Feedback-Driven Side-Channel Analysis for Networked Applications[*]

İsmet Burak Kadron
University of California Santa Barbara
Santa Barbara, CA, USA
kadron@cs.ucsb.edu

Nicolás Rosner
University of California Santa Barbara
Santa Barbara, CA, USA
nrosner@gmail.com

Tevfik Bultan
University of California Santa Barbara
Santa Barbara, CA, USA
bultan@cs.ucsb.edu

ABSTRACT

Information leakage in software systems is a problem of growing importance. Networked applications can leak sensitive information even when they use encryption. For example, some characteristics of network packets, such as their size, timing and direction, are visible even for encrypted traffic. Patterns in these characteristics can be leveraged as side channels to extract information about secret values accessed by the application. In this paper, we present a new tool called AutoFeed for detecting and quantifying information leakage due to side channels in networked software applications. AutoFeed profiles the target system and automatically explores the input space, explores the space of output features that may leak information, quantifies the information leakage, and identifies the top-leaking features.

Given a set of input mutators and a small number of initial inputs provided by the user, AutoFeed iteratively mutates inputs and periodically updates its leakage estimations to identify the features that leak the greatest amount of information about the secret of interest. AutoFeed uses a feedback loop for incremental profiling, and a stopping criterion that terminates the analysis when the leakage estimation for the top-leaking features converges. AutoFeed also automatically assigns weights to mutators in order to focus the search of the input space on exploring dimensions that are relevant to the leakage quantification. Our experimental evaluation on the benchmarks shows that AutoFeed is effective in detecting and quantifying information leaks in networked applications.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; **Web application security**.

KEYWORDS

Side-channel analysis, dynamic program analysis, network traffic analysis, input generation

ACM Reference Format:
İsmet Burak Kadron, Nicolás Rosner, and Tevfik Bultan. 2020. Feedback-Driven Side-Channel Analysis for Networked Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397365

[*]This material is based on research supported by NSF under Grants CCF-1901098 and CCF-1817242.

1 INTRODUCTION

As sensitive information migrates to online services, information leaks are becoming an urgent threat, and *side-channel* leaks, where private information can be extracted by analyzing visible side effects of computation, are becoming increasingly important. Well-known side-channel attacks include those based on power consumption [28], electromagnetic radiation [20], cache timing [48], and CPU-level branch prediction and race conditions, such as the Spectre [26] and Meltdown [30] attacks.

Although information leaks due to design flaws in hardware have been studied more extensively [3, 25, 31], *software-based* side-channels have also been reported [13, 27, 45, 47]. To prevent disastrous software-based side-channel attacks, we need tools that can automatically analyze software systems, detect software side channels, and assess the severity of the information leakage.

Recent work on software side channel detection [4, 8, 11, 32] focuses on white-box techniques. However, the source code of the system may not always be available. Even when it is, many systems do not lend themselves well to white-box analysis. Modern software systems often comprise multiple components joined by networks: clients, servers, peers. They may also include distributed storage, load balancing, microservices, etc. Components are often written in different programming languages, whereas most white-box tools target only one language. White-box tools often require significant manual effort to extract a slice of the system amenable to analysis and mock the rest. Such slicing is costly, error-prone, and may hide side channels. For example, a system may contain a timing vulnerability due to differences in the response time of one component. The side channel goes undetected when that component is stubbed out. This makes a strong case for black-box side-channel analysis tools that execute the whole system without modifications.

We present AutoFeed, a tool for feedback-driven black-box profiling of software systems that detects and quantifies side-channel leakage automatically. 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 [40, 47] and in program traces in general [14, 15]. 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. Our contribution in this paper 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 hyper-parameter 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 [17], 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 paper is organized as follows. In Section 2 we provide motivation and an overview of our approach. In Section 3 we describe the core techniques and heuristics we developed for AutoFeed. In Section 4 we describe our implementation. In Section 5 we present an experimental evaluation of AutoFeed. In Section 6 we discuss the related work. In Section 7 we conclude the paper.

2 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 5.2) 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 [14, 15, 40, 47] 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 [34] 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 (see 4) 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 3.3) 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 5.4). AutoFeed automatically discovers the distribution of observed data using KDE and automatically finds a suitable bandwidth parameter (see 3.4).

By enforcing a stop criterion, AutoFeed automates evaluating whether the leakage estimation is stable enough (see 3.5). This reduces the risks associated with having to manually decide when to stop. It also allows AutoFeed to run analyses batches unattended.

3 FEEDBACK-DRIVEN SIDE-CHANNEL ANALYSIS

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

3.1 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 [40]. 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 $m : \mathbb{I} \longrightarrow \mathbb{I} \cup \{\text{None}\}$ 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 $Leak'$
```

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:    $\vec{W}[i] \leftarrow \sum_j | F'_j - F_j |/(F_{max} - F_{min}) + [Sec'_j \neq Sec_j]$   ▷ 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 $|I_{new}| < N \land |I| > 0$ do
3:    $i \leftarrow$ RandomSelect($I$)   ▷ Select a random input
4:    $M' \leftarrow M$
5:    $done \leftarrow false$
6:    while $M' \neq \emptyset \land \neg done$ do
7:       $m \leftarrow$ RandomSelect($M'$, $\vec{W}$)   ▷ From set $M'$ according to weights $\vec{W}$
8:       $i_{new} \leftarrow m(i)$
9:       if $i_{new} \in I \lor i_{new} =$ 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 $I_{new}$
```

3.2 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 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.

3.3 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.

3.4 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* [43] 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 [40, 43].

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) [34]. 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 [44]. 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 [7, 21, 41]. 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. [9] 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. [50]

3.5 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.

Listing 1: Example usage of AutoFeed API

```
from autofeed 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 ExampleAppInput(Input):
    def __init__(self, num_people, temperature):
        assert num_people >= 0
        self.num_people = num_people
        self.temperature = temperature
    def __eq__(self, other):
        return self.num_people == other.num_people
            and self.temperature == other.temperature
    def __hash__(self):
        return hash((self.num_people, self.temperature))

class IncreaseNumberOfPeople(Mutator):
    def mutate(input):
        input.num_people += 1
        return input

class DecreaseNumber(Mutator):
    def mutate(input):
        if input.num_people > 0:
            input.num_people -= 1
            return input
        else: # we do not allow a negative number of people
            return None

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

4 IMPLEMENTATION

AutoFeed is written in Python. We use the trace-capturing library from [40], which relies on *scapy* [6] for packet sniffing. AutoFeed uses *scikit-learn* [36] and *numpy* [2] for probability estimation and leakage quantification, *matplotlib* [23] for plotting, and the Python *docker* [1] library for container orchestration. We ran AutoFeed on the DARPA STAC Reference Platform, which comprises three Intel NUC computers (see Section 5.3). Users can run AutoFeed on any number of computers and networks, including localhost. AutoFeed provides Python base classes, templates, and examples for:

Orchestration. We use Docker [18] for deployment and launching of components (clients, servers, peers) on different machines. We provide a `Container` abstraction to launch container C on host H, copy a file F to C, run a command K on C, and shut down C.

System setup and execution. The `App` class is a base class that represents a black-box system. Users can subclass `App` to add their own systems. An `App` must provide three things: a `launch` method that launches the system; a `shutdown` method that shuts it down; and a `run` method to execute inputs. Given a list of `Input` instances, the `App.run(inputs)` method should return a list of traces.

Packet sniffing. AutoFeed provides a `Sniffer` object that offers a simple interface for capturing traffic and labeling the captured traces. 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.

Input model. The `Input` base class represents a valid input for an `App`. Users subclass `Input` and add members to model relevant characteristics of an input instance. The only two mandatory methods are `eq` (equality comparator) and `hash`. These are used by AutoFeed to hash previously seen input instances and avoid unwanted repetitions. When in doubt, a simple way to implement a reasonable `hash` method 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 hash code.

Mutators. The `Mutator` base class represents a mutator that transforms valid inputs. The only required method is `mutate`, a static method that takes an `Input` and returns another. The method assumes that the `Input` is valid and must return another valid one. If the mutation cannot be applied or makes no sense for that `Input`, the method should return `None`.

Listing 1 shows an example that uses these classes. For space reasons, the `Input` subclass only has two members. The AutoFeed codebase contains examples with varying degrees of complexity, including apps, inputs, and mutators for the STAC benchmark.

5 EXPERIMENTAL EVALUATION

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 [16], which are publicly available [17]. 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 [40]. We added Railyard because we were interested in modeling its highly structured input format with the mutator-based approach.

5.1 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 3.3 and 3.4. 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 2. 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 2: 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))
```

5.2 STAC Systems

Airplan (265 classes, 1,483 methods) is the airline system from our Section 2 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

Table 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.

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.

5.3 Experimental Setup

We used the DARPA STAC Reference Platform [40], 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 1. The mutators are manually written to modify the secret and various

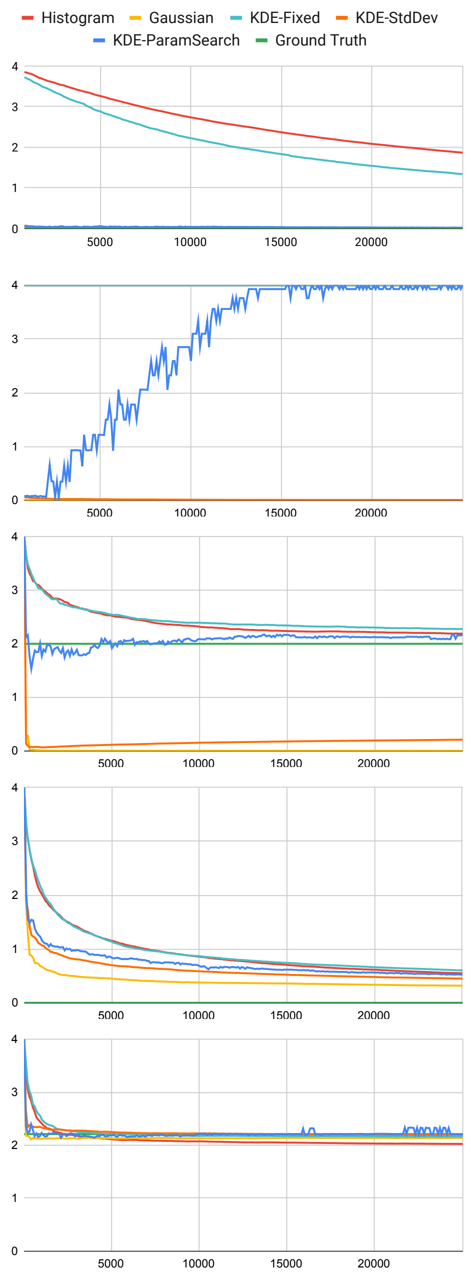

Figure 1: Information leakage results for Examples 1–5 using Gaussian, Histogram and KDE. X-axis shows number of data points. Y-axis shows leakage in bits. Ground truth for the Examples 1–5 are 0 bits, 4.00 bits, 2.00 bits, 0 bits and 2.21 bits respectively.

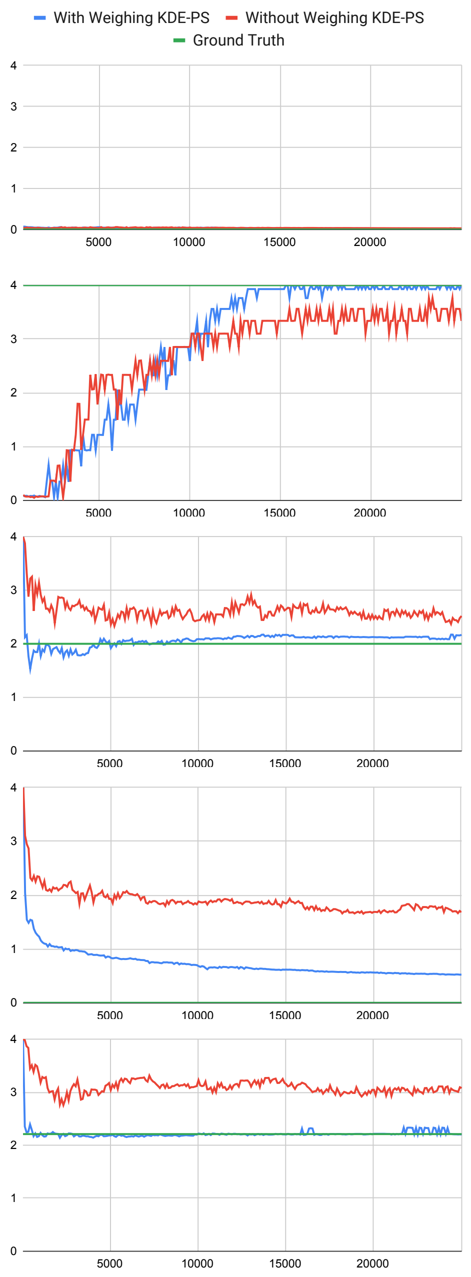

Figure 2: Information leakage results for Examples 1–5 with and without mutator weighing, using KDE-ParamSearch. X-axis shows number of data points. Y-axis is leakage in bits. Ground truth for the Examples 1–5 are 0 bits, 4.00 bits, 2.00 bits, 0 bits and 2.21 bits respectively.

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 [15] and F-BLEAU [14], 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.

5.4 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 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 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 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 **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 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.

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 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.

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. 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 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.

Table 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 ↓	100%	100%	99%	90%	100%	100%	100%	3	76 min.
Airplan 5	Space	Mitigated	Σ Sizes Phase 4 ↓	89%	94%	74%	82%	88%	93%	89%	4	114 min.
Airplan 3	Space	Absent	Size Pkt 20 ↓	46%	33%	21%	20%	45%	35%	47%	6	161 min.
Railyard	Space	Absent	Size Pkt 2 ↕	22%	27%	21%	27%	20%	22%	22%	12	202 min.
SnapBuddy	Space	Present	Σ Sizes Full Trace ↑	100%	100%	47%	100%	100%	100%	100%	3	47 min.
GabFeed 1	Time	Present	Δ Pkt 12-13 ↕	99%	100%	N/A	60%	100%	100%	98%	8	108 min.
GabFeed 2	Time	Absent	Δ Pkt 11-12 ↕	29%	71%	N/A	24%	31%	66%	31%	19	297 min.
GabFeed 5	Time	Absent	Δ Pkt 11-12 ↕	29%	65%	N/A	21%	31%	61%	32%	15	240 min.
PowerBroker 1	Time	Present	Δ Pkt 9-10 ↑	43%	100%	42%	53%	45%	100%	39%	20	313 min.
PowerBroker 2	Time	Absent	Δ Pkt 43-44 ↕	11%	32%	3%	18%	10%	24%	15%	18	263 min.
PowerBroker 4	Time	Absent	Δ Pkt 28-29 ↕	22%	9%	9%	32%	26%	9%	25%	16	220 min.
Bidpal 2	Time	Present	Δ Pkt 28-29 ↕	22%	100%	33%	32%	23%	100%	23%	17	217 min.
Bidpal 1	Time	Absent	Δ Pkt 35-36 ↕	2%	39%	3%	15%	3%	35%	14%	10	137 min.
TourPlanner	Time	Present	Δ Pkt 12-13 ↕	60%	70%	62%	42%	62%	67%	60%	12	2057 min.

Table 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 ↕	98%	97%	96%	96%
GabFeed 2	Time	Absent	Δ Pkt 11-12 ↕	31%	29%	9%	8%
GabFeed 5	Time	Absent	Δ Pkt 11-12 ↕	32%	23%	8%	7%
PowerBroker 1	Time	Present	Δ Pkt 9-10 ↑	39%	39%	37%	36%
PowerBroker 2	Time	Absent	Δ Pkt 43-44 ↕	15%	14%	3%	3%
PowerBroker 4	Time	Absent	Δ Pkt 28-29 ↕	25%	20%	9%	8%
Bidpal 2	Time	Present	Δ Pkt 28-29 ↕	23%	23%	22%	23%
Bidpal 1	Time	Absent	Δ Pkt 35-36 ↕	14%	9%	8%	3%
TourPlanner	Time	Present	Δ Pkt 12-13 ↕	60%	50%	5%	5%

6 RELATED WORK

Profit [40] is a black-box tool to detect and quantify side channels in network traffic. The user must provide the set of inputs to run. This makes the tool impractical. Different input suites yield different results, and the tool offers no way to assess their reliability. An input suite too small or skewed yields incorrect results; an input suite too large and diverse may entail immense wasted effort, making the analysis cost prohibitive. Striking a balance between an insufficient input suite and a wasteful one is tedious, costly, and problem-specific. In contrast, AutoFeed automates this process.

Chen et al. [13] 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 [10] 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 [19] to quantify information leakage based on distinguishability of data points. They use simpler aggregate features, like total size difference or edit distance.

Privacy Oracle [24] 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 [47] 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 [14] 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 [15] 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.

None of the aforementioned black-box tools offer automated input generation. As a consequence, none of them can offer *adaptive* input generation. AutoFeed provides dynamic, adaptive input generation, using feedback-driven self-adjustment to reduce wasted effort and improve the quality of the results obtained.

Of the related work that addresses the problem of software side-channel detection, a significant portion are white-box techniques and tools. As mentioned in Section 1, white-box tools require access to the source code of the target system, and cannot handle systems written in multiple languages or that involve multiple components.

DiffFuzz [32] 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.

CoCoChannel [8] analyzes the control flow graph of the system with respect to an execution cost model. Given a secret of interest, it builds symbolic cost expressions and reduces detecting imbalances to constraint solving. Themis [11] is an end-to-end static analysis tool for side-channel analysis of Java code based on Quantitative Cartesian Hoare Logic. Blazer [4] is a static program analysis tool that can prove the absence of timing side channels by decomposition. Scanner [12] is a static analysis tool for side-channel vulnerability detection in PHP-based Web applications. All of the above are white-box analysis tools that depend on source code, and suffer from the aforementioned limitations.

Several works use symbolic execution to quantify information leakage statically [22, 37–39]. They combine symbolic execution with model counting or quantitative information flow. All of these are white-box and require source code. Furthermore, their scalability is limited by that of symbolic execution. They are also limited to the analysis of systems written in a single language.

Fuzzing techniques are popular in security testing. Coverage-guided fuzzing [29, 42, 49] can generate complex, structured inputs. Many fuzzing engines use mutation. Some frameworks allow for custom mutators [35]. Others combine fuzzing with symbolic execution [33, 46]. 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.

Work by Bang et al. [5] 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.

7 CONCLUSIONS

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. Our experimental evaluation on the benchmarks shows that AutoFeed is effective in automatically detecting and quantifying information leaks.

REFERENCES

[1] [n.d.]. *Docker API for Python.*
[2] [n.d.]. *Numpy: scientific computing with Python.*
[3] Onur Acıiçmez. 2007. Yet another microarchitectural attack:: exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture.* ACM, 11–18.
[4] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. 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.* 362–375. https://doi.org/10.1145/3062341.3062378
[5] Lucas Bang, Nicolás Rosner, and Tevfik Bultan. 2018. 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.* 307–322. https://doi.org/10.1109/EuroSP.2018.00029
[6] Philippe Biondi. [n.d.]. *Scapy: Packet crafting for Python.*
[7] Adrian W Bowman. 1984. An alternative method of cross-validation for the smoothing of density estimates. *Biometrika* 71, 2 (1984), 353–360.
[8] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Pasareanu. 2018. 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.* 27–37. https://doi.org/10.1145/3213846.3213867
[9] Prabir Burman. 1989. A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods. *Biometrika* 76, 3 (1989), 503–514.
[10] Peter Chapman and David Evans. 2011. Automated Black-box Detection of Side-channel Vulnerabilities in Web Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11).* ACM, New York, NY, USA, 263–274. https://doi.org/10.1145/2046707.2046737
[11] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* 875–890. https://doi.org/10.1145/3133956.3134058
[12] Jia Chen, Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2017. Static Detection of Asymptotic Resource Side-channel Vulnerabilities in Web Applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017).* IEEE Press, Piscataway, NJ, USA, 229–239. http://dl.acm.org/citation.cfm?id=3155562.3155595
[13] Shuo Chen, Kehuan Zhang, Rui Wang, and XiaoFeng Wang. 2010. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. *2010 IEEE Symposium on Security and Privacy (SP)* 00 (2010), 191–206. https://doi.org/doi.ieeecomputersociety.org/10.1109/SP.2010.20
[14] Giovanni Cherubin, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. 2019. F-BLEAU: Fast Black-box Leakage Estimation. *CoRR* abs/1902.01350 (2019). arXiv:1902.01350 http://arxiv.org/abs/1902.01350
[15] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 2013. A Tool for Estimating Information Leakage. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 690–695. https://doi.org/10.1007/978-3-642-39799-8_47
[16] DARPA. 2015. *The Space-Time Analysis for Cybersecurity (STAC) program.* http://www.darpa.mil/program/space-time-analysis-for-cybersecurity
[17] DARPA. 2017. *Public release items for the DARPA Space-Time Analysis for Cybersecurity (STAC) program.* https://github.com/Apogee-Research/STAC
[18] Inc. Docker. [n.d.]. Docker. https://www.docker.com/
[19] Ronald A Fisher. 1936. The use of multiple measurements in taxonomic problems. *Annals of eugenics* 7, 2 (1936), 179–188.
[20] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings.* 251–261. https://doi.org/10.1007/3-540-44709-1_21
[21] Peter Hall, JS Marron, and Byeong U Park. 1992. Smoothed cross-validation. *Probability theory and related fields* 92, 1 (1992), 1–20.
[22] Xujing Huang and Pasquale Malacaria. 2013. 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.* 285–290. https://doi.org/10.1145/2517840.2517869
[23] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

[24] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. 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* (Alexandria, Virginia, USA) *(CCS '08)*. ACM, New York, NY, USA, 279–288. https://doi.org/10.1145/1455770.1455806

[25] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. 1998. Side channel cryptanalysis of product ciphers. In *European Symposium on Research in Computer Security*. Springer, 97–110.

[26] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[27] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. Springer, 104–113.

[28] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. 388–397. https://doi.org/10.1007/3-540-48405-1_25

[29] lcamtuf. [n.d.]. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/

[30] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[31] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. 1999. Investigations of Power Analysis Attacks on Smartcards. *Smartcard* 99 (1999), 151–161.

[32] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2018. DifFuzz: Differential Fuzzing for Side-Channel Analysis. *CoRR* abs/1811.07005 (2018). arXiv:1811.07005 http://arxiv.org/abs/1811.07005

[33] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. 2018. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 322–332. https://doi.org/10.1145/3213846.3213868

[34] Emanuel Parzen. 1962. On Estimation of a Probability Density Function and Mode. *Ann. Math. Statist.* 33, 3 (09 1962), 1065–1076. https://doi.org/10.1214/aoms/1177704472

[35] PeachTech. [n.d.]. PeachFuzzer. http://www.peach.tech/

[36] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[37] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. 328–342. https://doi.org/10.1109/CSF.2017.8

[38] Quoc-Sang Phan, Pasquale Malacaria, Corina S. Pasareanu, and Marcelo d'Amorim. 2014. 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*. 105–108. https://doi.org/10.1145/2632362.2632367

[39] Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. 2012. Symbolic Quantitative Information Flow. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5. https://doi.org/10.1145/2382756.2382791

[40] Nicolás Rosner, Ismet Burak Kadron, Lucas Bang, and Tevfik Bultan. 2019. 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*.

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

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

[43] Claude E Shannon. 1948. A mathematical theory of communication. *Bell system technical journal* 27, 3 (1948), 379–423.

[44] Bernard W. Silverman. 1986. *Density Estimation for Statistics and Data Analysis*. Springer. https://doi.org/10.1007/978-1-4899-3324-9

[45] Dawn Xiaodong Song, David A. Wagner, and Xuqing Tian. 2001. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*. http://www.usenix.org/publications/library/proceedings/sec01/song.html

[46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[47] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. 2018. Robust Smartphone App Identification via Encrypted Network Traffic Analysis. *IEEE Transactions on Information Forensics and Security* 13, 1 (Jan 2018), 63–78. https://doi.org/10.1109/TIFS.2017.2737970

[48] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*, Vol. 1. 22–25.

[49] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. Generating Software Tests. In *Generating Software Tests*. Saarland University. https://www.fuzzingbook.org/ Retrieved 2019-01-14 00:29:35-08:00.

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