single feature x such that C_i represents the domain for x when $p(x \mid y = i)$ is greater than $p(x \mid y = j)$ for all values of j not equal to i, the accuracy bound of this classifier can be defined as*

$$Acc_{Bound}(F) = \sum_{i}^{N} p(y = i) \int_{x \in C_i} p(x|y = i)\, \mathrm{d}x$$

Proof: To obtain the accuracy bound, we need to find the percentage of true positive data points for all classes. $\int_{C_i} p(x|y = i)\, \mathrm{d}x$ describes the cumulative probability of secret value $i \in \mathbf{Y}$ that is being classified correctly as $p(x|y = i) > p(x|y = j)$ for any value of j not equal to i when $x \in C_i$. Multiplication of this cumulative probability with the prior class probability $p(y = i)$ gives us the percentage of inputs with class i over all possible inputs that are correctly classified. When computed over all classes, this gives us the percentage of inputs classified correctly in total. ■

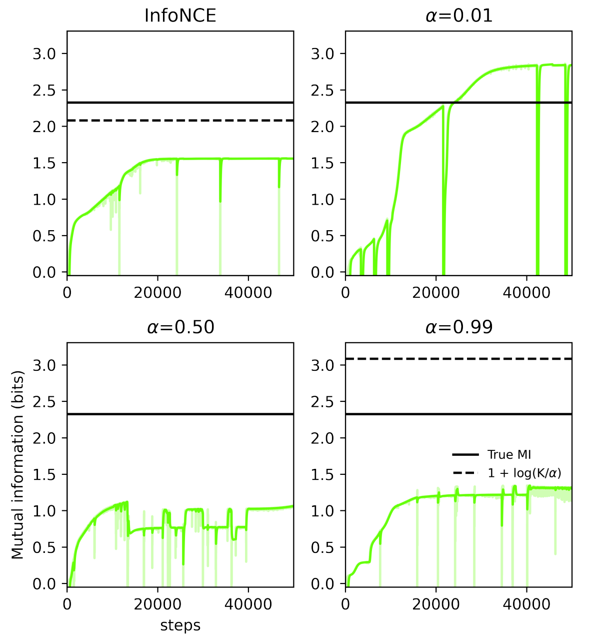
This theorem assumes that we have the actual $p(s)$ and $p(x|s)$ distributions. We assume $p(s)$ to be uniform which is also reflected in our datasets as we try to capture equal amount of data points for each classes. For $p(x|s)$, we estimate the distribution using KDE as described before. If KDE does not fully capture the actual distribution, then the upper bound may not hold.

Table 4.1: Information leakage results of IoT applications using various quantification methods in terms of bits. $|\mathbf{Y}|$ denotes secret domain size for that application. CC denotes the channel capacity, the upper bound of information leakage in bits. $I_{\textit{SF-KDE}}$ denotes the leakage of the most leaking feature using kernel density estimation (KDE). $I_{\textit{InfoNCE}}$, $I_{\alpha=0.01}$, $I_{\alpha=0.50}$, $I_{\alpha=0.99}$ denote the neural mutual information bounds computed for various methods.

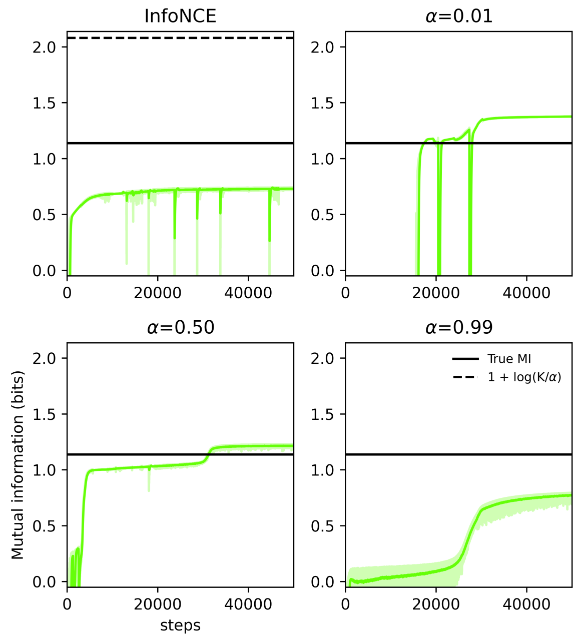
Application	$\lvert\mathbf{Y}\rvert$	CC	$I_{\textit{SF-KDE}}$	$I_{\textit{InfoNCE}}$	$I_{\alpha=0.01}$	$I_{\alpha=0.50}$	$I_{\alpha=0.99}$
grpc_stove_device	2	1.00	**100%**	2%	50%	22%	49%
grpc_stove_server	3	1.58	94%	38%	82%	44%	25%
grpc_ac_device	5	2.32	**83%**	1%	24%	1%	13%
grpc_ac_server	6	2.58	**45%**	23%	35%	12%	22%
grpc_cctv	2	1.00	**96%**	60%	62%	60%	60%
awsiot_stove_client	4	2.00	**91%**	50%	45%	25%	47%
awsiot_stove_device	4	2.00	**100%**	70%	53%	50%	35%
awsiot_ac_client	6	2.58	99%	39%	**100%**	54%	52%
awsiot_ac_device	6	2.58	84%	60%	**100%**	50%	39%
awsiot_lock_client	3	1.58	71%	37%	**80%**	73%	38%
awsiot_lock_device	3	1.58	**82%**	51%	63%	63%	50%

4.3 Experimental Evaluation

To experimentally evaluate our approach, we used previously obtained IoT trace datasets. On these datasets, we computed our leakage estimate on single features, multiple features using feature reduction, and leakage lower bound using neural networks. We also trained classifiers and computed classifier accuracy bounds over the extracted features. In cases where we used classifiers, we split the datasets to training, validation and testing datasets where we split the original dataset to subsets with size 70%, 10% and 20% respectively. We use the validation dataset to find the classifier parameters that maximize the validation accuracy. We use classifiers with the best parameters to obtain the results on test accuracy and repeat the experiment 5 times with different splits to average the results.



(a) Results for AWSIoT AC (Device-side traffic).



(b) Results for AWS IoT Lock (Client-side traffic).

Figure 4.1: Images describe the leakage lower bound over multiple features compared against highest leakage for a single feature. Best single dimensional leakage result is shown in black line and results of various leakage lower bound estimations are shown in green line. Dashed line represents the maximum amount of leakage that can be estimated with this lower bound technique.

4.3.1 Evaluating Information Leakage Estimation and Lower Bounds

Table 4.1 describes the results on IoT benchmarks for highest single feature leakage estimation, multi-feature information leakage lower bound results for 4 bound estimation methods. Combining features can only increase the overall leakage, therefore we can compare our lower bounds generated using neural networks against our best single feature leakages. If the neural leakage estimation provides a tighter lower bound, then it helps us estimate the total information leakage better than the single feature version. By observing the results, we can see that neural information bound generation achieves a tighter bound in only 3 out of 11 applications. Comparing neural estimates to single feature estimations, neural information leakage methods mostly converge to a lower value which could depend on the data distribution, architectures, etc. We explored various parameters to train these networks extensively but this method only seems to help in some cases, not all. We believe in those cases, neural information estimation is useful as it can provide tighter information leakage bounds.

Figure 4.1 shows two examples where neural leakage estimation performs better than the single feature lower bound. With 20000 steps, $I_{\alpha=0.01}$ converges to a higher lower bound estimate in these examples but the other approaches mostly stay below the estimation. Comparing the neural information leakage methods among themselves, $I_{\alpha=0.01}$ seems to be the estimator that achieves the highest information leakage result as seen in the Figure 4.1 and Table 4.1.

4.3.2 Evaluating Classifiers and Classifier Accuracy Upper Bounds

Table 4.2 describes the results of classifiers over the IoT benchmark application. Random Forest classifier performs the best overall compared to other classifiers based on its feature selection algorithm and it can be used to demonstrate the leakage when it is performing over the random guessing accuracy. For bounding the accuracy, we compare the accuracy upper bounds with a single feature classifier as we can only obtain $p(y|x)$ over single features, and it would not be fair to compare our single feature upper bound against the multi feature classifiers. On the bound results, in 13 out of 17 applications, the estimated accuracy bound holds and it is equal or higher than the actual accuracy. If it does not hold, this could be because KDE does not fully reflect the distribution. One assumption we make is a continuous distribution but if the leaking feature is based on packet sizes, the feature distribution is inherently discrete. Our method could estimate a lower leakage upper bound based on the fact that the classifier cannot correctly classify packets with size 10.5. That case will not occur for that feature as the packets

Table 4.2: Testing accuracy results of IoT applications over multiple classifiers. $|\mathbf{Y}|$ denotes secret domain size for that application. Acc_{Random}, Acc_{FcNN}, $Acc_{k\text{-}NN}$, Acc_{RF} denote the testing accuracy using random guessing, fully connected neural networks, k-nearest neighbor and random forest classifiers. Acc_{KDE} and Acc_{Bound} denote the highest testing accuracy with a single feature using KDE as a classifier and highest accuracy bound over single feature distributions. The highest value between Acc_{NB} and Acc_{Bound} estimations are bolded to show if the upper bound holds.

Application	$\|\mathbf{Y}\|$	Acc_{Random}	Acc_{FcNN}	$Acc_{k\text{-}NN}$	Acc_{RF}	Acc_{KDE}	Acc_{Bound}
grpc_stove_device	2	50%	68%	63%	80%	60%	**62%**
grpc_stove_server	3	33%	54%	55%	84%	55%	**99%**
grpc_switch	3	33%	99%	100%	100%	**99%**	**99%**
grpc_ac_device	5	20%	38%	36%	44%	35%	**48%**
grpc_ac_server	6	16%	28%	28%	30%	28%	**50%**
grpc_cctv	2	50%	99%	100%	100%	**100%**	**100%**
awsiot_stove_client	4	25%	100%	100%	100%	75%	**80%**
awsiot_stove_device	4	25%	85%	99%	97%	**100%**	99%
awsiot_ac_client	6	16%	79%	43%	83%	**66%**	47%
awsiot_ac_device	6	16%	84%	84%	84%	**74%**	65%
awsiot_lock_client	3	33%	46%	48%	72%	69%	**94%**
awsiot_lock_device	3	33%	81%	79%	84%	47%	**86%**
myq_device	3	33%	59%	57%	60%	53%	**80%**
myq_client	3	33%	83%	82%	100%	**100%**	92%
camera	3	33%	100%	99%	100%	**100%**	**100%**
motion	2	50%	94%	86%	94%	93%	**98%**
light	3	33%	77%	71%	83%	66%	**72%**

have discrete sizes, which makes that finding useless.

4.4 Chapter Summary

In this chapter, we explored various methods to extend our previous information leakage estimation using multi feature estimation and neural leakage estimation methods. We also described how we can train classifiers on the network traces and bound the accuracy for an expected amount of performance. During our experimental evaluation, we observed that neural leakage estimation methods provide tighter bounds only in a few cases. Classifier accuracy bound holds in almost all cases and as classifiers depend on many parameters, an upper bound on performance means we can measure the classifier accuracy against the upper bound on accuracy.

Chapter 5

Targeted Black-Box Side-Channel Mitigation

Internet of Things (IoT) devices are becoming more popular with increasing internet connectivity and bandwidth and they allow users to control or get information from sensors or appliances such as motion sensors, smart locks, and smart lights via smartphone apps. Although IoT devices present many benefits, they also carry the risk of being vulnerable to malicious actors. Security and privacy of IoT devices and applications are a timely and critical research problem because of vulnerabilities directly affecting end-users and their households [70, 71]. Even with encryption, eavesdroppers can monitor the network traffic and use the metadata of the traffic, such as the amount of bytes transmitted or duration of transmission, to leak information about user actions. For example, a sleep sensor that sends more packets when the user is awake reveals the sleeping habits of its user [13]. These types of information leaks due to non-functional characteristics of computer systems are called side-channels, and in this paper, we investigate mitigating the side-channel vulnerabilities in IoT applications due to network traffic.

If information leakage is detected in an IoT application, measures such as padding the packets with extra bytes, delaying packets and injecting extra packets can be used to obfuscate the relation between the network traffic and device or user activity which in turn reduces the information leakage. Too much padding or delays degrade the performance of the system and increase the power consumption, therefore a trade-off must be achieved to balance privacy and usability of the system.

In this chapter, we propose a black-box side channel analysis tool called Shark! to mitigate side-channel vulnerabilities in IoT applications. Shark! works by collecting encrypted network traces and labeling them with the secret value of the corresponding trace (which can be user actions or a device state such as a motion sensor's status at the time of the trace capture). Shark! analyzes the traces by extracting features such as packet sizes and timings, and quantifies the information leakage. Using the information leakage quantification, Shark! prioritizes which features to target and iteratively develop a mitigation strategy with respect to a tunable objective function balancing the information leakage reduction and overhead. The tunable objective function can be customized by the user by changing the parameter that controls the trade-off between information leakage and mitigation overhead. This enables Shark! to synthesize a set of Pareto optimal [22] mitigation strategies corresponding to different trade-offs between privacy and performance.

Compared to prior work on network side-channel mitigation approaches [72–75], we present the following novel contributions in this paper:

1. A method to prioritize the features for mitigation using metrics measuring information leakage, targeting the packets related with features that leak the most information (Section 5.3).

2. A search-based, tunable side-channel vulnerability mitigation method which finds the optimal packet padding and delaying strategy for mitigating both space and time network side-channels while keeping the overhead low, with the ability to adjust the trade-off between the leakage and the mitigation overhead (Section 5.4).

3. Implementation and experimental evaluation of the novel targeted black-box network side-channel mitigation approach we present in this paper (Section 5.5).

Our experimental evaluation of Shark! on three IoT benchmarks demonstrates that our approach overall performs better than the prior work in terms of reducing information leakage and overhead. Our evaluation shows that the approach can obtain a Pareto optimal mitigation strategy set where the user can select the strategy that fits their leakage and overhead constraints.

The rest of the chapter is organized as follows. In Section 5.1, we describe a motivating example, give the overview of our approach and our assumptions on attacker behavior. In Section 5.2, we give an overview of the network structure and protocols for the IoT applications and describe our system model. In Section 5.3, we go over the feature extraction and prioritization methods. In Section 5.4, we go over the targeted mitigation technique. In Section 5.5, we discuss the implementation details, IoT benchmarks, experimental evaluation of our approach, and its limitations. In Section 5.6, we conclude the paper.

5.1 Motivation & Overview

A Smart Garage Door Application.

We analyzed an IoT device produced by Chamberlain Group which manufactures various brands of garage doors and access control systems, and provides an application to control the systems called MyQ. The communication protocol between MyQ and the backend server is via a tailored HTTPS that supports the publish-subscribe model. To better demonstrate the root cause of the issue, we installed the MyQ application with version number 4.147 on a jailbroken iOS device, and we leveraged SSL Kill Switch [76] to bypass SSL certification pinning and utilize mitmproxy [77] to capture the original before-encrypted packets for analysis. To open or close the garage door, the MyQ application sends a TCP packet to the server with the payload `{"action_type":"open"}` or `{"action_type":"close"}` respectively.

While the content of HTTP headers, method, and URL may vary in each request, we do not observe any changes in their sizes. Yet, for two different actions, the size of the HTTP body is distinguishable. There exists a one byte difference between packets containing the strings "open" and "close". Even when the packets are encrypted, someone who is eavesdropping on the network traffic can infer when a user is opening or closing the garage door. These types of network side-channel information leakages are pervasive in IoT systems and have been used in prior work to identify devices and user actions [13, 78, 79].

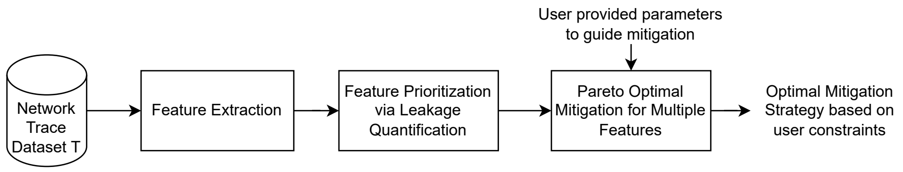

User provided parameters to guide mitigation
Network Trace Dataset T
Feature Extraction
Feature Prioritization via Leakage Quantification
Pareto Optimal Mitigation for Multiple Features
Optimal Mitigation Strategy based on user constraints

Figure 5.1: Shark! workflow describing the main building blocks of our approach.

Our Leakage Analysis and Targeted Mitigation Approach.

Shark! automatically discovers the information leakage in IoT applications like the one described above and mitigates them by synthesizing a packet padding and delaying strategy based on the most leaking features while minimizing the cost of mitigation. Figure 5.1 illustrates the workflow and the steps of our approach.

For example, to investigate the garage door application discussed above, one can use Shark! to listen to the communications and collect a set of traces where the user opens the door and another set of traces where the user closes the door. Shark! has packet capture capabilities to label each captured trace with the corresponding action (the secret information that we would like to protect) and create a trace dataset for analysis.

Shark! uses the labeled trace dataset to extract common features among all traces. It extracts aggregate features such as total size, total duration, the average size of packets in the trace, etc., and packet-level features such as the first packet size, the time difference between third and fourth packet, etc. To detect and quantify side-channel leakages, it models the distribution of each feature per secret and calculates the mutual information between the feature and secret using the distributions which reflect how many bits of information was learned by observing each feature. Shark! ranks the results according to the amount of information leaked and provides a ranked list of features and the amount of information leaked by observing that feature.

Finally, to mitigate the vulnerability, Shark! uses the generated feature ranking to iteratively synthesize a packet padding and delaying strategy based on the top leaking features which minimizes the information leakage and time and space overhead based on the user's constraints. Because the aforementioned metrics conflict with each other (minimizing overhead means no padding, which means leakage is not reduced at all), we use a linear combination of leakage and overhead metrics as the objective function and the user can tune the weights for them based on the needs of each IoT application. At the end of the iteration, Shark! provides a mitigation strategy which is the optimum strategy with respect to user's constraints on information leakage and time and space overhead.

Assumptions about attacker capabilities.

In our attack model, we assume that an eavesdropper is able to capture traces of user actions using duplicate IoT devices (which is possible for commercial devices) and train classifiers using the captured traces to identify devices or user actions. To perform the attack, we assume that the attacker is either in the same local area network or on a network switch between the device and server to monitor the victim's network traffic. We also assume that the attacker has a way to identify different device or client application traffic by either observing the MAC addresses of IoT devices if they are in the local area network or using separate classifiers for device identification, which is feasible as shown in literature [80, 81] and our experiments. The attacker passively observes each trace associated with the action once and uses the classifiers to obtain the secret information (device type/action depending on the task). We assume that observation of the trace associated with the action happens only once because the attacker cannot cause the user to perform an action. This prevents the amplification strategy to circumvent packet padding as the attacker cannot cause the user to perform the same action multiple times knowingly. We also assume that the attacker has no prior information that can impact its information gain such as sleep patterns or job schedules of the users.

5.2 IoT Network Model

In this paper, we focus on network communications to and from IoT middleware or backend servers. IoT device users send commands to or get updates from their devices using clients such as their smart phones. The servers relay this information to and from the devices or the hubs. We are not interested in monitoring the local wireless communication or personal area networks (PAN), therefore the standard local protocols for IoT like Zigbee, Z-Wave, Bluetooth, or Wi-Fi are beyond the scope of our analysis. Focusing on communications over the internet increases the attack surface by enabling remote attacks. An eavesdropper on the local network can only collect encrypted packets when they are close to the signals of emitters or when they have access to the devices of the victims. Communication with remote hosts via the internet enables eavesdroppers to attack at various points during data transmission.

There are several IoT protocols for communicating over the internet. gRPC [82] is a remote procedure call (RPC) framework based on HTTP/2.0. It encodes data with Protobuf, a serialization library that converts objects to binary streams. MQTT [83] is an open-source protocol standard designed to support communications between two or more separated devices. It has several implementations like RabbitMQ, HiveMQ, and AWS IoT. Since no default serialization method is provided, developers commonly conduct JSON serialization on objects sent in request and response. STOMP [84] is an alternative to MQTT, commonly used in low-energy devices as it provides minimal functionality. Similar to MQTT, developers have full freedom on deciding which serialization methods to use. Other protocols such as Websocket, AMQP, JMS, and CoAP are also commonly used in the IoT ecosystem. However, since they share most patterns of the aforementioned protocols [85], in our experimental evaluation, we only focused on the three protocols listed above. Our side-channel analysis technique is applicable to applications written using any of these communication protocols.

5.3 Feature Extraction and Prioritization

In this section, we describe the initial steps of our targeted mitigation approach (see Figure 2.3), the feature extraction and feature prioritization techniques.

5.3.1 Feature Extraction from Network Traces

To have an accurate information leakage analysis, we have to process the traces obtained from capturing network packets and extract meaningful features from the network packets. Based on our definition of traces and the corresponding secrets in Section 5.2, we define feature functions $f : \mathbf{T} \rightarrow \mathbb{R}$ (where $\mathbf{T}$ is the domain of traces) which map each trace t to a numerical feature value that can potentially correlate with the corresponding action label y. We extract packet-based features such as size of each packet and inter-arrival time to the consecutive packet. We also extract trace-level features such as the total size, total duration, mean, standard deviation, min and max of packet size and time differences. These feature definitions are based on the leakage sources in the prior work [21, 73]. Our feature function definitions are described in Table 5.1. For each function f_i, we apply it to the trace set T to obtain the vector of values for that feature, $F_i = \langle f_i(t_{(1)}), f_i(t_{(2)}), \ldots, f_i(t_{(|T|)}) \rangle$ which we use to determine the probability distributions.

Table 5.1: Definition of network trace features.

Feature Function	**Definition**	**Description**
$f^{\text{sum-size}}(t)$	$\sum_{p \in t} p.\text{size}$	Sum of sizes of packets in trace t.
$f^{\text{max-size}}(t)$	$\max_{p \in t} p.\text{size}$	Max. of sizes of packets in trace t.
$f^{\text{min-size}}(t)$	$\min_{p \in t} p.\text{size}$	Min. of sizes of packets in trace t.
$f^{\text{size}}(t, i)$	$p_i.\text{size}$	Size of packet i.
$f^{\text{num-pkt}}(t, k)$	$\sum_{p \in t} [p.\text{size} = k]$	Number of packets with size k.
$f^{\text{var-size}}(t)$	$\sigma(p.\text{size} \in t)$	Std. deviation of sizes of packets in trace t.
$f^{\text{avg-size}}(t)$	$\sum_{p \in t} p.\text{size}/\lvert t \rvert$	Avg. of sizes of packets in trace t.
$f^{\text{duration}}(t)$	$p_n.\text{time} - p_1.\text{time}$	Total time of trace t.
$f^{\Delta\text{time}}(t, i)$	$p_{i+1}.\text{time} - p_i.\text{time}$	Time diff. of packets i & $i + 1$.

5.3.2 Feature Prioritization via Leakage Quantification

To target mitigation to specific features, we need to have a metric of importance where mitigating more *important* features would reduce the information leakage more than mitigating less important features. To measure and quantify importance of each feature, we use an information theoretic measure, Shannon entropy [45] which quantifies information in terms of amount of bits. If we have n unique secret values, $\log_2 n$ will be the maximum amount of information leakage for that secret according to Shannon entropy. We define a feature vector $F_i = \langle f_i(t_{(1)}), f_i(t_{(2)}), \ldots, f_i(t_{(|T|)}) \rangle$ to represent the value of feature i over each trace $t_{(j)}$ in T. Recall that, we use a secret vector $\vec{y} = (y_{(1)}, y_{(2)}, ..., y_{(|T|)})$ which represents the secret value $y_{(j)}$ of the corresponding trace $t_{(j)}$. We use mutual information $I(\mathbf{Y}; \mathbf{X}_i)$ to quantify the information leakage for feature i, where $\mathbf{Y}$ is the domain of all secret values and $\mathbf{X}_i$ is the domain of feature values for feature i. The mutual information for feature i, I_i, is defined as

$$I_i = I(\mathbf{Y}; \mathbf{X}_i) = -\sum_{y \in \mathbf{Y}} p(y) \log_2 p(y) - \left(-\sum_{x \in \mathbf{X}_i} \hat{p_i}(x) \sum_{y \in \mathbf{Y}} \hat{p_i}(y|x) \log_2 \hat{p_i}(y|x) \right)$$

where the first part of the equation, initial entropy, represents the initial uncertainty about the secret. Second part of the equation, conditional entropy, represents the uncertainty after observing a feature value. The subtraction of these two measures gives the average amount of information leaked by observing the feature values F_i.

We do not have the exact probability distributions $p_i(x)$, $p_i(y|x)$. Therefore, we first compute the estimated probability distribution $\hat{p}_i(x|y)$ using the feature vector F_i and the corresponding secret vector $\vec{y}$. We use Kernel Density Estimation (KDE) with k-fold cross validation [49] to estimate the probability distribution $\hat{p}_i(x|y)$ [21, 47–49]. We assume the secret distribution $p(y)$ to be uniform to represent the attacker has no prior information about the secret. It can be modified in the cases of prior information to reduce the initial amount of information. We compute $\hat{p}_i(x)$ and $\hat{p}_i(y|x)$ using $p(y)$ and $\hat{p}_i(x|y)$ and the Bayes' rule.

After quantifying the information leakage of each feature separately, Shark! ranks them from the highest amount of information leaked to the lowest, which we then use in our targeted mitigation strategy. Algorithm 5 describes the feature ranking method where the highest ranked feature f_m is found by the formula: $m = \arg\max_i I_i$.

Procedure 5 FeaturePrioritization(T, L_f) Given a set of traces T, and a list of feature functions L_f, FeaturePrioritization extracts features over traces, ranks them based on the information leakage per feature and returns a list of feature rankings.

```
1: $L_f^{\text{ranked}} \leftarrow \langle\rangle$
2: for each $f_i$ in $L_f$ do
3:     $F_i \leftarrow \langle f_i(t_{(1)}), f_i(t_{(2)}), \dots f_i(t_{(|T|)})\rangle$        ▷ Extract feature $f_i$ from traces
4:     $Q_i \leftarrow$ QuantifyLeakage($F_i$)        ▷ Quantify information leakage for feature $f_i$
5:     $L_f^{\text{ranked}}$.append($Q_i, f_i$)
6: Sort($L_f^{\text{ranked}}$)        ▷ Sort features based on the information leakage amount
7: return $L_f^{\text{ranked}}$
```

5.4 Pareto Optimal Mitigation for Multiple Features

To obtain a mitigation strategy that balances the trade-off between added network overhead and remaining information leakage, we define an optimization problem based on obtaining low cost and low information leakage. We describe our cost models and tunable mitigation search below.

5.4.1 Cost Models.

To measure the impact of any mitigation method on the cost of the transmission, we use metrics based on byte overhead and timing overhead and find how much our mitigation technique impacts the network traffic. To compare two trace sets, we use T to denote the original trace set and T' to denote traces where the mitigation strategy is applied.

Cost for space mitigation on two sets of traces can be measured as

$$C_{space}(T, T') = \frac{\sum_{i=0}^{|T'|} f^{\text{sum-size}}(t'_{(i)}) - \sum_{i=0}^{|T|} f^{\text{sum-size}}(t_{(i)})}{\sum_{i=0}^{|T|} f^{\text{sum-size}}(t_{(i)})}$$

Similarly, cost for time mitigation on two sets of traces can be measured as

$$C_{time}(T, T') = \frac{\sum_{i=0}^{|T'|} f^{\text{duration}}(t'_{(i)}) - \sum_{i=0}^{|T|} f^{\text{duration}}(t_{(i)})}{\sum_{i=0}^{|T|} f^{\text{duration}}(t_{(i)})}$$

These cost metrics represent the average increase in number of bytes transmitted and duration of the traces, respectively. Increase in either of these metrics would impact power usage, quality of communication and total used bandwidth as transmissions would either take more time, more bandwidth or both.

5.4.2 Tunable Mitigation on Targeted Features.

Using the feature ranking obtained over traces, we can generate a mitigation strategy based on the top leaking features. This generated strategy needs to balance the constraints of the user on information leakage and overhead of the leakage on the communications. In addition, this strategy needs to be applicable to unseen traces as well, therefore our strategy should not be specific to mitigating information leakage in our dataset. With these constraints in mind, we present our targeted mitigation strategy. It takes a set of traces $T = (t_{(1)}, t_{(2)}, ...)$, iteratively modifies T to T' based on the feature ranking L_f^{ranked}, and generates mitigation rules if the modification is improving the user constraints.

We use three tunable parameters, α, β, and γ to define the objective function to minimize. These parameters specify the user constraints on leakage and overhead. The objective function $\Theta(T', T)$ is defined as

$$\alpha \times \textsc{QuantifyLeakage}(T') + \beta \times C_{space}(T, T') + \gamma \times C_{time}(T, T').$$

The parameter α denotes the weight of the information leakage of the modified set of traces T' (higher α corresponds to higher emphasis on lowering leakage). The parameter β denotes the weight of the space cost of mitigation (higher β corresponds to higher emphasis on low space overhead), and the parameter γ denotes the weight of the time cost of mitigation (higher γ corresponds to higher emphasis on low time overhead).

We describe our mitigation approach in Algorithm 6. As seen in Lines 3–6, our approach iterates over the ranked feature list L_f^{ranked}, modifies the traces based on the feature (which we explain in the following subsection), quantifies the information leakage of the modified trace and calculates the objective function value Θ. $\textsc{QuantifyLeakage}(T')$ performs feature extraction and quantification over the modified set of traces T' and returns the highest information leakage measure over all features as described in Sections 5.3.1 and 5.3.2. If the objective function is minimizing compared to the previous iteration (Lines 7–10), our approach updates the current minimum value Θ_{min} to Θ, updates the trace set T to the modified trace set T' and saves the feature for the revised feature ranking L_f^{revised}. If the modification based on the feature does not improve the objective function (it could be because the modification increases the overhead too much), we exclude the feature from the revised feature ranking. After the execution, our approach returns the revised feature ranking L_f^{revised} which can be used to modify unseen traces.

Procedure 6 TargetedMitigation($T, L_f^{\text{ranked}}, \alpha, \beta, \gamma$) Given a set of traces T, list of ranked features L_f^{ranked}, leakage weight α, space cost weight β and time cost weight γ, TargetedMitigation iterates over the features and shapes the traffic to reduce the information leakage while minimizing the objective function, returning the list of features L_f^{revised} that improve the objective function (where $\Theta_{\min}$ denotes the current minimum value of the objective function).

```
1: $\Theta_{\min} \leftarrow \infty$
2: $L_f^{\text{revised}} \leftarrow \langle\rangle$
3: for $j \leftarrow 1$ to $|L_f^{\text{ranked}}|$ do
4:     for each $t_{(k)}$ in $T$ do
5:         $t'_{(k)} \leftarrow$ Modify($t_{(k)}, L_f^{\text{ranked}}[j]$)      ▷ Modify trace to reduce leakage from feature $L_f^{\text{ranked}}[j]$
6:     $\Theta \leftarrow \alpha \times$ QuantifyLeakage($T'$) $+ \beta \times C_{space}(T, T') + \gamma \times C_{time}(T, T')$
7:     if $\Theta < \Theta_{\min}$ then
8:         $\Theta_{\min} \leftarrow \Theta$
9:         $T \leftarrow T'$
10:        $L_f^{\text{revised}}$.append($L_f^{\text{ranked}}[j]$)
11: return $L_f^{\text{revised}}$      ▷ List of features that improve the objective function
```

5.4.3 Targeted Trace Modification

For obfuscating space and time side-channels, we define and utilize three methods to modify the traces. First method Pad(t, D) pads every packet with a padding size based on the distribution D. This function helps with defining padding over all packets to mitigate information leakage of aggregate features such as $f^{\text{sum}-\text{size}}$. It takes a trace $t = (p_1, p_2, ..., p_n)$ and returns $t' = (p'_1, p'_2, ..., p'_n)$ where p'_i.size $\leftarrow p_i$.size $+ x_i$ and $x_i \sim D$.

Second method, Delay(t, d_{limit}), delays each packet based on the uniform delaying up to a certain limit. This helps with mitigating delays over multiple packets such as f^{duration}. It takes a trace $t = (p_1, p_2, ..., p_n)$ and returns $t' = (p'_1, p'_2, ..., p'_n)$ where if p_i.src $\leftarrow p_{i+1}$.src, p'_i.time $\leftarrow \mathcal{U}(p_i$.time$, p_{i+1}$.time$)$ if and p_i.dst $\leftarrow p_{i+1}$.src, for all $j \geq i$, p'_j.time $\leftarrow p'_j$.time $+ \mathcal{U}(0.0, \max(\mu_{\delta\text{time}}/2, d_{limit})$. This delay injection assumes two party communication (e.g. between server and client). If two packets are sent from the same source, the first one can be delayed at most until the next packet. If the destination of one packet is the source for the next packet, we assume the second packet is a response to the first. In this case, delaying the request would delay the response and all the packets that come after it. In this case, we delay those packets at most half of the average time difference between consequent packets or the delay limit d_{limit} if the time difference is too large.

Our third method, $\textsc{Inject}(t, k, s)$, injects k random packets with size s into the trace. It takes a trace $t = (p_1, p_2, ..., p_n)$ and returns $t' = (p'_1, p'_2, ..., p'_{n+k})$ where k packets are added to the trace where for any injected packet p'_i, its source, destination and ports are sampled from existing packets, and size and timing of the injected packet is defined as $p'_i.\text{size} \leftarrow s$ and $p'_i.\text{time} \leftarrow \mathcal{U}(p_1.\text{time}, p_n.\text{time})$. Injecting extra packets can obfuscate side-channels caused by both timing and space side-channels such as number of packets with a specific size or timing delays between certain packets and we use this method to obfuscate various types of information leakages.

For obfuscating different types of features, we employ different packet modifications such as changing the size of packets by padding the content, delaying the packets or injecting new packets as explained using the aforementioned methods. Using these modifications, we explain how $\textsc{Modify}(t, f)$ works for each feature type to mitigate side-channels based on each type of feature.

- For feature $f^{\text{size}}(i)$, we equalize the size of i^{th} packet in each trace to avoid information leakage. For each trace t, we modify the size of i^{th} packet p_i to the maximum size of i^{th} packet over all traces. It can be described as $\forall t' \in T', p'_i.\text{size} \leftarrow \max_{t \in T} f^{size}(t, i)$.
- For feature $f^{\Delta\text{time}}(i)$, we delay the $(i+1)^{\text{th}}$ packet to equalize the delta between them. For each trace t, we modify the response time for the $(i+1)^{\text{th}}$ packet to maximum delay between i^{th} and $(i+1)^{\text{th}}$ packets. It can be described as $p'_{i+1}.\text{time} \leftarrow \max_{t \in T} f^{\Delta\text{time}}(t, i) + p_i.\text{time}$.
- For feature $f^{\text{max}-\text{size}}$, for each trace t, we inject a packet to t with size equal to maximum size over all packets to obfuscate this feature. For each t, we modify it to create t' where

$t' \leftarrow \textsc{Inject}(t, 1, \max_{t \in T} f^{\text{max-size}}(t))$.

- For feature $f^{\text{min-size}}$, we pad all packets with size below a threshold to equalize the sizes of minimum packets. It can be described as $\forall p \in t, p.\text{size} < \max_{t \in T} f^{\text{min-size}}(t) : p'.\text{size} \leftarrow \max_{t \in T} f^{\text{min-size}}(t)$.
- For feature $f^{\text{num-pkt}}(k)$, we inject packets to equalize the number of packets with size k. For each t, we pick a random number of packets $n \sim \mathcal{U}(0, 2 \times \max_{t \in T} f^{\text{num-pkt}}(t, k))$ and modify the original trace to create t' where $t' \leftarrow \textsc{Inject}(t, n, k)$.
- For the size and timing based aggregate features, ($f^{\text{sum-size}}$, $f^{\text{var-size}}$, $f^{\text{avg-size}}$, f^{duration}) we search for a padding strategy to apply to all packets by searching for best parameters D or d_{limit} over a set of parameters for $\textsc{Pad}(t, D)$ and $\textsc{Delay}(t, d_{\mathit{limit}})$ respectively. We describe the set of parameters used for the search method in Section 5.5.1.

5.4.4 Online Mitigation

As shown in the previous subsections, while performing the search in Algorithm 6, trace modification ($\textsc{Modify}(t, f)$) and leakage quantification ($\textsc{QuantifyLeakage}(T')$) is done offline and applied to each trace. In real traffic, this is not possible as the traces must be processed and modified as each packet arrives. To address this, using the modified feature set, we implement a packet-based mitigation system that modifies each packet based on the results of the search in Algorithm 6. Using the results of the search, we can synthesize the online mitigation that takes a packet and modifies it based on the features we should modify to reduce the information leakage.

Algorithm 7 describes a method that processes each packet based on the mitigations and returns the modified packet. It takes a packet p, modifies it based on the packet index i and each leaking feature in the revised feature list. Instead of the $\textsc{Modify}(t, f)$ described in Section 5.4.3, we use a method called $\textsc{ModifyPacket}(p, i, f)$ which extends $\textsc{Modify}(t, f)$ on a packet p_i

rather than the full trace t. For PAD and DELAY functions, there's no change as both methods apply same modifications to each packet of the trace indiscriminately. For INJECT(t, k, s), as we limit it to injecting k packets of size s within a trace t, we obtain the average number of packets per trace from the training set T as $\mu_{\text{num}-\text{pkt}}$ and inject a packet for each $\mu_{\text{num}-\text{pkt}}/k$ packets.

A router that processes packets can use this method to apply padding and delays to each received packet that is transmitted between IoT device, server and smartphone application. In a real world implementation, this algorithm would need to be optimized to apply the mitigation quickly and send the packet with minimal delays and it can be done as shown in prior work [73] and we will demonstrate it in our experimental evaluation.

Procedure 7 ONLINEMITIGATION$(p, L_f^{\text{revised}})$ Given a packet p and the revised feature set L_f^{revised}, ONLINEMITIGATION modifies the packet based on the targeted features. We do not include dummy packet injections in this description, those packets are sent without processing any packets.

```
1: $i \leftarrow i + 1$                                        ▷ Packet index counter
2: for $j \leftarrow 1$ to $|L_f^{\text{revised}}|$ do
3:     $p \leftarrow$ MODIFYPACKET$(p, i, L_f^{\text{revised}}[j])$    ▷ Modify packet to apply mitigation strategy
       based on feature $L_f^{\text{revised}}[j]$, described in Section 5.4.3
4: return $p$                                   ▷ Modified packet based on padding and delays
```

5.5 Implementation and Experimental Evaluation

In this section we first describe our implementation, followed by the discussion of benchmarks we used in our experiments, and then describe the results of our experimental evaluation.

5.5.1 Implementation

SHARK! is implemented in Python. For trace capture, automation of analysis, mutual information calculation, and feature ranking capabilities, we use tools described in [18, 21]. For online mitigation, we used Scapy's network capture, padding and packet sending capabilities [52].

For measuring the effectiveness of the mitigation, we used existing implementations of random forest classifier, k-nearest neighbor, and fully connected neural network algorithms in *scikit-learn* library [53]. For the random forest classifier, we set the number of decision trees to 100 which we obtained after testing the random forest with 50, 100, 150 and 200 and picking the value maximizing accuracy after cross validation.

For the mitigation parameters D and d_{limit} of $\textsc{Pad}(t, D)$ and $\textsc{Delay}(t, d_{limit})$, we use grid search over a set of fixed parameters, picking the parameter that minimizes the the objective function. For D, we use various uniform distributions, $\mathcal{U}(0, k)$ where $k \in \{25, 50, 100, 150, 200, 250\}$ bytes and existing packet padding methods in the related work such as exponential padding. We also calculate the per packet size difference between the largest and smallest traces and set that parameter as k for the search. For delay parameter d_{limit} search, we use the set of parameters $\{10\text{ms}, 20\text{ms}, 50\text{ms}, 100\text{ms}, 200\text{ms}\}$. We chose these range of parameters to represent mitigation methods with low and high impact on information leakage and network overhead.

For the experimental evaluation of Shark! against related work, we set γ to ∞ and only use α and β parameters to focus on space cost and accuracy in the objective function to have a fair comparison between Shark! and most of the prior work which focuses on only packet padding. For quantification, we use the default parameters in [21] implementation which works for estimating distributions of space and time features. Our approach and some of the prior work behaves stochastically and to address this issue, we ran each experiment 3 times and averaged the results over 3 runs to alleviate the randomness.

5.5.2 IoT Benchmarks

IoT protocol benchmarks.

To cover the common design and architecture patterns in our experimental evaluation we used the gRPC, MQTT, and STOMP protocols discussed in Section 5.2 to create applications with various protocols. We created 10 applications using the protocols representing both smart home and industrial IoT systems such as ovens, air conditioners, smart locks and electrical switches. Details of the applications can be examined in this repository [86].

To create the trace dataset for evaluation, we captured 2000 traces per secret value for each application by inducing the action or system state. For the user-controlled applications, we obtained the traffic generated by both the client and the device, and we analyze them separately as the attacker could have access to only one stream. For sensors, we only captured the traffic between server and device as there is no client to give commands.

The applications in our IoT benchmark are written in Python. The benchmark is publicly available in an anonymized repository [86]. Table 5.2 summarizes the types of each device and the protocol that is being used. In each application, there is a device service to represent the IoT device updating information and receiving actions; the client service to represent control systems of IoT devices; and the backend service to coordinate the communication between device service and client service. For the MQTT (AWS IoT) and STOMP applications, we only implemented client and device since neither allows logic definitions at the server side. While we leverage AWS MQ platform to create the server that supports communication between client and device via STOMP protocol, AWS IoT platform is used for application using MQTT protocol. For gRPC applications, we define our own functions like logging in and registering device at server side instead of implementing the server to be a general message queue like the former applications. We deploy our server to a Google Kubernetes Engine instance with 3 nodes (each having 4 cores vCPU and 11 GB RAM). To simulate the real-world production environment and imitate the timing features in the backend of real-world IoT services, we also utilize a SQL database (MySQL) and a cache database (Redis).

For each application, the secret of interest is the general actions the user can take if it is user-controlled (like a light) or the state of the device if it is a sensor (like a camera). For air conditioner apps, the possible actions are turning the AC to make the air hotter, colder, blowing

Table 5.2: All IoT benchmark applications and secret domain size of each application.

Application	Device Type	Protocol	$\lvert\mathbf{Y}\rvert$
grpc_ac	Air Conditioner	gRPC/Protobuf	6
grpc_cctv	CCTV	gRPC/Protobuf	2
grpc_oven	Oven	gRPC/Protobuf	4
grpc_switch	Electrical Switch	gRPC/Protobuf	3
awsiot_ac	Air Conditioner	MQTT (AWS IoT)	6
awsiot_oven	Oven	MQTT (AWS IoT)	5
awsiot_lock	Lock	MQTT (AWS IoT)	5
stomp_ac	Air Conditioner	STOMP (AWS MQ)	5
stomp_oven	Oven	STOMP (AWS MQ)	4
stomp_lock	Lock	STOMP (AWS MQ)	2

hot or cold air with the fans, or turning off the device. For oven apps, the possible actions are to broil, stop heating, or bake with or without a fan. For the lock apps, the different actions the user can take are to lock or unlock the lock. For the CCTV app, the states are no motion or motion detected. For the electrical switch app, the states are the switch being on, off, or in an unknown state.

For all MQTT applications and grpc_ac applications, we also include no action where the user does not do anything as a secret of interest. This is important as we want to be able to detect whether an action happens or not as well. We did not include no action case as a secret of interest in STOMP applications and grpc_oven; the reason is that in our implementation, both clients and devices just re-transmit the latest action given to the device in those implementations, therefore there is no observable difference between the previous action and no action state.

IoT real-world benchmarks.

We used an existing IoT application benchmark from previous work, the UNSW device identification dataset [79] which contains network traces from a variety of IoT devices in which the task is to identify devices.

In addition to the UNSW benchmark, we also used four IoT devices, a Samsung SmartThings camera, a SmartThings magnetic motion sensor, a Sengled smart light bulb, and MyQ garage door, to generate network traffic with various user actions. To sniff the network traffic, we used a computer with Ubuntu 20.04 OS as a Wi-Fi hotspot to monitor the traffic and capture the device traffic. For each user action, we generated 100 traces where each trace is 15 seconds long. For the SmartThings camera, we generated traces where no action and no motion is detected, the sound is detected without motion, and both sound and motion is detected. For the motion sensor, we generated traces where the motion happened or did not happen in front of the sensor. For the smart light bulb, we obtained traces where the user performs the action of turning on or turning off the lights or doing nothing. For the garage door, we generated traces for opening, closing the garage door, and where nothing happens.

5.5.3 Experimental Evaluation

We compare our approach against prior mitigation methods which use a fixed or randomized packet padding strategy [72–75] such as such as linear padding (increasing sizes of each packet to the nearest increment of 128) [72], exponential padding (increasing sizes of each packet to the nearest power of 2) [72], uniform padding (padding each packet with 1-1500 bytes randomly) [72, 74], uniform-255 padding (padding each packet with 1-255 bytes randomly) [72], maximum transmission unit (MTU) padding (increasing size of each packet to maximum transmission unit of 1500 bytes for TCP/UDP packets) [72], MTU padding with 0-20 ms delay to each packet (MTU-20ms) [75], mice & elephants padding (increasing size of each packet with size less than 100 to 100, to 1500 - max packet size otherwise) [72], Level-X where X is 100, 500, 700 or 900 (increasing size of every packet with size less than X to X, pad them randomly otherwise) [73]. None of the aforementioned approaches had a public implementation, therefore we implemented all the methods ourselves.

To evaluate Shark! against the various related work and compare its performance against advanced attacks, we trained random forest classifiers [63, 64], k-nearest neighbors [65] and fully-connected neural networks [66] on the traces. Random forest classifiers performed the best in terms of accuracy, therefore we used random forest classifiers in our evaluations. Hence, we are evaluating the ability of different mitigation strategies against the best performing classifier's ability to infer the secret from network traces.

For a fair evaluation, we split the trace set T into two trace sets with equal size called *seen* traces, T_{seen}, and *unseen* traces, T_{unseen}, simulating the case where the mitigation strategy is synthesized offline and then deployed on the device. We synthesize our mitigation strategy only on *seen* traces and apply our mitigation strategy and test our approach on *unseen* traces. To compare the side-channel mitigation methods of Shark! and the prior work, we split the unseen trace set T_{unseen} to training trace set T_{train} and testing trace set T_{test} in 80%/20% split with 5-fold cross-validation to alleviate cases where the arbitrary splitting of trace sets can affect the classification results. We train the classifier on T_{train} and measure the accuracy, precision, recall, F_1 score of the classifier on T_{test}.

Comparison of Shark! to prior work over various classifiers.

For comparison purposes, we trained random forest classifiers for the aforementioned prior works and Shark! over 10 IoT protocol benchmark applications, 4 real world applications and UNSW trace dataset. As a baseline, we also include the performance of the classifiers where we use traces with no mitigation, with full mitigation (padding all packets to full size, delaying packets to make the transmission like heartbeat and injecting packets to make trace size equal overall) and random guessing probability.

Table 5.3 shows the average accuracy, precision, recall, F$_1$-score [87] and space cost using the random forest classifier. The results show that when our mitigation approaches the ones with prefix SHARK!) are used, modified traces leak less information and induce less overhead compared to prior work depending on the objective function and the user can select a trade-off between them. SHARK! (Overhead) achieves lowest overhead compared to other works while reducing leakage some amount. SHARK! (Balanced) has higher overhead compared to exponential padding approach but reduces leakage, precision and recall to lower levels. SHARK! (Leakage) has high overhead compared to most of the works but it achieves leakage very close to full mitigation with only 3x space cost. Our method is able to mitigate sources of side-channels by padding a single packet or injecting a few packets which have low impact on the overhead whereas it can reduce the information leakage significantly. For the aggregate features, our approach finds the padding that improves the objective function, therefore our feature prioritization and iterative mitigation synthesis using the objective function always improves upon the padding over previous steps, enabling SHARK! achieve lower overhead, leakage or both.

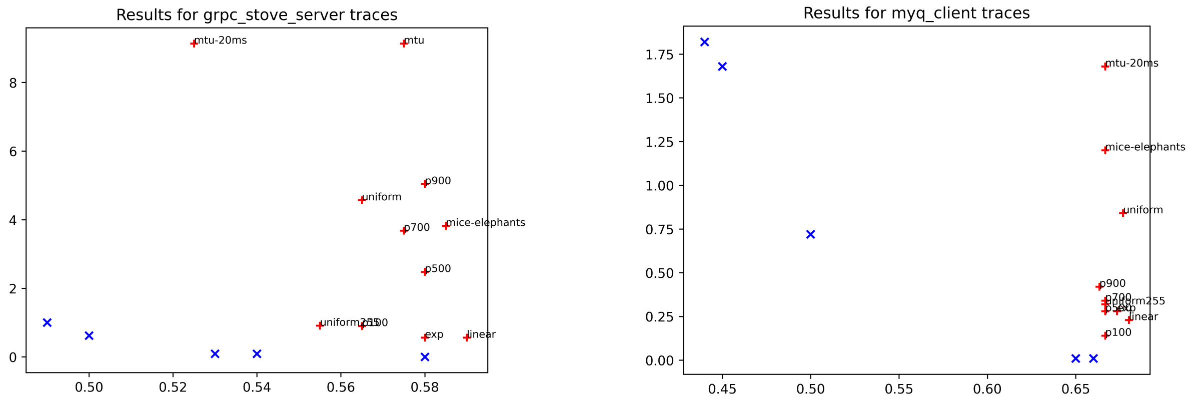


Figure 5.2: Average accuracy (x-axis) and overhead (y-axis) results of prior work and SHARK! using the random forest classifier. Red pluses represent the results of prior approaches, blue crosses represent the results of SHARK! with different weights for objective function. Lower values are the better results.

Table 5.3: Average testing accuracy, precision, recall, F_1-score and packet size overhead results of the prior work and SHARK! with 6 objective functions on all benchmarks, trained on a random forest classifier. Overhead, Balanced and Leakage results of SHARK! represent cases where the objective function weights α (leakage weight) and β (overhead weight) are 1 and 1, 1 and 0.1, 1 and 0.01 respectively. Bold values are minimum values among the methods.

Mitigation Method	Accuracy	Precision	Recall	F_1-Score	C_{space}
No mitigation	0.83	0.84	0.83	0.83	**0.00**
Full mitigation	**0.44**	**0.44**	**0.44**	**0.44**	113.39
Random Guess	0.30	0.30	0.30	0.30	N/A
Uniform	0.55	0.55	0.55	0.54	7.48
Uniform255	0.56	0.57	0.56	0.56	1.35
Mice & Elephants	0.57	0.57	0.57	0.57	2.62
Linear	0.57	0.58	0.58	0.58	0.93
Exp	0.58	0.58	0.58	0.58	**0.32**
MTU	0.56	0.56	0.56	0.56	14.92
MTU-20ms	**0.53**	**0.54**	**0.54**	**0.53**	14.92
Level-100	0.58	0.59	0.59	0.58	0.76
Level-500	0.56	0.57	0.56	0.56	4.54
Level-700	0.56	0.57	0.56	0.56	6.58
Level-900	0.56	0.57	0.56	0.56	8.65
IoTPATCH (Overhead)	0.67	0.56	0.55	0.54	**0.09**
IoTPATCH (Balanced)	0.52	**0.46**	**0.47**	**0.46**	0.65
IoTPATCH (Leakage)	**0.47**	0.49	0.49	0.48	3.01
IoTPATCH (Overhead w/Time)	0.64	0.55	0.55	0.54	**0.09**
IoTPATCH (Balanced w/Time)	0.50	**0.45**	**0.46**	**0.46**	0.65
IoTPATCH (Leakage w/Time)	**0.45**	0.47	0.47	0.47	3.01

Pareto optimality of Shark!.

Figure 5.2 shows the accuracy (x-axis) and space overhead (y-axis) results for the prior work and Shark! with different objective function parameters which weigh overhead and leakage at different levels. Both plots show that Shark! with various objective functions provides a Pareto optimal solution set, where different objective functions result in different points in the accuracy-overhead space. The users can run Shark! with different objective functions, get the mitigation strategies with various results and pick the one that fits their needs and requirements. For example, in the MyQ garage door example in Figure 5.2, they can pick the strategy with low overhead where the attacker can guess accurately with 65% accuracy or pick the strategy with higher overhead while reducing the accuracy of the attacker to 45%.

Effectiveness of targeting timing side-channels.

To demonstrate the value of targeting timing side-channels, we set the timing overhead weight γ equal to 0.1 for the objective function and compared Shark! with and without timing information leakages.

Table 5.3 shows the results of Shark! with/without the timing mitigation. Compared to Shark! without any timing mitigation, timing mitigation reduces the testing accuracy 2-3% more, similar to the accuracy difference between MTU and MTU-20ms padding. The average time cost for Shark! with targeting side-channels is 10% whereas traces where MTU with 0-20 ms delays have in average 90% time cost. Results of both our approach and MTU with 0-20 ms delays demonstrate the importance of targeting timing side-channels and our method only uses timing mitigation when it is needed, achieving lower overhead.

Limitations.

Our results on feature prioritization and side-channel mitigation depend on the quality of the captured network trace set that contains a variety of user behaviors. If the number of traces are low or they are captured in a way such that some other unrelated event (such as time of day, device updates, etc.) correlates with the action, Shark! can try to mitigate the traffic, assuming it leaks information when in fact it does not leak information in the real world. To alleviate validity concerns and to simulate attacker conditions, in our evaluations we split the trace sets such that we synthesize the mitigation strategy on seen traces and demonstrate effectiveness of mitigation against attacks on another set of unseen traces.

Shark! detects and quantifies the information leakage and it can find the optimum mitigation strategy to reduce leakage, however implementing that strategy is left to the user. The mitigation strategies generated by Shark! can be implemented on a network similar to the prior works [73,75] which use software defined networking to manipulate network traffic data.

5.6 Chapter Summary

We presented a targeted black-box side-channel mitigation approach for IoT applications called Shark! which analyzes captured network traces by extracting features based on packet sizes and timings and creates a feature ranking based on the information leakage quantification. Shark! uses this feature ranking to synthesize a mitigation strategy based on the needs of the user, balancing the trade-off between the information leakage and mitigation overhead. We evaluate our approach on network traces collected from a set of IoT applications with various protocols, four IoT devices and a device identification dataset. Our experimental results demonstrate that Shark! outperforms the prior work and provides Pareto optimal mitigation strategies based on user's constraints.

Chapter 6

TSA: A Tool for Network Side-Channel Analysis, Quantification, and Mitigation

In this chapter, we present TSA, a **T**ool for detecting and quantifying network **S**ide-channels **A**utomatically with black-box testing and input generation methods. TSA is based on the technical approaches presented in previous Chapters 2 and 3 [18,21]. We extend these prior works into a flexible open source tool with documented APIs which the users can utilize to analyze their applications. In network trace analysis, we provide the option of trace alignment to extract more meaningful features for trace analysis. To demonstrate the capabilities of TSA, we analyze 7 applications in DARPA STAC benchmark, and one IoT benchmark on device identification using TSA.

TSA can be used in two ways. If the user is testing an application and has not collected any traces, they can use TSA to generate inputs to test the application, capture the network traces, and analyze the captured traces in a feedback driven loop where TSA terminates the analysis if the information leakage estimate converges to a value. If they already captured some traces previously, they can use TSA to just perform side-channel analysis and quantification.

When using TSA, the user provides some seed inputs for the target system, and a set of mutators which, given a valid input, return another one. The user chooses a *secret of interest*—some aspect of the input that they consider sensitive, whose leakage they want to detect and quantify. TSA then repeatedly executes the target system, generates new inputs, captures network traffic, and adjusts input generation strategy based on the feedback it obtains by analyzing captured traffic. For analysis, TSA extracts features that may leak side-channel information using the size, time and direction of the captured network packets. Afterwards, it computes the mutual information using Shannon entropy and finds features that maximize the information gain about the secret of interest. The final output from TSA is an automatically generated *ranking* of the top n most-leaking features, sorted by how much information they each leak about the secret of interest.

The *envisioned users* of TSA include researchers and software engineers, and other people who want to analyze the side-channel information leakage of their applications. The *challenge* we propose to address is automatically analyzing side-channel information leakages of applications using a small set of inputs and mutators. In Section 6.1, we go over the tool architecture and API to describe how it can be used for analysis. In Section 6.2, we describe the results of case studies to demonstrate different uses of the tool. In Section 6.3, we conclude the paper.

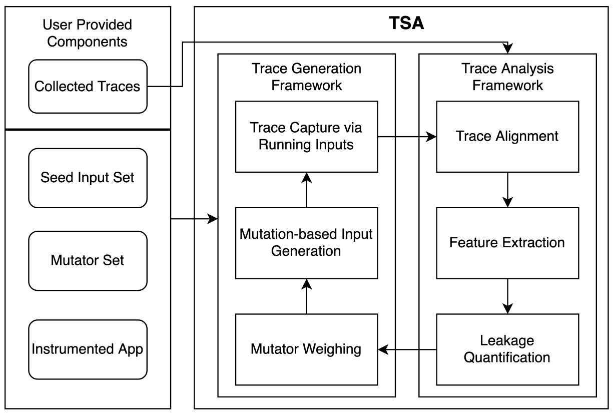


Figure 6.1: Architecture of TSA.

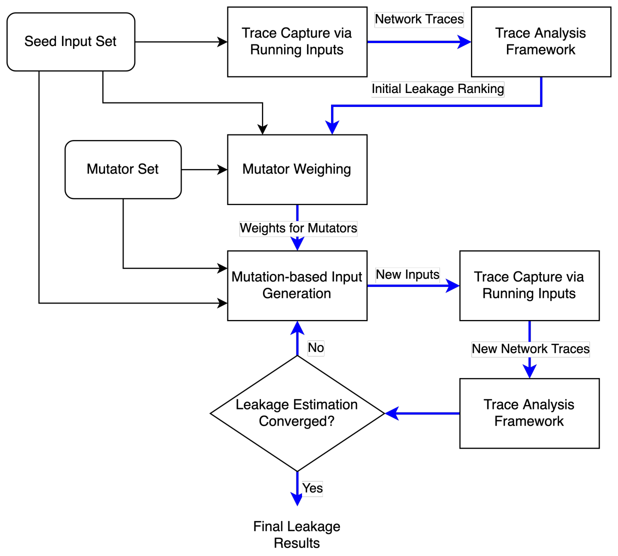


Figure 6.2: Workflow of TSA. Thick blue arrows denote the flow of execution.

6.1 TSA

In this section, we describe TSA's architecture and main workflow of TSA's execution. We also describe how TSA is used on an example application, how the user provides inputs, mutators and application orchestration.

6.1.1 TSA **Framework**

Figure 6.1 describes the core framework of TSA and two ways the user can provide data for the analysis. If the user provides seed inputs, mutators and an instrumented application, TSA uses its *trace generation framework* to generate new inputs and run those inputs to generate network traces. TSA's *trace analysis framework* takes the generated traces and performs side-channel analysis with trace alignment, feature extraction and quantifies the information leakage over each feature. The leakage quantification results are used to determine the importance of mutators which are used to generate new inputs in mutation-based input generation. If the user only provides collected and labeled network traces, TSA's *trace analysis framework* performs side-channel analysis as described and returns the leakage quantification results.

TSA **Workflow Summary**

Figure 6.2 describes the workflow of TSA when used with a set of seed inputs and mutators. We only describe the workflow of this use as TSA's workflow with a set of collected network traces is explained clearly in Figure 6.1 and in the previous section. To obtain an initial leakage estimation, TSA runs the seed inputs over the instrumented application to obtain an initial set of traces, uses trace alignment and feature extraction to obtain features and use quantification methods to quantify the information leakage. Using this initial leakage estimation, TSA evaluates the influence of mutators on the leakage estimation based on changes in top feature or secret and computes weights for mutators which are proportional to their likelihood of changing secret value or perturbing feature values.

After this initial setup, for each iteration, TSA generates new inputs using mutators and previous inputs based on the computed weights, runs these new inputs over the system to obtain new traces and runs the analysis over all of the collected traces to obtain the leakage estimation for that iteration. If the stopping criterion is satisfied, then it returns the final information leakage estimation on the application. Otherwise, it starts a new iteration, repeating the previously described steps. We perform analysis over all the collected traces and generate new inputs using all the previously generated inputs but we do not show the accumulation of traces and inputs on the figure for simplification.

6.1.2 TSA **API**

To analyze their applications, users need to define the input model, provide a set of mutators and write code to orchestrate system setup and execution. To make this process easier, we provide an API with classes for defining inputs, mutators and application orchestration. Listing 6.1 provides an example code segment the user may write to test an example shopping application extending TSA API classes. The TSA codebase contains examples with varying degrees of complexity, including apps, inputs, and mutators for the STAC benchmark which the users can refer to as well.

Input model

To help the users write inputs, we provide `Input` base class which represents a valid input for the application. Users can subclass `Input` and add fields and members to model the relevant characteristics. For example, if the user wants to test if their purchases are leaked, user can define a shopping list input as Python class `ShoppingInput` extending `Input` with a list named `shoppingcart` representing their purchases and string named `zipcode` representing the ZIP code of their shipping address. The users can also write assertions in the constructor such as ZIP code belonging to a set of valid US ZIP codes to check validity of the input when it is being created.

The only mandatory methods to implement are methods `hash` and `secret`. First method `hash` is used by TSA to check if the newly generated inputs are unique. A simple way to implement `hash` is to pack all relevant class members in a tuple and call Python's primitive `hash` method on that tuple. This ensures that any change in any member affects the resulting hash value. The second method `secret` defines the secret of interest in relation to the input. This is up to the user and in our example, it can be number of elements in the shopping cart, the total cost of all items in the shopping cart, prefix of user's ZIP code (denoting general area of delivery), or any other sensitive information.

Listing 6.1: Example usage of TSA API

```python
from tool import Platform, Container, Sniffer, App, Input, Mutator

class ExampleApp(App):
    def launch(self):
        Platform.cleanuphosts(["homer.example.edu", "marge.example.edu"])
        # Deploy two containers on two different machines
        self.servercontainer = Container("example/server:v1.0")
        self.clientcontainer = Container("example/client:v1.0")
        self.server = Platform.launch(self.servercontainer, "homer.example.edu")
        self.client = Platform.launch(self.clientcontainer, "marge.example.edu")
        # Run the server
        server_cmd = "bash -c 'cd /home/server && ./startServer.sh'"
        self.server.exec(server_cmd, detach=True)
    def shutdown(self):
        self.server.killrm()
        self.client.killrm()
    def run(self, inputs):
        sniffer = Sniffer(ports=[8080, 8081])
        sniffer.start()
        for input in inputs:
            sniffer.startinteraction(input.secret())
            self.client.createfile(input, "/home/client/input.txt")
            cmdfmt = "bash -c 'cd /home/client && ./startClient.sh {} {}'"
            self.client.exec(cmdfmt.format("homer.example.edu", "input.txt"))
        sniffer.stop()
        return sniffer.traces()

class ShoppingInput(Input):
    def __init__(self, shoppingCart, zipcode):
        assert len(shoppingCart) > 0
        assert len(zipcode) == 5
        self.shoppingCart = shoppingCart
        self.zipcode = zipcode
        self.itemList = ['apple', 'orange', ...]
    def __eq__(self, other):
        return self.shoppingCart == other.shoppingCart
            and self.zipcode == other.zipcode
    def __hash__(self):
        return hash((self.shoppingCart, self.zipcode))
    def secret(self):
        return len(self.shoppingCart)

class AddItem(Mutator):
    def mutate(input):
        randomItem = random.choice(input.itemList)
        input.shoppingCart.append(randomItem)
        return input

class RemoveItem(Mutator):
    def mutate(input):
        if len(input.shoppingCart) > 0:
            randomIndex = random.randrange(len(input.ShoppingCart))
            input.shoppingCart.pop(randomIndex)
            return input
        else:
            return None

class ChangeZIPCode(Mutator):
    # ... etc ...
```

Mutators

`Mutator` base class in TSA API represents a mutator that transforms valid inputs. The users can write their own mutators extending `Mutator` and providing their own implementation for the method `mutate`, which is a static method that takes an `Input` and returns another `Input`. The method assumes `Input` is a valid input for the system and tries to return another valid input. If it cannot, the method should return `None`. For example, the user may write a mutator which adds an item to the shopping cart or another mutator which removes an item from the shopping cart if possible.

Listing 6.1 shows an example with three mutators. The first mutator, `AddItem`, adds an item to the shopping cart field and returns the new input. The second mutator, `RemoveItem`, removes a random item from the shopping cart field if it is not empty, and returns the new input. If the shopping cart is empty, it returns `None` as it cannot remove items from an empty list. The third mutator, `ChangeZIPCode`, changes the ZIP code of the input from a set of valid ZIP codes, returning the new input. We provide the code for the first two mutators for space reasons as the code to check valid ZIP codes is complicated.

System Setup and Execution

To execute the system under test, we provide `App` base class which can be extended by the users. When implementing the instrumentation of system execution, the user must implement three methods, `launch`, `shutdown` and `run`. `launch` and `shutdown` methods set up the system before analysis and shut down the system after analysis respectively. `run` method takes a list of `Input` objects and runs them one by one over the system under test, returning a set of captured network traces. The user needs to provide how the inputs interact with the system by implementing `run` method. This imitates how a user might use the input as a scenario. For example, for `ExampleApp.run()` might have a script that searches each item of the input file on a website, puts the first result on the shopping cart and checks out using the ZIP code in the input.

To instrument deployment and launching of components such as clients, servers and peers, we use Docker [88] in our examples. We provide the classes called `Container` and `Platform` which set up and launch Docker containers respectively. We provide methods to create containers on hosts and on the containers, and we provide methods to copy files, run commands and shutdown. Using Docker is optional, but recommended for simplicity and reproducibility. This also allows running TSA analyses on cloud platforms with minimal changes.

Packet sniffing

To help with capturing traffic, TSA provides a `Sniffer` class that offers a simple interface for capturing traffic and labeling the captured traces. To set up network capture, the user can create a Sniffer object, denoting specific ports they want to listen and start the sniffing which runs in a separate thread. Before starting each interaction, the `App`'s `run` method should call `Sniffer.startinteraction(secret)` to ensure that the captured traffic is labeled with the correct secret. Lastly, `run` should finish sniffing with `Sniffer.stop()` method and return the traces obtained from `Sniffer`.

TSA **Setup**

Our tool runs in a feedback-driven manner, where it generates inputs by picking mutators based on a heuristic, generates new traces by running the inputs on the instrumented app and runs our analysis on the newly obtained traces. If the leakage estimation of top-k features do not change below an ϵ value for N steps, then the estimation stops. To setup this feedback loop, we provide default values to the stop criterion parameters but the users can provide their own values for k, ϵ and N variables to set up their own stop criterion. Users can also provide a parameter to determine how many times each input will run on the system. Some systems may exhibit non-deterministic behaviors, therefore running each input multiple times may be beneficial for the accuracy of the analysis.

6.1.3 TSA **Usage**

TSA[1] is available as a command-line tool and Python package. As a Python package, TSA can be used as a library that provides classes for sniffing network communication, parsing network traces and extracting features, quantifying information leakage and visualizing the feature distributions and information leakage. Defining input models, mutators, system instrumentation and providing seed inputs require writing them in Python, therefore this is the *recommended* way to use TSA for *feedback-driven analysis*. We provide examples on how the TSA is used as a Python package in our repository.

TSA's command-line interface is used for analyzing already captured network traces. TSA's command-line arguments include network trace and label file names, which ports to examine for filtering traffic, and folder location for generated plots. There are flags for choosing the leakage quantification options, choosing whether to use alignment on network traces, and whether to quantify only space or time features if the user is interested in only one of them.

In both usages, TSA outputs the leakage information as a ranking over the extracted features. If requested, TSA also provides plots for the feature distributions per secret to show how features and secrets correlate.

6.2 TSA **Evaluation**

To demonstate the performance of our approach when input sets and mutators are given, we used TSA to analyze information leakages of 2 applications in the DARPA STAC benchmark [41] which contain implementations of various client-server or peer-to-peer web applications such

[1]The tool's source code, experimental evaluation code, evaluation results, and documentation are publicly available at `https://github.com/kadron/tsa-tool`

as a messaging app, a peer-to-peer bidding application, a railyard or air traffic management system. The applications we analyzed are AIRPLAN and RAILYARD. This benchmark has multiple versions of each application where some versions are found to be leaking information through manual analysis. This provides some coarse ground truth where we can compare our leakage results against that ground truth. We also used TSA to analyze an IoT benchmark generated by researchers from University of New South Wales (UNSW) Sydney [79] where the task is to identify the device by observing the trace. The traces were already provided in this benchmark, thus we used TSA to just analyze the traces and not generate new traces. We report the leakages in terms of percentages and amount of bits leaked compared with the full amount of information of the secret set.

AIRPLAN Case Study. AIRPLAN is an air traffic management application where the users of the system can upload route maps describing the connections between certain airports and properties of the connections such as distance, fuel usage, number of passengers per flight, etc. AIRPLAN can also be used to find the ideal path between airports such as the path minimizing fuel usage or maximizing amount of passengers carried. To analyze AIRPLAN, we provided an input model describing the route map graph, denoted the secret as the number of airports (as described in DARPA STAC benchmark), and provided an interaction script which launches the server, logs in to the website, uploads the route map, checks that it is uploaded and logs out. For analysis, we provided 13 seed inputs corresponding to one for each secret value (a graph with 2 nodes, 3 nodes, etc.) and 10 mutators which add/remove nodes, add/remove flights, modify airport names and each weight separately.

Using our tool, we find that AIRPLAN 2, the vulnerable application leaks 100% (3.70 out of 3.70 bits) of the information within 76 minutes of analysis and 3 iterations. AIRPLAN 5 is a modified version of AIRPLAN 2 where the vulnerability is patched and TSA reports that it leaks 89% (3.29/3.70 bits) of the information within 114 minutes of analysis. AIRPLAN 3 is marked not vulnerable in the DARPA STAC benchmark and TSA reports that it leaks 47% (1.74/3.70 bits) of the information within 161 minutes of analysis. These results are consistent with the ground truth where the vulnerable application leaks 100% and other versions have less leakage depending on different versions.

Railyard Case Study. Railyard is a train station management system where the station manager can use the application to provide a description of the train such as the number of cars, list of cargo and crew in each car and the stops that the car will visit. There are different types of cars for specific cargo and crew. The secret of interest in the benchmark is the set of different cargo types. For analysis, we wrote an input structure describing the cars, crew, cargo and stops. We used the TSA API to write an interaction script that sets up the train where the train departs after the specification is set. We provided 64 inputs with different configurations denoting possible combinations of possible cargo and 10 mutators that add/remove a train car, a piece of cargo, a crew member, or a stop. Our analysis shows that this application leaks 22% (1.32/6.00 bits) of the information within 202 minutes which is similar to the coarse ground truth provided by DARPA STAC benchmark where it is marked non-vulnerable. There is no vulnerable version of Railyard in the DARPA STAC benchmark to compare against but 22% leakage shows that the application does not leak a significant amount of information.

UNSW IoT Benchmark Case Study. To demonstrate that TSA can be used to analyze previously obtained network traces, we used TSA on the UNSW IoT Benchmark. In this dataset, each trace is marked the secret, MAC address which matches to the device type used when generating the benchmark. There's no ground truth to the benchmark but previous work found that classifiers can perform with 95% accuracy over the benchmark. [89] TSA reports that the feature with the highest information leakage leaks 58%(2.55/4.39 bits). Another tool, F-BLEAU reports a higher min-entropy leakage with 80%. This is because combining multiple features may increase the information leakage and the original benchmark also reports around 95% classifier accuracy to verify this result.

6.3 Chapter Summary

We presented TSA for automatically detecting and quantifying network side-channel information leakages. In our presentation, we described how TSA works given inputs and mutators, and how its API can be modified to analyze other applications. In our experimental evaluation, we showed TSA's performance against two existing benchmarks.

Chapter 7

Related Work

In this chapter, we summarize the prior work related to the topics of this dissertation. We describe the related work on software and network side-channel analysis, input generation and testing for side-channel analysis, and side-channel mitigation.

Related Work on Software Side-Channel Analysis. One relevant related work uses sequence alignment algorithms on unencrypted packet *payloads* in order to infer similar segments packet contents [90]. This technique applies to the plain-text content of the packets. Our work on the other hand applies sequence alignment algorithms to the packet attributes (time-stamps and packet sizes) for automatically inferring phases of network interactions, and does not assume that packets are unencrypted. Work by Chapman, et. al. illustrates methods for detecting the potential side channels in client-server application traffic [91]. Their approach crawls the given web application to build a model of the system side channel and uses Fisher criterion for quantifying leakage. A different approach by Chen, et. al. focuses differentiating leakage measurements by analyzing state diagrams for web applications [11]. Yet another approach by Mather, et. al. uses a packet-level analysis of network traffic for estimating information leakage for network applications [92]. A number of works present specialized techniques for discovering specific types of vulnerabilities, like identifying the source identity of an HTTP stream [93, 94] or automatically determining network-traffic-based fingerprints for websites [95].

The BLAZER tool [96] also addresses the applications in the DARPA STAC benchmark. Their approach focuses on showing safety properties of non-vulnerable programs but is able to indicate possible side-channel vulnerabilities by detecting observationally imbalanced program branches using a white-box static program analysis approach. Another recent tool called SCANNER has shown success in statically detecting side-channel vulnerabilities in web applications that result from secret-dependent resource usage differences [97]. The tool SIDEBUSTER focuses on side-channel detection and quantification during the software development phase using taint analysis [98]. These three tools all assume access to the source code of the application whereas we use a fully black-box approach. A number of works analyze mobile application for analyzing side-channels in networks of mobile devices [12, 99, 100].

Another line of work relies on formal methods and software verification techniques, like symbolic execution along with model-counting constrain solvers, to statically quantify the amount of information an attacker can gain about a secret in a system [101–104]. These works analyze a variety of attacker models, from active attackers who adaptively query the system to incrementally infer secret information to passive attackers who observe systems which they cannot query, and use methods from quantitative information flow [34, 105, 106] to automatically derive bounds on side-channel information leakage. These are white-box analysis techniques that rely on the ability to symbolically execute a given application.

Chen et al. [11] study side-channel leaks in Web applications using a stateful model that relates transitions between system states to side-channel observables. They show vulnerabilities and look into mitigation costs. They do not provide a tool or quantify leakage. Chapman and Evans [107] present a technique for black-box side channel detection in Web applications by crawling the application and building an automaton. They associate transitions between app states with captured network traffic, and build classifiers to recognize, on future traffic, which transition is likely to have been triggered. They use the Fisher criterion [108] to quantify information leakage based on distinguishability of data points. They use simpler aggregate features, like total size difference or edit distance.

Privacy Oracle [109] finds leaks using differential testing. Like AutoFeed, it is black-box, and it uses alignment to detect meaningful relationships across network traces. But it assumes that network traffic is unencrypted. AutoFeed does not rely on such an assumption: it exploits publicly observable side-channel metadata.

AppScanner [42] is a tool for identifying different apps from encrypted network traces. It is black-box and trains classifiers on traces which can identify which app is being used. They focus on a single type of secret, whereas AutoFeed is a more general tool.

F-BLEAU [43] is a black-box side channel detection and quantification tool that uses k-nearest neighbors estimation to generalize the estimation to unseen data, and min-entropy to quantify information leakage. Leakiest [20] is a tool for side channel detection that uses models based on histograms and KDE, with bandwidth based on std.dev. It provides confidence intervals, but only if there was enough data, which cannot be known until after the analysis.

WeFDE from Li et al. [110] is a website fingerprinting approach which tries to detect which website is visited over the Tor network. WeFDE uses KDE with Monte Carlo sampling to quantify the information leakage over all features. All of these approaches provide information leakage quantification capabilities given extracted features and labeled traces with various probability estimation and different quantification measures. Comparing WeFDE to our approaches, in addition to quantifying the information leakage with a dynamic probability estimation method similar to WeFDE, our approach uses user provided inputs and mutators to automate the input generation and trace capture in a feedback-driven manner.

Related Work on Input Generation, Testing and Feedback-driven Analysis. Fuzzing techniques are popular in security testing. Coverage-guided fuzzing [111–113] can generate complex, structured inputs. Many fuzzing engines use mutation. Some frameworks allow for custom mutators [114]. Others combine fuzzing with symbolic execution [115, 116]. However, coverage-guided fuzzers depend on code instrumentation, and thus require source code. Also, fuzzing engines are generally built toward the goal of breaking the system—that is, finding inputs that cause crashes or assertion violations, rather than quantifying leakage. Fuzzing engines also tend to assume that it is possible to execute the system in milliseconds, while AutoFeed deals with systems that can take many seconds per input.

DifFuzz [16] is a side-channel analysis technique based on differential fuzzing. Like AutoFeed, it involves a feedback loop, but it is white-box. Its evaluation uses manually sliced parts of programs, where crucial classes or methods relevant to the side channel are manually isolated and compiled together with the tool as a program. AutoFeed can analyze unmodified systems, and since it interacts with them at the network level, it can analyze systems written in any language or combination thereof. Other key differences are that AutoFeed handles noise and nondeterminism while DifFuzz assumes determinism and precise measurements, and that AutoFeed quantifies the amount of information leaked.

QFuzz [117], similar to DifFuzz, is a side-channel analysis technique based on fuzzing. It generates new inputs using fuzzing for Java code to explore different program paths and unlike DifFuzz which can only measure side-channel differences between two inputs, QFuzz quantifies the information leakage using min-entropy. It is still a white-box technique and while it has its advantages, the same approach is difficult to apply to software that uses network capabilities.

Work by Bang et al. [15] performs online synthesis of adaptive side-channel attacks. It uses another kind of feedback loop. Like AutoFeed, it profiles the program through the network. However, it is still a white-box technique due to its need to symbolically execute the program before running it. Due to its dependency on symbolic execution, it cannot handle large systems.

Related Work on Side-channel Mitigation. In the network side-channel mitigation area, there have been several proposed mitigation techniques based on packet padding and delaying [72–75]. In the evaluation of Chapter 5, we compared our work against the proposed techniques and experimentally demonstrated that our work synthesizes mitigation strategies with better accuracy and overhead due to feature prioritization and refinement on an objective function.

Apthorpe et al.'s work Stochastic Traffic Padding [118] mitigates side-channels caused by a burst of packets (such as a camera uploading a photo, Amazon Echo downloading music files for play, etc.) where the timing of the burst of packets leaks when the event happens. They obfuscate the timing of the event by sending fake bursts of packets over a trace. Their method is tailored on mitigating a specific type of information leakage based on user taking or not taking an action whereas our approach tries to mitigate information leakages in general. The padding method is not publicly available to use and the authors did not respond to our requests for their implementation.

Liu et al.'s work, SniffMislead [119] aims to obfuscate the correlation between network side channels and user actions by simulating dummy users over the network. They capture traces and use classifiers to identify which packets are relevant to the action and replay those packets over the trace as phantom users to confuse any eavesdroppers. Our approach is more general as it can be used to reduce information leakage for device fingerprinting or action fingerprinting as we evaluate approach over a variety of benchmarks. We also provide a tunable mitigation strategy for the privacy and overhead constraints of the user.

Several techniques have been proposed for mitigating timing channels [120–123]. Closest to our timing mitigation approach is a study by Askarov et al. [121]. They investigate techniques for general black-box mitigation of timing side-channels on an event stream and they propose a buffering approach to regularly send the events instead which enforces an information leakage bound on the timing information. In this case, maintaining the quality of service of the communication is important, therefore we explore approaches based on random delays rather than sending packets in regular intervals.

Chapter 8

Conclusion

As we demonstrated throughout the dissertation, network side-channel vulnerabilities are very common in many network applications such as websites, IoT devices, smartphone applications. Most of these vulnerabilities are caused by specific implementations which can be tested and fixed. For example, the IoT example we gave leaked the user action because the actions were relayed with packets containing "open" or "close" strings with unique lengths. Even if this side-channel is fixed by padding packets, implementation details can cause one action to have a different time signature or different packet response. Detecting and quantifying these side-channels require testing various behaviors of the system under test, analyzing the various metadata of the captured traces, quantifying the information leakage accurately, and finding a mitigation strategy that balances the trade-off between quality of service and information leakage. In this dissertation, we provided techniques to address the problem of detecting, quantifying and mitigating side-channels like the example above with various novel techniques.

To address the problem of detecting and quantifying side-channels, we presented Profit. Profit runs the system under test with given inputs to capture the network traffic. It extracts features based on aligned traces and quantifies the information leakage over each feature with probability estimation and information theory techniques. We demonstrated the effectiveness of

Profit on DARPA STAC benchmark where Profit extracts meaningful features on sections of network traces based on trace alignment, ranks the features based on their leakage amount and finds which applications are vulnerable to the side-channel information leakages.

To improve automation in side-channel analysis, we presented AutoFeed, which is a feedback-driven side-channel analysis method. AutoFeed provides a mutation-based input generation technique where in each iteration, AutoFeed generates new inputs using the user-provided input seed set and mutators. AutoFeed guides the mutation based on the amount of leakage in features to explore new behaviors in network traces. To improve leakage quantification, AutoFeed also finds the best fitting probability distribution using kernel density estimation and cross-validation techniques. We showed that AutoFeed is effective in detecting and quantifying information leakages over both canonical examples and DARPA STAC benchmarks.

To quantify the information leakage more accurately, we introduced methods on quantifying information leakage over multiple features. To address the issues of computational cost and probability estimation difficulty of quantifying side-channels over multiple features, we calculate leakage bounds where we use neural networks to estimate probability and sample over the feature space to reduce computational cost of quantification. We also use classifiers to demonstrate side-channel attacks and provide accuracy bounds to estimate classifier performance.

To address the problem of side-channel mitigation, we presented a method, Shark! synthesizes a mitigation strategy balancing network overhead and amount of information leakage. To synthesize the mitigation strategy, Shark! takes user constraints to create an objective function and tries to minimize it by obfuscating certain features by padding and delaying existing packets and injecting dummy packets over the network trace. Once the mitigation strategy is found, we synthesize a function that processes a network traffic as a stream to perform the mitigation strategy. We showed that Shark! synthesizes mitigation strategies with lower leakage and overhead over IoT benchmarks compared to the related work.

To unify all the approaches, we also provide a tool, TSA for the research community. TSA can be used in various ways to either test the application or just analyze captured network traces. We provide a Python library, an API for application testing and a command-line tool for TSA.

While the techniques and tools we presented improve upon the previous related work on side-channel detection, quantification and mitigation, more work can be done to further the improvements. To provide different methods of input generation, grammar-based fuzzers can be used to generate network traces for analysis. This would still require user to provide a grammar but that could be easier for some users. Our work is based on black-box analysis which works well for network applications but it is difficult to figure out the exact source of information leakage in the code. Some white-box analysis techniques such as concolic execution and taint analysis could be done to map network features to specific code segments and automated repair techniques could be used to mitigate side-channels.

Overall, in this final chapter, we would like to point out that building on top of previous work on network side-channel analysis, our contributions have improved state-of-the-art with novel techniques on the network side-channel analysis, input generation and automated testing, and side-channel mitigation. Specifically, we introduced a side-channel analysis approach with more detailed features obtained by trace alignment, an automated testing approach for network side-channels, improved leakage quantification using classifiers, automated mitigation strategy synthesis and combined these approaches into a single tool. We also demonstrated the effectiveness of these side-channel analysis approaches through experimental evaluation on various benchmarks.

Bibliography

[1] E. Geller, *Russian hackers infiltrated podesta's email, security firm says*, Oct, 2016.

[2] D. Shepardson, *Equifax failed to patch security vulnerability in march: former ceo*, Oct, 2017.

[3] V. Goel, *Yahoo says 1 billion user accounts were hacked*, Dec, 2016.

[4] P. C. Kocher, J. Jaffe, and B. Jun, *Differential power analysis*, in *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pp. 388–397, 1999.

[5] K. Gandolfi, C. Mourtel, and F. Olivier, *Electromagnetic analysis: Concrete results*, in *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, no. Generators, pp. 251–261, 2001.

[6] Y. Yarom and K. Falkner, *Flush+ reload: A high resolution, low noise, l3 cache side-channel attack.*, in *USENIX Security Symposium*, vol. 1, pp. 22–25, 2014.

[7] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, *Spectre attacks: Exploiting speculative execution*, *CoRR* **abs/1801.01203** (2018) [arXiv:1801.0120].

[8] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, *Meltdown: Reading kernel memory from user space*, in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pp. 973–990, 2018.

[9] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, *Measuring HTTPS adoption on the web*, in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 1323–1338, USENIX Association, 2017.

[10] D. X. Song, D. A. Wagner, and X. Tian, *Timing analysis of keystrokes and timing attacks on SSH*, in *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA* (D. S. Wallach, ed.), USENIX, 2001.

[11] S. Chen, K. Zhang, R. Wang, and X. Wang, *Side-channel leaks in web applications: A reality today, a challenge tomorrow*, *2010 IEEE Symposium on Security and Privacy (SP)* **00** (2010) 191–206.

[12] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, *Robust smartphone app identification via encrypted network traffic analysis*, *IEEE Transactions on Information Forensics and Security* **13** (Jan, 2018) 63–78.

[13] N. Apthorpe, D. Reisman, and N. Feamster, *A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic*, *arXiv preprint arXiv:1705.06805* (2017).

[14] T. Brennan, S. Saha, T. Bultan, and C. S. Pasareanu, *Symbolic path cost analysis for side-channel detection*, in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pp. 27–37, 2018.

[15] L. Bang, N. Rosner, and T. Bultan, *Online synthesis of adaptive side-channel attacks based on noisy observations*, in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pp. 307–322, 2018.

[16] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, *Diffuzz: Differential fuzzing for side-channel analysis*, *CoRR* **abs/1811.07005** (2018) [arXiv:1811.0700].

[17] T. Brennan, N. Rosner, and T. Bultan, *Jit leaks: Inducing timing side channels through just-in-time compilation*, in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 96–111, 2020.

[18] N. Rosner, I. B. Kadron, L. Bang, and T. Bultan, *Profit: Detecting and quantifying side channels in networked applications*, in *26th Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[19] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, *Analyzing android encrypted network traffic to identify user actions*, *IEEE Transactions on Information Forensics and Security* **11** (2015), no. 1 114–125.

[20] T. Chothia, Y. Kawamoto, and C. Novakovic, *A tool for estimating information leakage*, in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings* (N. Sharygina and H. Veith, eds.), vol. 8044 of *Lecture Notes in Computer Science*, pp. 690–695, Springer, 2013.

[21] I. B. Kadron, N. Rosner, and T. Bultan, *Feedback-driven side-channel analysis for networked applications*, in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[22] Y. Censor, *Pareto optimality in multiobjective problems*, *Applied Mathematics and Optimization* **4** (1977), no. 1 41–59.

[23] DARPA, *Public release items for the DARPA Space-Time Analysis for Cybersecurity (STAC) program*, 2017.

[24] G. Combs *et. al.*, *Wireshark-network protocol analyzer*, *Version 0.99* **5** (2008).

[25] C. Notredame, *Progress in multiple sequence alignment: a survey*, *Pharmacogenomics* **3** (2002), no. 1 131–144.

[26] S. B. Needleman and C. D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, *J. Mol. Biol.* **48** (Mar, 1970) 443–453.

[27] I. Elias, *Settling the intractability of multiple alignment*, *J. Comput. Biol.* **13** (Sep, 2006) 1323–1339.

[28] D. F. Feng and R. F. Doolittle, *Progressive sequence alignment as a prerequisite to correct phylogenetic trees*, *J. Mol. Evol.* **25** (1987), no. 4 351–360.

[29] G. J. Barton and M. J. Sternberg, *A strategy for the rapid multiple alignment of protein sequences. Confidence levels from tertiary structure comparisons*, *J. Mol. Biol.* **198** (Nov, 1987) 327–337.

[30] M. P. Berger and P. J. Munson, *A novel randomized iterative strategy for aligning multiple protein sequences*, *Comput. Appl. Biosci.* **7** (Oct, 1991) 479–484.

[31] M. A. Larkin, G. Blackshields, N. P. Brown, R. Chenna, P. A. McGettigan, H. McWilliam, F. Valentin, I. M. Wallace, A. Wilm, R. Lopez, J. D. Thompson, T. J. Gibson, and D. G. Higgins, *Clustal W and Clustal X version 2.0*, *Bioinformatics* **23** (Nov, 2007) 2947–2948.

[32] C. Notredame, D. G. Higgins, and J. Heringa, *T-Coffee: A novel method for fast and accurate multiple sequence alignment*, *J. Mol. Biol.* **302** (Sep, 2000) 205–217.

[33] K. Katoh, K. Misawa, K. Kuma, and T. Miyata, *Mafft: a novel method for rapid multiple sequence alignment based on fast fourier transform*, *Nucleic Acids Research* **30** (2002), no. 14 3059–3066.

[34] G. Smith, *On the foundations of quantitative information flow*, in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, pp. 288–302, 2009.

[35] C. Shannon, *A mathematical theory of communication*, *Bell System Technical Journal* **27** (July, October, 1948) 379–423, 623–656.

[36] T. M. Cover and J. A. Thomas, *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing).* Wiley-Interscience, 2006.

[37] X. Hong, S. Chen, A. Qatawneh, K. Daqrouq, M. Sheikh, and A. Morfeq, *Sparse probability density function estimation using the minimum integrated square error*, *Neurocomputing* **115** (2013) 122 – 129.

[38] M. Shiga, V. Tangkaratt, and M. Sugiyama, *Direct conditional probability density estimation with sparse feature selection*, *Machine Learning* **100** (Sep, 2015) 161–182.

[39] F. Bunea, R. B. Tsybakov, and M. H. Wegkamp, *Sparse density estimation with l1 penalties*, in *In Proceedings of 20th Annual Conference on Learning Theory (COLT 2007) (2007*, pp. 530–543, Springer-Verlag.

[40] P. Jacob and P. E. Oliveira, *Relative smoothing of discrete distributions with sparse observations*, *Journal of Statistical Computation and Simulation* **81** (2011), no. 1 109–121.

[41] DARPA, *The Space-Time Analysis for Cybersecurity (STAC) program*, 2015.

[42] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, *Robust smartphone app identification via encrypted network traffic analysis*, *IEEE Transactions on Information Forensics and Security* **13** (Jan, 2018) 63–78.

[43] G. Cherubin, K. Chatzikokolakis, and C. Palamidessi, *F-BLEAU: fast black-box leakage estimation*, *CoRR* **abs/1902.01350** (2019) [arXiv:1902.0135].

[44] E. Parzen, *On estimation of a probability density function and mode*, *Ann. Math. Statist.* **33** (09, 1962) 1065–1076.

[45] C. E. Shannon, *A mathematical theory of communication*, *Bell system technical journal* **27** (1948), no. 3 379–423.

[46] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*. Springer, 1986.

[47] M. Rudemo, *Empirical choice of histograms and kernel density estimators*, *Scandinavian Journal of Statistics* (1982) 65–78.

[48] A. W. Bowman, *An alternative method of cross-validation for the smoothing of density estimates*, *Biometrika* **71** (1984), no. 2 353–360.

[49] P. Hall, J. Marron, and B. U. Park, *Smoothed cross-validation*, *Probability theory and related fields* **92** (1992), no. 1 1–20.

[50] P. Burman, *A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods*, *Biometrika* **76** (1989), no. 3 503–514.

[51] W. Zucchini, A. Berzel, and O. Nenadic, *Applied smoothing techniques*, *Part I: Kernel Density Estimation* **15** (2003).

[52] P. Biondi, "Scapy: Packet crafting for Python." `https://scapy.net/`.

[53] 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*, *Journal of Machine Learning Research* **12** (2011) 2825–2830.

[54] "Numpy: scientific computing with Python." `http://www.numpy.org/`.

[55] J. D. Hunter, *Matplotlib: A 2d graphics environment*, *Computing In Science & Engineering* **9** (2007), no. 3 90–95.

[56] "Docker api for Python." `https://docker-py.readthedocs.io/`.

[57] J. H. Friedman, *On bias, variance, 0/1—loss, and the curse-of-dimensionality*, *Data mining and knowledge discovery* **1** (1997), no. 1 55–77.

[58] A. v. d. Oord, Y. Li, and O. Vinyals, *Representation learning with contrastive predictive coding*, *arXiv preprint arXiv:1807.03748* (2018).

[59] M. I. Belghazi, A. Baratin, S. Rajeshwar, S. Ozair, Y. Bengio, A. Courville, and D. Hjelm, *Mutual information neural estimation*, in *International conference on machine learning*, pp. 531–540, PMLR, 2018.

[60] B. Poole, S. Ozair, A. Van Den Oord, A. Alemi, and G. Tucker, *On variational bounds of mutual information*, in *International Conference on Machine Learning*, pp. 5171–5180, PMLR, 2019.

[61] J. R. Quinlan, *Induction of decision trees*, *Machine learning* **1** (1986), no. 1 81–106.

[62] J. Quinlan, *C4. 5: programs for machine learning.* Elsevier, 2014.

[63] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees.* CRC press, 1984.

[64] T. K. Ho, *The random subspace method for constructing decision forests*, *IEEE transactions on pattern analysis and machine intelligence* **20** (1998), no. 8 832–844.

[65] N. S. Altman, *An introduction to kernel and nearest-neighbor nonparametric regression*, *The American Statistician* **46** (1992), no. 3 175–185.

[66] J. Schmidhuber, *Deep learning in neural networks: An overview*, *Neural networks* **61** (2015) 85–117.

[67] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, *Efficient backprop*, in *Neural networks: Tricks of the trade*, pp. 9–48. Springer, 2012.

[68] X. Glorot, A. Bordes, and Y. Bengio, *Deep sparse rectifier neural networks*, in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323, 2011.

[69] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference, and prediction.* Springer Science & Business Media, 2009.

[70] F. A. Alaba, M. Othman, I. A. T. Hashem, and F. Alotaibi, *Internet of things security: A survey*, *Journal of Network and Computer Applications* **88** (2017) 10–28.

[71] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, *Demystifying iot security: an exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations*, *IEEE Communications Surveys & Tutorials* **21** (2019), no. 3 2702–2733.

[72] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, *Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail*, in *2012 IEEE symposium on security and privacy*, pp. 332–346, IEEE, 2012.

[73] A. J. Pinheiro, P. F. de Araujo-Filho, J. d. M. Bezerra, and D. R. Campelo, *Adaptive packet padding approach for smart home networks: A tradeoff between privacy and performance*, *IEEE Internet of Things Journal* **8** (2020), no. 5 3930–3938.

[74] S. Xiong, A. D. Sarwate, and N. B. Mandayam, *Defending against packet-size side-channel attacks in iot networks*, in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2027–2031, IEEE, 2018.

[75] M. Uddin, T. Nadeem, and S. Nukavarapu, *Extreme sdn framework for iot and mobile applications flexible privacy at the edge*, in *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom*, pp. 1–11, IEEE, 2019.

[76] A. Diquet, "Ssl kill switch 2." `https://github.com/nabla-c0d3/ssl-kill-switch2`, 2020.

[77] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, *mitmproxy: A free and open source interactive HTTPS proxy*, 2010–. [Version 5].

[78] M. R. Shahid, G. Blanc, Z. Zhang, and H. Debar, *Iot devices recognition through network traffic analysis*, in *2018 IEEE International Conference on Big Data (Big Data)*, pp. 5187–5192, IEEE, 2018.

[79] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, *Classifying iot devices in smart environments using network traffic characteristics*, *IEEE Transactions on Mobile Computing* **18** (2018), no. 8 1745–1759.

[80] A. J. Pinheiro, J. d. M. Bezerra, C. A. Burgardt, and D. R. Campelo, *Identifying iot devices and events based on packet length from encrypted traffic*, *Computer Communications* **144** (2019) 8–17.

[81] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky, *Packet-level signatures for smart home devices*, in *Network and Distributed Systems Security (NDSS) Symposium*, vol. 2020, 2020.

[82] *grpc concepts*, 2020-03-18.

[83] *Mqtt version 5.0 oasis standard*, 2019-03-07.

[84] “STOMP Protocol Specification, Version 1.2.”

[85] N. Naik, *Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP*, in *2017 IEEE International Systems Engineering Symposium (ISSE)*, (Vienna, Austria), pp. 1–7, IEEE, Oct., 2017.

[86] Anonymous, “Iot benchmark applications.” `https://anonymous.4open.science/r/30c5353a-0802-446b-82f8-5debf7fc08ec/`, 2020.

[87] A. Tharwat, *Classification assessment methods*, *Applied Computing and Informatics* (2020).

[88] I. Docker, “Docker SDK and API.”

[89] A. J. Pinheiro, J. M. Bezerra, and D. R. Campelo, *Packet padding for improving privacy in consumer iot*, in *2018 IEEE Symposium on Computers and Communications (ISCC)*, pp. 00925–00929, IEEE, 2018.

[90] O. Esoul and N. Walkinshaw, *Using segment-based alignment to extract packet structures from network traces*, in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 398–409, July, 2017.

[91] 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*, CCS ’11, (New York, NY, USA), pp. 263–274, ACM, 2011.

[92] L. Mather and E. Oswald, *Quantifying side-channel information leakage from web applications*, *IACR Cryptology ePrint Archive* **2012** (2012) 269.

[93] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine, *Privacy Vulnerabilities in Encrypted HTTP Streams*, pp. 1–11. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[94] M. Liberatore and B. N. Levine, *Inferring the source of encrypted http connections*, in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS ’06, (New York, NY, USA), pp. 255–263, ACM, 2006.

[95] A. Hintz, *Fingerprinting Websites Using Traffic Analysis*, pp. 171–178. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[96] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, *Decomposition instead of self-composition for proving the absence of timing channels*, in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pp. 362–375, 2017.

[97] J. Chen, O. Olivo, I. Dillig, and C. Lin, *Static detection of asymptotic resource side-channel vulnerabilities in web applications*, in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, (Piscataway, NJ, USA), pp. 229–239, IEEE Press, 2017.

[98] 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*, CCS '10, (New York, NY, USA), pp. 595–606, ACM, 2010.

[99] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, *Analyzing android encrypted network traffic to identify user actions*, *IEEE Transactions on Information Forensics and Security* **11** (Jan, 2016) 114–125.

[100] M. Conti, Q. Li, A. Maragno, and R. Spolaor, *The dark side(-channel) of mobile devices: A survey on network traffic analysis*, *CoRR* **abs/1708.03766** (2017) [arXiv:1708.0376].

[101] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, *Symbolic quantitative information flow*, *SIGSOFT Softw. Eng. Notes* **37** (Nov., 2012) 1–5.

[102] Q. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, *Synthesis of adaptive side-channel attacks*, in *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pp. 328–342, 2017.

[103] X. Huang and P. Malacaria, *Sideauto: quantitative information flow for side-channel leakage in web applications*, in *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society, WPES 2013, Berlin, Germany, November 4, 2013*, pp. 285–290, 2013.

[104] Q. Phan, P. Malacaria, C. S. Pasareanu, and M. d'Amorim, *Quantifying information leaks using reliability analysis*, in *2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, July 21-23, 2014*, pp. 105–108, 2014.

[105] D. Clark, S. Hunt, and P. Malacaria, *Quantitative analysis of the leakage of confidential data*, *Electr. Notes Theor. Comput. Sci.* **59** (2001), no. 3 238–251.

[106] F. Biondi, A. Legay, B. F. Nielsen, P. Malacaria, and A. Wasowski, *Information leakage of non-terminating processes*, in *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, pp. 517–529, 2014.

[107] 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*, CCS '11, (New York, NY, USA), pp. 263–274, ACM, 2011.

[108] R. A. Fisher, *The use of multiple measurements in taxonomic problems*, *Annals of eugenics* **7** (1936), no. 2 179–188.

[109] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, *Privacy oracle: A system for finding application leaks with black box differential testing*, in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, (New York, NY, USA), pp. 279–288, ACM, 2008.

[110] S. Li, H. Guo, and N. Hopper, *Measuring information leakage in website fingerprinting attacks and defenses*, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (D. Lie, M. Mannan, M. Backes, and X. Wang, eds.), pp. 1977–1992, ACM, 2018.

[111] lcamtuf, "American Fuzzy Lop."

[112] K. Serebryany, *libFuzzer, a library for coverage-guided fuzz testing*, *LLVM project* (2015).

[113] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *Generating software tests*, in *Generating Software Tests.* Saarland University, 2019. Retrieved 2019-01-14 00:29:35-08:00.

[114] PeachTech, "PeachFuzzer."

[115] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Driller: Augmenting fuzzing through selective symbolic execution.*, in *NDSS*, vol. 16, pp. 1–16, 2016.

[116] Y. Noller, R. Kersten, and C. S. Păsăreanu, *Badger: Complexity analysis with fuzzing and symbolic execution*, in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, (New York, NY, USA), pp. 322–332, ACM, 2018.

[117] Y. Noller and S. Tizpaz-Niari, *Qfuzz: quantitative fuzzing for side channels*, in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 257–269, 2021.

[118] N. J. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster, *Keeping the smart home private with smart(er) iot traffic shaping*, *Proc. Priv. Enhancing Technol.* **2019** (2019), no. 3 128–148.

[119] X. Liu, Q. Zeng, X. Du, S. L. Valluru, C. Fu, X. Fu, and B. Luo, *Sniffmislead: Non-intrusive privacy protection against wireless packet sniffers in smart homes*, in *24th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 33–47, 2021.

[120] J. Agat, *Transforming out timing leaks*, in *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, (New York, NY, USA), p. 40–53, Association for Computing Machinery, 2000.

[121] A. Askarov, D. Zhang, and A. C. Myers, *Predictive black-box mitigation of timing channels*, in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, (New York, NY, USA), p. 297–307, Association for Computing Machinery, 2010.

[122] W.-M. Hu, *Reducing timing channels with fuzzy time*, in *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 8–20, 1991.

[123] S. Zdancewic and A. Myers, *Observational determinism for concurrent program security*, in *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.*, pp. 29–43, 2003.