Profit: Detecting and Quantifying Side Channels in Networked Applications

Nicolás Rosner*, Ismet Burak Kadron*, Lucas Bang†, and Tevfik Bultan*
*University of California Santa Barbara †Harvey Mudd College
{rosner, kadron, bultan}@cs.ucsb.edu bang@cs.hmc.edu

***Abstract*—We present a black-box, dynamic technique to detect and quantify side-channel information leaks in networked applications that communicate through a TLS-encrypted stream. Given a user-supplied profiling-input suite in which some aspect of the inputs is marked as secret, we run the application over the inputs and capture a collection of variable-length network packet traces. The captured traces give rise to a vast side-channel feature space, including the size and timestamp of each individual packet as well as their aggregations (such as total time, median size, etc.) over every possible subset of packets. Finding the features that leak the most information is a difficult problem.**

Our approach addresses this problem in three steps: 1) Global analysis of traces for their alignment and identification of *phases* across traces; 2) Feature extraction using the identified phases; 3) Information leakage quantification and ranking of features via estimation of probability distribution.

We embody this approach in a tool called Profit and experimentally evaluate it on a benchmark of applications from the DARPA STAC program, which were developed to assess the effectiveness of side-channel analysis techniques. Our experimental results demonstrate that, given suitable profiling-input suites, Profit is successful in automatically detecting information-leaking features in applications, and correctly ordering the strength of the leakage for differently-leaking variants of the same application.

I. Introduction

Our world's professional, governmental, and personal activities are quickly migrating to networked software systems. Standalone systems are becoming an artifact of the past. To mitigate information leakage, most of the top-100 online services are now using SSL/TLS encryption, and its adoption by smaller websites and services is growing at a fast pace [24]. This is a positive step toward avoiding trivial leaks, but 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 (e.g., packet sizes, timings, directions, flags) that can be obtained from message headers and whose patterns may be exploitable as side channels. Side-channel analysis of encrypted network traffic has been used, for example, to gain some knowledge about user keystrokes during SSH connections [46], to identify medical conditions of a patient from the encrypted traffic generated by a healthcare website [12], and to identify which app, among a known set of fingerprinted apps, is being used by a mobile phone user [47].

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23536
www.ndss-symposium.org

Detecting network-based side channels requires searching for correlations between sensitive information that the system accesses (i.e., some kind of secret inputs to the system) and the outputs of the system that are observable over the network. Packet traces are a complex form of output. Each captured trace contains a large number of observable aspects, including not just the size and the timing of each individual packet, but also their aggregations over every possible subset of packets. This results in an intractable number of potential features to investigate for information leakage. Deciding which features to analyze and quantifying the amount of information leaked are challenging problems. We present Profit, a tool that, given a software application and a profiling-input suite, determines whether and how much information leakage occurs in a network stream for a particular secret of interest. Profit uses black-box profiling to build a model of the correlation between the secret value associated with each input and the observable side-channel outputs on the network. It does so by successively running the set of inputs through the system and capturing a set of packet traces (pcaps), each one labeled with the secret value used to produce it. To identify the features that leak information and to quantify the amount of information leaked, Profit uses a three-step approach:

1) *Alignment and Phase Detection:* We use multiple-sequence-alignment techniques from genomics to align the packets of captured traces. By studying recurrent patterns and variations across aligned traces, we automatically identify *phases* in the traces, which often reflect application-level behavior such as a login phase, a file upload phase, etc.
2) *Feature Extraction:* We define a feature space that includes observations about individual packets and about sequences of packets (e.g., the time difference between two packets, the total size of all packets). Phase detection allows us to identify additional features that we would not consider otherwise (e.g., total size of all packets in the third phase). We also extract features from the full original traces.
3) *Information Leakage Quantification:* We use information-theoretic techniques based on Shannon entropy to quantify the amount of information leaked by network observations based on the features identified during phase detection and feature extraction and the automatically inferred probability distributions for features. After quantifying the leakage for each extracted feature, we present the user with a *ranking* of the features that are most worth looking into, sorted by their amount of information leakage.

Combined, these three steps provide a black-box approach for detecting and quantifying information leakage from applications that communicate over an encrypted network stream.

We experimentally evaluate Profit using applications from the DARPA Space/Time Analysis for Cybersecurity (STAC) program [20], which are publicly available [21]. The STAC applications are developed by DARPA to evaluate the effectiveness of side-channel vulnerability detection techniques. They are software systems that

- are of significant size (often hundreds of classes)
- are hard to analyze statically
- are Web-based, client-server, or peer-to-peer
- communicate over TLS-encryped TCP streams
- require user inputs that are often complex and structured (sequences of actions, file uploads, etc.)

Many of these applications contain a side-channel vulnerability that causes the value of some secret variable of interest to be leaked, totally or partially, at the network packet level. DARPA gave us applications that contained and that did not contain vulnerabilities, without telling us which ones did. In some cases they gave us multiple variants of each application, each of which could potentially contain different vulnerabilities that leak different amounts of information about the secret.

Since the challenge has ended, we now know (from our own investigations and from DARPA's official correct answers) whether or not a side-channel vulnerability is present in each of the application variants. We also know the most accurate way to spot each vulnerability on the network. This allows us to evaluate whether the ranking of top-leaking features that Profit generates is consistent with reality. In some cases we also know the relative strength of the vulnerability in different variants, allowing us to evaluate whether Profit's leakage estimations are consistent with reality. For example, a variant of a vulnerable application that contains a mitigated version of a vulnerability should leak less than a variant that contains the same vulnerability in full strength, and leak more than a variant in which the vulnerability is not present at all.

Our experiments with the DARPA STAC benchmark show that, given a suitable profiling-input suite, 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 paper is organized as follows. In Section II we give an overview of our approach and discuss two motivating examples. In Section III we discuss the system model that we use. In Section IV we present the trace alignment and phase detection techniques. In Section V we present the feature extraction and leakage quantification techniques. In Section VI we discuss the experimental evaluation of our approach using the DARPA STAC benchmark. In Section VII we discuss the limitations of our approach. In Section VIII we discuss the related work. In Section IX we conclude the paper.

II. Motivation and Overview

Before we present the technical details of our analysis, we discuss two examples that motivate our approach.

During the challenge we investigated the applications from scratch, with almost no initial knowledge. In each case, we did not know whether a vulnerability was present, and if so, what parts of the code might cause it, which network packets might be affected by it, and how. We were only told which was the secret variable of interest, which gave us some hints regarding which functionalities of the application to consider.

First example

TourPlanner is a client-server system. Given the names of five cities that the user would like to visit, it can compute a Hamiltonian circuit that minimizes certain costs. The secret of interest specified by DARPA is the set of five cities that the user wants to visit. In other words, we want to find out whether someone who can eavesdrop on the TLS-encrypted TCP traffic between client and server would gain information about the set of cities that the user wants to visit. The eavesdropper could build a statistical model by profiling the system beforehand (that is, by interacting with it many times through the network). Armed with such a profile, they could inspect the packets exchanged between another client and the server, and use the statistical model to infer some information about the secret value contained in that interaction.

Suppose that we have a profiling-input suite consisting of many different queries (i.e., many random sets of five cities). For each set in the suite, we send a "compute tour" request to the server and receive the response. To account for network noise, we can execute each input multiple times. All traffic is sniffed and recorded. This black-box profiling yields a series of $\langle$input, output$\rangle$ pairs, where each output is a captured packet trace file (pcap) that contains thousands of observables, including every header of every layer of every packet. Even if we focus only on the most important side-channel observables (such as the time, size and direction of each packet), automatically finding features that have maximum correlation with the secret is a hard problem, since there are too many features.

Figure 1 (left side) shows a sample of 1,000 captured traces that correspond to 100 different inputs, each one executed 10 times. We represent each packet with a bubble. The size of the bubble is proportional to the size of the packet. The x-axis shows the timestamp of each packet. The y-axis denotes the trace number (i-th trace). It is hard to extract any information by looking at the raw traces as they were captured. Even the fact that each input was run multiple times is hard to see.

Figure 1 (right side) shows the same traces after a very simple alignment: add an offset to each trace so that the first packet of every trace occurs at the same time. Now it is clear that each input was repeated multiple times. More importantly, visible patterns emerge which suggest that (i) the first few packets of each trace are not input-dependent, and (ii) some of the inter-packet time differences (*deltas*) might be correlated with the secret input (the set of five cities).

It turns out that, during the computation of the optimal tour, the server sends some progress packets back to the client. The four deltas shown in Figure 2 are affected by the time taken by each step of the computation, and may be used as a fingerprint for the user's query. Each one of the four deltas reveals some information and has some correlation with the secret. When all four delta-values are considered together, the resulting vector in $\mathbb{R}^4$ bears a strong correlation with the secret.

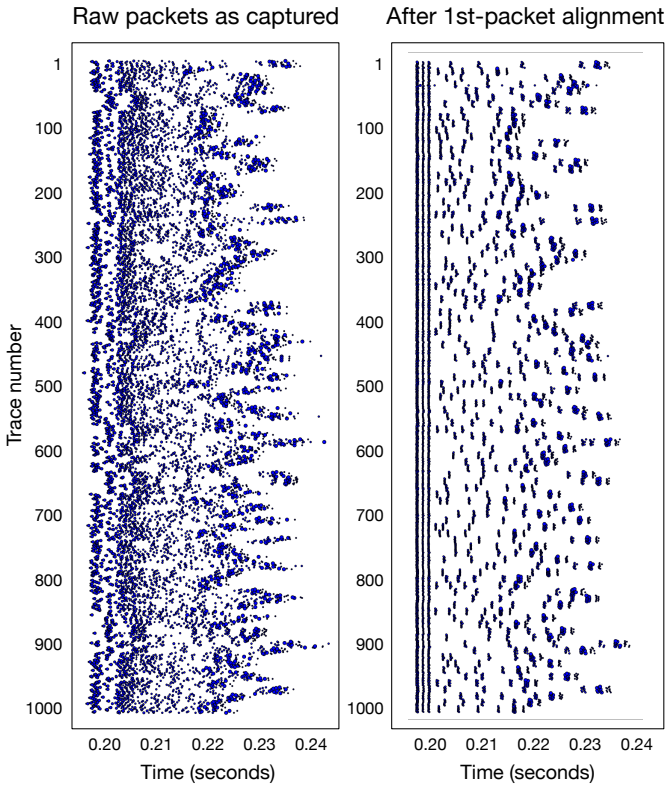


Fig. 1: TOURPLANNER: A sample of 1,000 traces obtained by running 100 different inputs, 10 times each. Bubble size is packet size. Left: raw packets from original pcap file. Right: same data after a simple alignment based on the first packet.

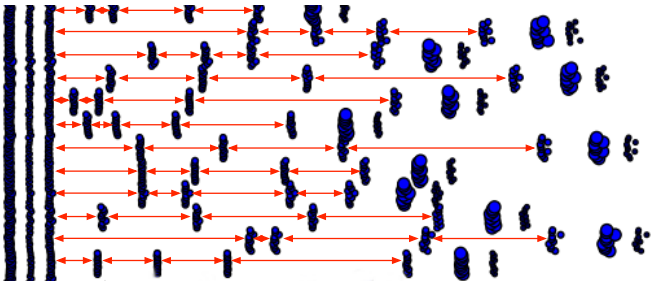


Fig. 2: Right side of Figure 1, magnified for detail. The four time differences shown with red arrows constitute a fingerprint in $\mathbb{R}^4$ that correlates with the secret.

Second example

One basic assumption of this work is that *during profiling*, the Profit user can control the secret value for each execution, even if that value would not normally be publicly visible. This is true in many real-world scenarios. In the most common scenario, the Profit user is a security analyst who is trying to determine whether their own system leaks a certain secret. In a different kind of scenario, an external attacker could build a replica of a system, populate it with their own data, and use it to build a profile that can then be leveraged against the real system. This scenario becomes increasingly likely with the growing trend of running open-source software components on standardized cloud hardware.

GABFEED is another DARPA STAC application. It is a Web-based forum. Users can post messages, search the posted messages, engage in direct chat, etcetera. The server has a private key that is used for authentication purposes.

The following situation arose during our exploration of GABFEED. The secret variable that we were interested in was the Hamming weight (number of ones) of the server's private key. We were studying the following interaction: A user performs a search for something public and then, after an authentication step, performs another search for something nonpublic. (We would later confirm that the delay between two of the network packets that are spawned by this interaction is proportional to the Hamming weight of the binary representation of the server's private key.)

To build a profile, we executed this interaction repeatedly, using different server keys with varying Hamming weight, and captured all network traffic. However, examining pcaps by hand (using a tool like WireShark [16]) is cumbersome. Even when you do have some hypothesis about which packets might be leaking information, verifying it by artisanally skimming through thousands of packets is an extremely tedious task. Considering that we had no hypothesis whatsoever as to which packets might be leaking information, manual inspection of the captured traces would have been a daunting endeavor.

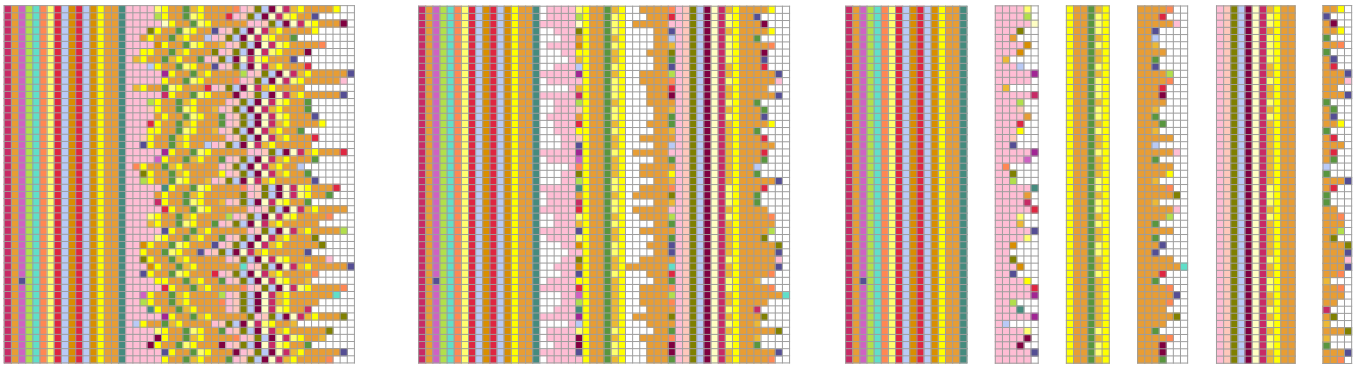
An automated tool that can assist in such a search would need to examine a vast feature space—not just the size of each packet, its flags, and its direction, but also all possible time differences (deltas), all sums of sizes over all possible subsets of packets, etcetera. This is infeasible for all but the shortest network traces. Therefore, an appropriate feature space needs to be automatically selected for further consideration.

Figure 3a shows the network traffic captured by Profit for the GABFEED application for 50 successive interactions, using many different server private keys with 12 different Hamming weights for the key, and different search queries. Each row represents a complete interaction (a captured trace) as a sequence of packet sizes. Colors encode packet sizes and direction. The color palette is intentionally not a gradient in order to keep small variations visible.

In this example, the crucial feature is harder to characterize due to the fact that both search operations introduce a variable number of packets before and after the packet that leaks information. As a consequence of this, even *naming* the crucial feature in terms of the captured traffic becomes difficult, since it is not "the i-th packet" for any consistent value of i.

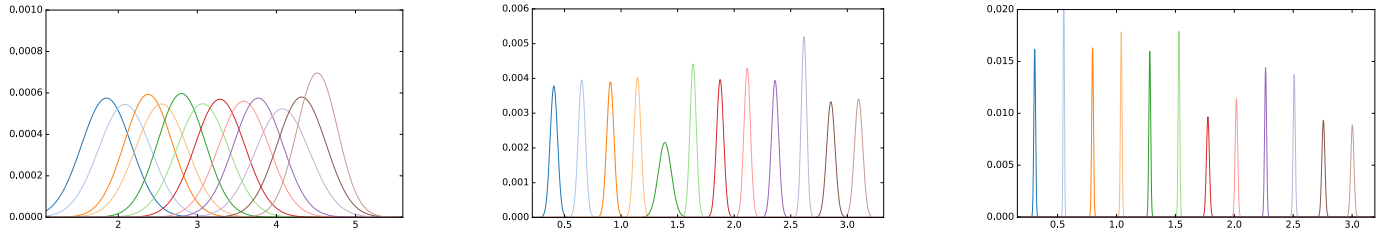
In the general case, interactions with an application often involve a sequence of actions, and some actions spawn a non-constant number of packets. The captured traces can be seen as consisting of several *phases*—subsequences of packets that may correspond to different phases of application-level behavior, e.g., uploading a file and then downloading another one. Recurring patterns that appear across most of the traces can be helpful for automatically detecting phase boundaries.

Figures 3b and 3c show the same 50 GABFEED traces after being globally aligned and then separated into phases,



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

Fig. 3: Trace alignment and phase detection (50 traces shown) for GABFEED. Colors represent different packet sizes.



(a) Feature: Time elapsed between first and last packets of the whole trace.

(b) Feature: Time elapsed between first and last packets of the fifth phase of the trace.

(c) Feature: Time elapsed between packets #3 and #4 of the fifth phase of the trace.

Fig. 4: Probability densities for the top-leaking time-based features in the GABFEED example. X-axis is feature value in seconds. Y-axis is probability. Different curves represent different secret values. Intuitively, features with less overlap between curves leak more information—for a given observation of the feature value, there is less uncertainty about the secret value.

respectively, by Profit. This process enables Profit to synthesize the crucial feature that successfully captures this side channel.

Figure 4 shows one plot for each of the three top-leaking features among the numerous features that were considered by Profit during the leakage quantification step. Each plot shows the probability distributions for each value of the secret, i.e., for each Hamming weight of the private key.

Without knowledge of phases, the best (i.e., most-leaking) feature that Profit can report is the time elapsed between the first and last packets of each trace—that is, the duration of each trace. Since there are 12 different values of the secret (Hamming weight of key), there are $\log_2 12 = 3.58$ 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 4a 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 (the first and last curves, for instance, are almost completely non-overlapping).

With phase knowledge, Profit can consider more refined features, such as the time elapsed between the first and last packets *of the fifth phase*. Figure 4b 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 includes some noise from other packets in that phase, which entails some overlap.

In fact, the top-ranking feature reported by Profit was the time difference *between packets #3 and #4 of the fifth phase*. As illustrated in Figure 4c, this even more specific feature has maximal separation between the distributions of each secret's probability given an observation. This is the only feature that yielded 100% leakage (3.58 of 3.58 bits) according to Profit's feature extraction and leakage quantification.

Figure 5 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 the aligned block into phases. After alignment and phase detection, each network trace is divided into a fixed number of subtraces, where each subtrace 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.

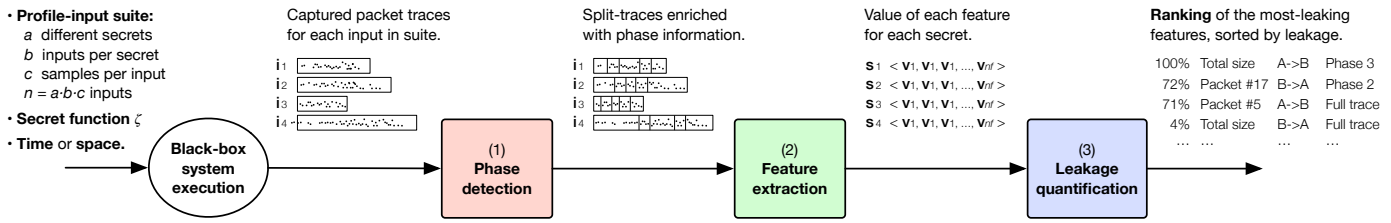


Fig. 5: Profit workflow.

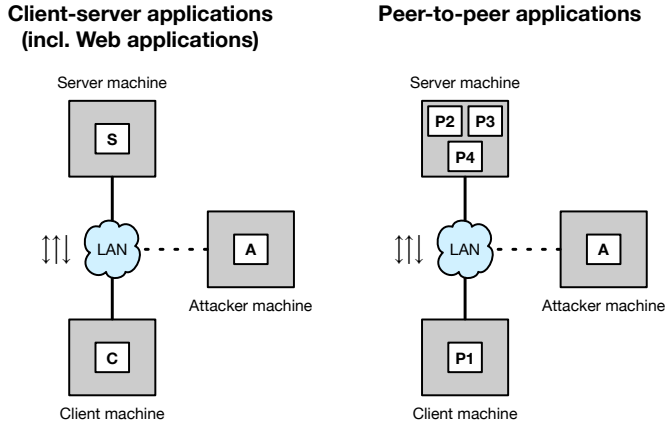


Fig. 6: DARPA STAC configuration and traffic directions.

In 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.

III. 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 value of *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-value-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 itself are outside the scope of this work. However, an attack which attempts to find an encryption key are in scope. 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.

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 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.

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 (see Figure 6). 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, as shown in Figure 6, the lower side denotes a designated peer that runs on the client machine, and the upper side denotes all other peers, running on the server machine. Finally, the notation $t^{\updownarrow}$ is equivalent to just t.

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.

IV. Alignment and phase detection

We now describe our heuristics for trace alignment, which is based on tools from computational biology, and for phase detection, which is based on the aligned traces.

Our goal is to study the patterns that appear across traces, use them to detect potentially meaningful phases, and generate a mechanism that can split any trace into phases accordingly. In other words, we want to generate a trace-splitting function from a collection of traces. Its construction requires examining multiple traces at once, but once constructed, it can be applied to any individual trace. Note that the word *phases* can be used in a global sense, denoting the trace-splitting function obtained from a collection of traces, and in a local sense, denoting the result of splitting one particular trace into phases.

A. 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 use them to identify the *variable* parts in between them, 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 [38] 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 [22], [36]. Many heuristic approaches exist, typically based on progressive methods [25] or iterative refinement [2], [3]. Some popular heuristic toolkits that yield a good compromise between accuracy and execution time are the CLUSTAL [33] family, T-COFFEE [37], and MAFFT [32]. Typically they are limited to 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 in which the alphabet can comprise up to 255 symbols and no biology-specific meaning is attributed to the symbols.

We align the traces based on their sequences of packet sizes, i.e., for a trace $t = \langle p_1, p_2, \ldots, p_{|t|} \rangle$ 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. However, we found it difficult to leverage both 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. Moreover, a size-based alignment can also help find time-based features, as we saw in the GabFeed example from Section II.

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

Fig. 7: A few 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

Fig. 8: Post-alignment matrix (with gaps).

Recall Figure 3 from Section II, which shows 50 traces captured from GabFeed before alignment, after alignment, and after phase splitting. White boxes represent the absence of a packet. Alignment yields a list of sequences of packet sizes in which each sequence may contain *gaps*, shown in white. Gaps are inserted so as to maximize the alignment of patterns that are recurrent across many traces. As a consequence, stable patterns are aligned into columns. The variable patterns located between these columns give rise to meaningful features which would be hard to isolate without alignment.

B. Phase detection

As exemplified by Figure 3 (b), thanks to the inserted gaps, the matrix of aligned sequences presents 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*.

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. Figures 7

and 8 are analogous to Figures 3a and 3b, respectively, but show a much smaller example with shorter sequences. For the example shown in Figure 8, assuming $\omega = 3$ and $\psi = 0.25$, the synthesized pattern would be

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

where int stands for "any integer" and $\star$ 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 used to extract the corresponding groups. The synthesized expression becomes a parser for sequences of packet sizes, and thus an implementation of the trace-splitting function that we wanted to construct.

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.

V. Leakage Quantification

Once the traces have been separated into phases, we apply 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.

A. Feature extraction

Feature extraction is commonly used in leakage quantification and machine learning to extract information from data [47]. In our approach, we process full network traces to obtain subtraces by splitting the full trace using the phase alignment and use the subtraces and full trace to extract features of interest. Each subtrace t (including the full trace) is converted to three subtraces $t^{\uparrow}$, $t^{\downarrow}$ and $t^{\updownarrow}$ according to the direction of packets as explained in Section III. We define a feature set $\mathbb{F} = \{f^1, f^2, \dots, f^n\}$ 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 the same number of features for each sampled trace, we align each subtrace on the left side 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 I. Applying a feature function f^j to each packet series obtained from traces results in a feature profile $P^i = \langle (s_1, v_1^j), (s_2, v_2^j), \dots, (s_n, v_n^j) \rangle$, which is used to compute information leakage.

TABLE I: 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.
$f^{\text{size}}(\langle p_1, \dots, p_n \rangle, i)$	$p_i.\text{size}$	Size of packet i.
$f^{\text{total-time}}(\langle p_1, \dots, p_n \rangle, i)$	$p_n.\text{time} - p_1.\text{time}$	Total time of subtrace.
$f^{\Delta\text{time}}(\langle p_1, \dots, p_n \rangle, i)$	$p_{i+1}.\text{time} - p_i.\text{time}$	Time diff. of packets i & $i+1$.

B. Information theory for quantifying leakage

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} sampled 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 [45]. Specifically, we make use of *Shannon's information entropy* which can be considered a measurement of uncertainty [19], [43].

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

$$\mathcal{H}(S) = -\sum_{s \in \mathbb{S}} p(s) \log_2 p(s) \tag{1}$$

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

$$\mathcal{H}(S|V) = -\sum_{v \in \mathbb{V}} p(v) \sum_{s \in \mathbb{S}} p(s|v) \log_2 p(s|v) \tag{2}$$

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:

$$\mathcal{I}(S;V) = \mathcal{H}(S) - \mathcal{H}(S|V) \tag{3}$$

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 9.

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.

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 $p(s)$. Using $\hat{p}(s|v)$, we apply equations 1, 2, and 3 to compute the estimated information gain (leakage) for the secret given the current feature of interest, $\hat{\mathcal{I}}(S, V)$. In later sections, we refer to the value that Profit computes for $\hat{\mathcal{I}}$ as Leak_H (histogram-based estimation) or Leak_G (Gaussian-based estimation).

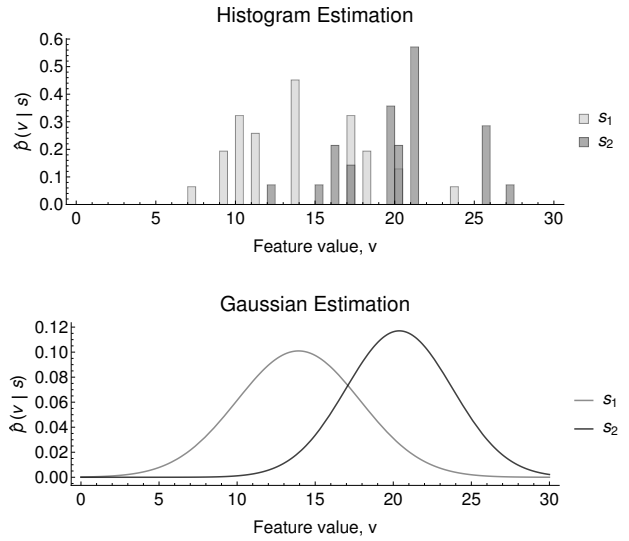


Fig. 9: Estimating a probability distribution from samples using histograms or Gaussian estimates.

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 9. Using histogram-based estimation with a bin-width of $\Delta v = 0.5$ 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 shown in Figure 9, 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 [8], [27], [30], [44]. Our experimental evaluation (Section VI) indicates that Gaussian fitting works well for estimating entropy in network traffic features.

VI. Experimental evaluation

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

A. DARPA STAC systems and vulnerabilities

The applications in our benchmark are from the DARPA Space/Time Analysis for Cybersecurity program [20], 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 paper; we focus on STAC's side-channel-related applications. These STAC applications [21] 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 is 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's offered bid, 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 II, is a Web-based forum where users can post messages, search posted messages, and chat. In GABFEED 2, an authentication 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, the delay between two network packets involved in this authentication is proportional to the Hamming weight of 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.

B. 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 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.

Designing a profiling-input suite 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) Execution cost: The product of the above can spawn a large set of inputs. For some systems, executing all inputs may be costly. Running time vs. coverage is a classic trade-off.

For all the experiments presented in this paper, 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 can be sensitive to parameter tuning. Our data often contains arbitrarily long unalignable regions, so we set MAFFT to the mode recommended for that purpose by its developers (`–genafpair –maxiter 1000 –op 1.5 –ep 0.0`), 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 IV), external alignment sufficed. For input suites with more than 1000 traces, the traces that did not parse due to anomalies (see Section IV-B) were always less than 1%.

Comparison with Leakiest. We have used a leakage quantification tool, Leakiest [14], to compare to our leakage quantification methods. Leakiest computes mutual information between an observable feature and a secret from a set of samples. We provide Leakiest with the same feature-secret pairs as in our methods. We have used Leakiest in its discrete probability estimation mode for space features [10] and continuous probability estimation for time features [13] to calculate $p(y|x)$ and information leakage estimates.

Histogram-based leakage parameters: We used a bin size of 5 for space-based side-channel analyses, and of 0.001 for time-based side-channel analyses.

Implementation details: Profit comprises about 3,000 lines of Python and 400 lines of Mathematica code. It uses Scapy [6] for capturing network packets, Scipy [31] and Scikit-learn [39] for computing information leakage, Matplotlib [29] for plotting probability densities, and MAFFT [32] for trace alignment. We used Mathematica to prototype the trace alignment module and generate the trace alignment visualizations.

C. Experimental results

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

Comparison of different leakage estimation approaches: Figure 10 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 bin sizes, with overfitting taking place in the smallest bin size and underfitting in the 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. We can obtain different values of leakage by changing the bin size; thus, the results are meaningless unless the most accurate bin size is known in advance.

Table VI describes the leakage results obtained over all applications with Gaussian, histogram-based estimation with specific bin sizes and estimation using Leakiest over the best feature that was manually found. For both AIRPLAN 2, AIRPLAN 3 and their variants, the calculated leakage for all three approaches according to the best feature is over 95% for vulnerable AIRPLAN variants and under 91% for non-vulnerable variants. For SNAPBUDDY, both Gaussian and histogram-based approaches reported leakage but Leakiest reported 0% for the best feature. For some other feature, it reported 18% leakage but that is the best result it can find. We attribute this result to Leakiest requiring a lot of samples per secret to be confident of the leakage. With a low number of samples, it underestimates the leakage. For BIDPAL and POWERBROKER, in the variants where the vulnerability is present, Gaussian-based estimation underestimates the leakage and histogram-based method reports a high leakage. In variants where the vulnerability is absent, the histogram-based report overestimates the leakage and report over 90% leakage. We attribute Gaussian reporting lower than expected leakage to low amount of samples we could obtain for this application where the estimated mean and variance is not fully accurate.

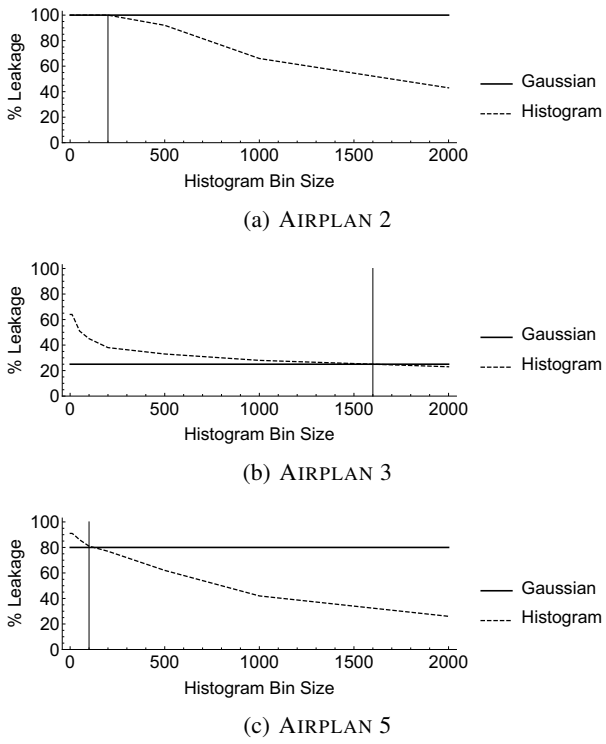
Leakiest reported that it could not run because of low number of samples per secret. For all GABFEED variants and TOURPLANNER, all three approaches report similar leakages.

In comparison, Leakiest works well when the number of samples per secret is high, but we could not use it for applications that take a long time to run, due to the number of samples per secret being too low. For Profit's histogram-based approach, its results are dependent on the parametrization and it overestimates or underestimates in some cases. Gaussian-based estimation underestimates leakage in cases with low number of samples as well, but it is more resilient to those cases than the other two estimation methods. For these reasons, we have used the Gaussian-based estimation in Table VII.

Example of Profit output: Table VIII shows a ranking returned by Profit, where the top-leaking features are shown and ranked according to how much information they leak.

Comparison of best feature and top feature reported by Profit: Table VII 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 II, 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. The *Best feature* column shows the manually found best feature and the $Leak_G$ column shows the percentage of information leakage computed by Profit using Gaussian-based probability estimation for that feature. We have chosen Gaussian-based estimation because of the reasons described in Section VI-C. 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). We will describe the results according to vulnerable applications and groups of vulnerable and non-vulnerable applications using Table VII in the following sections.

Results for vulnerable applications: 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 VI-A, 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 VII, 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



(a) AIRPLAN 2

(b) AIRPLAN 3

(c) AIRPLAN 5

Fig. 10: Information leakage comparison of Gaussian and histogram-based entropy estimation for changing bin sizes for three versions of the AIRPLAN application. Dashed line is Histogram-based leakage estimation result, solid horizontal line is Gaussian-based estimation result. Solid vertical line shows the bin size where two estimation results meet.

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: 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 VII, 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.

VII. LIMITATIONS

Quality of the profiling-input suite: The most important limitation of our approach that the user should keep in mind is that the quality of the leakage quantifications computed

Application	Description	# Classes	# Methods
AIRPLAN	Web-based system for management of airline routes and crew	265	1,483
SNAPBUDDY	Web-based system for image sharing with photo uploads and location tracking	338	2,561
BIDPAL	Peer-to-peer system for management of multiple auctions	251	2,960
GABFEED	Web-based forum with authentication, posting, search, chat	115	409
POWERBROKER	Peer-to-peer system used by power suppliers to buy and sell energy	315	3,445
TOURPLANNER	Client-server system that calculates optimal tours over cities	321	2,742

TABLE II: DARPA STAC systems used in our evaluation (summary).

Application	Secret	Type	Network-level manifestation of the vulnerability (Known by DARPA)
AIRPLAN 2	Number of cities	Space	Total size of the last HTML page sent by server after the *Upload Map* workflow
AIRPLAN 3	Strong connectivity	Space	Size of the third HTML page sent by server after click on *Get Map Properties*
SNAPBUDDY 1	Location of user	Space	Sum of sizes of a few packets (2, 3, or 4) sent by client during *Change location* request
BIDPAL	Secret bid value	Time	Time delta between two of the packets in a server response about bid comparison
GABFEED 1	Server key Hamming wt.	Time	Time delta between two packets in challenge-response authentication
POWERBROKER 1	Price offered	Time	Time delta between two packets in the server response about price comparison
TOURPLANNER	Places to visit	Time	4 timing deltas of 5 consecutive packets during travelling salesman problem calculation

TABLE III: Known network-level manifestation of each vulnerability.

Application	Secret	# Different secrets	# Unique inputs	# Runs per input	# Runs per secret	# Phases detected	# Features found
AIRPLAN 2, 3, 5	Number of cities	13	500	5	192	5	169
AIRPLAN 3, 4	Strong connectivity	2	500	5	1250	5	189
SNAPBUDDY 1	Location of user	294	294	10	10	3	184
BIDPAL 2, 1	Secret bid value	49	49	4	4	5	158
GABFEED 1, 5, 2	No. of 1s in server key	12	60	5	25	2	52
POWERBROKER 1, 2, 4	Price offered	49	49	4	4	5	184
TOURPLANNER	Places to visit	250	250	20	20	4	62

TABLE IV: Execution of each system. Number of phases detected and features obtained.

Application	Network-level manifestation of the vulnerability (Known by DARPA)	Best feature (manually found)
AIRPLAN 2	Total size of the last HTML page sent by server after the *Upload Map* workflow	Sum $\downarrow$ phase 4
AIRPLAN 3	Size of the third HTML page sent by server after click on *Get Map Properties*	Pkt 10 $\downarrow$ phase 3
SNAPBUDDY 1	Sum of sizes of a few packets (2, 3, or 4) sent by client during *Change location* request	Sum $\uparrow$ phase 2
BIDPAL 2	Time delta between two of the packets in a server response about bid comparison	Δ 19-20 $\downarrow$ full trace
GABFEED 1	Time delta between two packets in challenge-response authentication	Δ 4-5 $\downarrow$ phase 2
POWERBROKER 1	Time delta between two packets in the server response about price comparison	Δ 9-10 $\uparrow$ full trace
TOURPLANNER	4 timing deltas of 5 consecutive packets during travelling salesman problem calculation	Total time $\downarrow$ phase 3

TABLE V: Best feature for each vulnerability. The known network-level manifestation of each vulnerability (Table III) is mapped to the feature space identified by Profit from the traces (Table IV). This mapping was built manually, for evaluation.

Application	Secret	Type	Vulnerability	Best feature (manually found)	Leak_G	Leak_H	Leak_L
AIRPLAN 2	Number of cities	Space	Present	Sum $\downarrow$ phase 4	100%	100%	97%
AIRPLAN 5	Number of cities	Space	Mitigated	Sum $\downarrow$ phase 4	79%	91%	68%
AIRPLAN 3	Number of cities	Space	Absent	Sum $\downarrow$ phase 4	25%	64%	0%
AIRPLAN 3	Strong connectivity	Space	Present	Packet 10 $\downarrow$ phase 3	100%	98%	98%
AIRPLAN 4	Strong connectivity	Space	Absent	Packet 10 $\downarrow$ phase 3	0%	0%	0%
SNAPBUDDY 1	Location of user	Space	Present	Sum $\uparrow$ phase 2	95%	100%	0%
BIDPAL 2	Secret bid value	Time	Present	Δ 19-20 $\downarrow$ full trace	59%	99%	N/A
BIDPAL 1	Secret bid value	Time	Absent	Δ 19-20 $\downarrow$ full trace	9%	92%	N/A
GABFEED 1	Server key Hamming wt.	Time	Present	Δ 6-7 $\downarrow$ full trace	100%	100%	100%
GABFEED 5	Server key Hamming wt.	Time	Absent	Δ 6-7 $\downarrow$ full trace	24%	27%	22%
GABFEED 2	Server key Hamming wt.	Time	Absent	Δ 6-7 $\downarrow$ full trace	19%	26%	21%
POWERBROKER 1	Price offered	Time	Present	Δ 9-10 $\uparrow$ full trace	60%	100%	N/A
POWERBROKER 2	Price offered	Time	Absent	Δ 9-10 $\uparrow$ full trace	13%	95%	N/A
POWERBROKER 4	Price offered	Time	Absent	Δ 9-10 $\uparrow$ full trace	9%	95%	N/A
TOURPLANNER	Places to visit	Time	Present	Total time $\downarrow$ phase 3	30%	48%	27%

TABLE VI: Quantification of leakage for each variant of each applications, obtained by different leakage quantification methods. Leak_H for Histogram-based approach, Leak_G for Gaussian-based approach, Leak_L for Leakiest-based approach)

Application	Secret	Type	Vulnerability	Best feature for vulnerability (manually found)	Leak_G	Top-ranking feature (reported by Profit)	Leak_G
AIRPLAN 2	Number of cities	Space	Present	Sum ↓ phase 4	100%	Sum ↓ phase 4	100%
AIRPLAN 5	Number of cities	Space	Mitigated	Sum ↓ phase 4	79%	Sum ↓ phase 4	79%
AIRPLAN 3	Number of cities	Space	Absent	Sum ↓ phase 4	25%	Packet 20 ↓ full trace	36%
AIRPLAN 3	Strong connectivity	Space	Present	Packet 10 ↓ phase 3	100%	Packet 10 ↓ phase 3	100%
AIRPLAN 4	Strong connectivity	Space	Absent	Packet 10 ↓ phase 3	0%	Packet 1 ↑ phase 2	4%
SNAPBUDDY 1	Location of user	Space	Present	Sum ↑ phase 2	95%	Sum ↑ phase 2	95%
BIDPAL 2	Secret bid value	Time	Present	Δ 19-20 ↓ full trace	59%	Δ 19-20 ↓ full trace	59%
BIDPAL 1	Secret bid value	Time	Absent	Δ 19-20 ↓ full trace	9%	Δ 16-17 ↑ full trace	19%
GABFEED 1	Server key Hamming wt.	Time	Present	Δ 6-7 ↓ full trace	100%	Δ 6-7 ↓ full trace	100%
GABFEED 5	Server key Hamming wt.	Time	Absent	Δ 6-7 ↓ full trace	24%	Δ 6-7 ↓ full trace	24%
GABFEED 2	Server key Hamming wt.	Time	Absent	Δ 6-7 ↓ full trace	19%	Δ 11-12 ↕ full trace	20%
POWERBROKER 1	Price offered	Time	Present	Δ 9-10 ↑ full trace	60%	Total time ↕ full trace	60%
POWERBROKER 2	Price offered	Time	Absent	Δ 9-10 ↑ full trace	13%	Total time ↕ full trace	13%
POWERBROKER 4	Price offered	Time	Absent	Δ 9-10 ↑ full trace	9%	Δ 16-17 ↑ full trace	18%
TOURPLANNER	Places to visit	Time	Present	Total time ↓ phase 3	30%	Total time ↓ phase 3	30%

TABLE VII: Leakage achieved for each variant of each application using the top-ranked feature automatically identified by Profit as leaking the most. Comparison with leakage achieved using the manually identified best feature for that vulnerability. Leakage was computed using Leak_G quantification.

Rank	Feature	Dir.	Subtrace	Leak (%)	Leak (bits)
1	Total size (sum)	↓	Phase 4	79%	2.94 of 3.70
1	Total size (sum)	↕	Phase 4	79%	2.94 of 3.70
1	Total size (sum)	↓	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 VIII: Example of a feature ranking returned by Profit (in this case, for AIRPLAN 5).

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 real probability distribution—that is, the one that we would see if we could afford to execute the whole input domain $\mathbb{I}$. If the input set $\mathbf{I}$ 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: Since we model the probability density function for each secret with a Gaussian curve, we are assuming that, for a given feature, and for each secret, the probability of the feature given the secret should follow an approximately normal distribution. If the user expects significantly different distributions, or if a goodness-of-fit 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 VI-A, 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 which 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 VI-C).

Local area network environment: The network environment and the hardware setup used in this work follow the DARPA STAC reference platform specification. As illustrated in Figure 6, the configuration uses separate machines for client, server, and eavesdropper. They are connected by means of a standard Ethernet local area network. This implies certain favorable conditions, e.g., the network latency, the round-trip time variance, and the rates of phenomena such as packet loss or packet reordering are relatively low compared to those of the public Internet. Nevertheless, as seen in our experimental results, such phenomena are still present—network noise, for instance, is a significant factor in our approach, and needs to be statistically modeled and accounted for. The relatively well-controlled environment of a LAN is a good testbed for new ideas, which can then be adapted to a less protected setting such as the public Internet.

VIII. RELATED WORK

One relevant related paper uses sequence alignment algorithms on the *contents* of unencrypted packets in order to infer the contents of similar segments [23]. This technique applies to the plain-text contents of the packets. Our work, on the other hand, applies sequence alignment algorithms to the visible attributes of encrypted packets in order to automatically detect phases in network interactions, and does not assume that the payloads are unencrypted.

Work by Chapman, et. al. illustrates methods for detecting the potential side channels in client-server application traffic [9]. 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 [12]. Yet another approach by Mather, et. al. uses a packet-level analysis of network traffic for estimating information leakage for network applications [35]. A number of works present specialized techniques for discovering specific types of vulnerabilities, like identifying the source identity of an HTTP stream [7], [34] or automatically determining network-traffic-based fingerprints for websites [26].

In particular, Conti et. al. [17] use a similar set of side-channel packet metadata to infer properties about encrypted network streams. They analyze the shape of sequences of packet sizes, and use machine learning and classification techniques that leverage the dynamic time warping (DTW) [4] metric, which is a type of alignment. However, their approach is specifically tailored to guessing which action (within a set of predefined user actions from a fixed set of smartphone applications) was executed by a user. Our approach is more general and aims at providing support for explorative side-channel analysis of arbitrary applications that communicate over a network stream. Regarding the use of dynamic time warping, we did experiment with DTW in early prototypes of Profit's alignment. However, in the type of analyses that we address, packet traces may contain arbitrarily long unalignable sections between the alignable patterns. We found that DTW is not the right tool for that purpose, which led us to experimenting with multiple sequence alignment [38] instead.

The BLAZER tool [1] 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 [11]. The SIDEBUSTER tool focuses on side-channel detection and quantification during the software development phase using taint analysis [48]. These three tools all assume access to the source code of the application, whereas Profit uses a fully black-box approach. A number of works analyze mobile application for analyzing side-channels in networks of mobile devices [17], [18], [47].

Another line of work relies on formal methods and software verification techniques, like symbolic execution along with model-counting constraint solvers, to statically quantify the amount of information an attacker can gain about a secret in a system [28], [40]–[42]. 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 [5], [15], [45] 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.

IX. CONCLUSIONS AND FUTURE WORK

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 applications demonstrates that Profit is able to identify the features that leak information for the vulnerable application variants. Moreover, Profit is able to correctly order the amount of leakage in different variants of the same application. In the future, we plan to extend Profit with 1) fuzzing techniques for input generation, 2) more flexible ways to estimate the probability distributions of features, such as kernel density estimation, 3) feature reduction techniques for reducing the feature space, and 4) relational analysis to quantify joint information leakage from correlated features.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable feedback, and the MAFFT authors for their assistance. This material is based on research sponsored by DARPA under the agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] 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*, 2017, pp. 362–375.

[2] 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.*, vol. 198, no. 2, pp. 327–337, Nov 1987.

[3] M. P. Berger and P. J. Munson, "A novel randomized iterative strategy for aligning multiple protein sequences," *Comput. Appl. Biosci.*, vol. 7, no. 4, pp. 479–484, Oct 1991.

[4] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series." in *KDD workshop*, vol. 10, no. 16. Seattle, WA, 1994, pp. 359–370.

[5] 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*, 2014, pp. 517–529.

[6] P. Biondi. Scapy: Packet crafting for Python. https://scapy.net/.

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

[8] 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*. Springer-Verlag, 2007, pp. 530–543.

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

[10] K. Chatzikokolakis, T. Chothia, and A. Guha, "Statistical measurement of information leakage," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2010, pp. 390–404.

[11] 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*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 229–239. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155595

[12] 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)*, vol. 00, pp. 191–206, 2010.

[13] T. Chothia and A. Guha, "A statistical test for information leaks using continuous mutual information," in *2011 24th Computer Security Foundations Symposium*. IEEE, 2011, pp. 177–190.

[14] T. Chothia, Y. Kawamoto, and C. Novakovic, "A tool for estimating information leakage," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 690–695.

[15] D. Clark, S. Hunt, and P. Malacaria, "Quantitative analysis of the leakage of confidential data," *Electr. Notes Theor. Comput. Sci.*, vol. 59, no. 3, pp. 238–251, 2001. [Online]. Available: https://doi.org/10.1016/S1571-0661(04)00290-7

[16] G. Combs *et al.*, "Wireshark-network protocol analyzer," *Version 0.99*, vol. 5, 2008.

[17] 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*, vol. 11, no. 1, pp. 114–125, Jan 2016.

[18] M. Conti, Q. Li, A. Maragno, and R. Spolaor, "The dark side(-channel) of mobile devices: A survey on network traffic analysis," *CoRR*, vol. abs/1708.03766, 2017. [Online]. Available: http://arxiv.org/abs/1708.03766

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

[20] DARPA. (2015) The Space-Time Analysis for Cybersecurity (STAC) program. [Online]. Available: http://www.darpa.mil/program/space-time-analysis-for-cybersecurity

[21] DARPA. (2017) Public release items for the DARPA Space-Time Analysis for Cybersecurity (STAC) program. [Online]. Available: https://github.com/Apogee-Research/STAC

[22] I. Elias, "Settling the intractability of multiple alignment," *J. Comput. Biol.*, vol. 13, no. 7, pp. 1323–1339, Sep 2006.

[23] 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)*, July 2017, pp. 398–409.

[24] 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: USENIX Association, 2017, pp. 1323–1338. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/felt

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

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

[27] 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*, vol. 115, pp. 122 – 129, 2013.

[28] 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*, 2013, pp. 285–290. [Online]. Available: http://doi.acm.org/10.1145/2517840.2517869

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

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

[31] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[32] 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*, vol. 30, no. 14, pp. 3059–3066, 2002. [Online]. Available: http://dx.doi.org/10.1093/nar/gkf436

[33] 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*, vol. 23, no. 21, pp. 2947–2948, Nov 2007.

[34] 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*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 255–263. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180437

[35] L. Mather and E. Oswald, "Quantifying side-channel information leakage from web applications," *IACR Cryptology ePrint Archive*, vol. 2012, p. 269, 2012. [Online]. Available: http://eprint.iacr.org/2012/269

[36] 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.*, vol. 48, no. 3, pp. 443–453, Mar 1970.

[37] C. Notredame, D. G. Higgins, and J. Heringa, "T-Coffee: A novel method for fast and accurate multiple sequence alignment," *J. Mol. Biol.*, vol. 302, no. 1, pp. 205–217, Sep 2000.

[38] C. Notredame, "Progress in multiple sequence alignment: a survey," *Pharmacogenomics*, vol. 3, no. 1, pp. 131–144, 2002.

[39] 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*, vol. 12, pp. 2825–2830, 2011.

[40] 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*, 2017, pp. 328–342. [Online]. Available: https://doi.org/10.1109/CSF.2017.8

[41] 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*, 2014, pp. 105–108. [Online]. Available: http://doi.acm.org/10.1145/2632362.2632367

[42] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, "Symbolic quantitative information flow," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, Nov. 2012. [Online]. Available: http://doi.acm.org/10.1145/2382756.2382791

[43] C. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, July, October 1948.

[44] M. Shiga, V. Tangkaratt, and M. Sugiyama, "Direct conditional probability density estimation with sparse feature selection," *Machine Learning*, vol. 100, no. 2, pp. 161–182, Sep 2015.

[45] 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)*, 2009, pp. 288–302.

[46] 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. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec01/song.html

[47] 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*, vol. 13, no. 1, pp. 63–78, Jan 2018.

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