Targeted Black-Box Side-Channel Mitigation for IoT*

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

Chaofan Shou
shou@cs.ucsb.edu
University of California Santa Barbara
Santa Barbara, CA, USA

Emily O'Mahony
emilyomahony@ucsb.edu
University of California Santa Barbara
Santa Barbara, CA, USA

Yilmaz Vural
yilmazvural@cs.ucsb.edu
University of California Santa Barbara
Santa Barbara, CA, USA

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

ABSTRACT

In this paper we present techniques for generating targeted mitigation strategies for network side-channel vulnerabilities in IoT applications. Our tool IoTPatch profiles the target IoT application by capturing the network traffic and labeling the network traces with the corresponding user actions. It extracts features such as packet sizes and times from the captured traces, and quantifies the information leakage by modeling the distribution of feature values. In order to mitigate the side-channel vulnerabilities, IoTPatch uses the information leakage measure over features to prioritize specific features and synthesizes a packet padding and delaying strategy based on an objective function for minimizing information leakage and time and space overhead. IoTPatch provides a tunable mitigation strategy where the trade-off between the information leakage and performance overhead can be adjusted to accommodate needs of different applications. We evaluate IoTPatch on three network benchmarks and demonstrate that IoTPatch can discover and quantify the information leakage and synthesize a set of Pareto optimal mitigation strategies performing better than the prior work in terms of reducing leakage and overhead.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; **Web application security**; **Network security**; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Side-channel analysis, Network traffic analysis, Internet of Things

ACM Reference Format:
Ismet Burak Kadron, Chaofan Shou, Emily O'Mahony, Yilmaz Vural, and Tevfik Bultan. 2022. Targeted Black-Box Side-Channel Mitigation for IoT. In *Proceedings of the 12th International Conference on the Internet of Things (IoT '22), November 7–10, 2022, Delft, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3567445.3567447

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

This work is licensed under a Creative Commons Attribution International 4.0 License.

IoT '22, November 7–10, 2022, Delft, Netherlands
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9665-3/22/11.
https://doi.org/10.1145/3567445.3567447

1 INTRODUCTION

Internet of Things (IoT) devices are becoming more popular with increasing internet connectivity and bandwidth and they allow users to control or get information from sensors or appliances such as motion sensors, smart locks, and smart lights via smartphone apps. Although IoT devices present many benefits, they also carry the risk of being vulnerable to malicious actors [4]. Even with encryption, eavesdroppers can monitor the network traffic and use the metadata of the traffic, such as the amount of bytes transmitted or duration of transmission, to leak information about user actions. A sleep sensor that sends more packets when the user is awake reveals the sleeping habits of its user [7] or device type can be revealed via network traffic characteristics [24]. These types of information leaks due to non-functional characteristics of computer systems are called side-channels, and in this paper, we investigate mitigating the side-channel vulnerabilities in IoT applications due to network traffic.

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

We propose a black-box side channel analysis tool called IoTPatch to mitigate side-channel vulnerabilities in IoT applications. As described in Figure 1, IoTPatch works by collecting encrypted network traces and labeling them with the secret value of the corresponding trace (which can be user actions or a device state such as a motion sensor's status at the time of the trace capture). IoTPatch analyzes the traces by extracting features such as packet sizes and timings, and quantifies the information leakage. Using the information leakage quantification, IoTPatch prioritizes which features to target and iteratively develop a mitigation strategy with respect to a tunable objective function balancing the information leakage reduction and overhead. The tunable objective function can be customized by the user by changing the parameter that controls the trade-off between information leakage and mitigation overhead. This enables IoTPatch to synthesize a set of Pareto optimal [10] mitigation strategies corresponding to different trade-offs between privacy and performance.

Compared to prior work on network side-channel mitigation approaches [11, 20, 26, 27], we present the following novel contributions in this paper:

(1) A method to prioritize the features for mitigation using metrics measuring information leakage, targeting the packets related with features that leak the most information (Sections 3.1, 3.2).
(2) A search-based, tunable side-channel vulnerability mitigation method which finds the optimal packet padding and delaying strategy for mitigating both space and time network side-channels while keeping the overhead low, with the ability to adjust the trade-off between the leakage and the mitigation overhead (Section 4).
(3) Implementation and experimental evaluation of the novel targeted black-box network side-channel mitigation approach we present in this paper (Section 5).

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

The rest of the paper is organized as follows. In Section 2, we give an overview of the network structure and protocols for the IoT applications, our system model and assumptions on the attacker behavior. In Section 3, we go over the feature extraction and prioritization methods. In Section 4, we go over the targeted mitigation technique. In Section 5, we discuss the implementation details, IoT benchmarks, experimental evaluation of our approach, and its limitations. In Section 6, we discuss the related work. In Section 7, we conclude the paper.

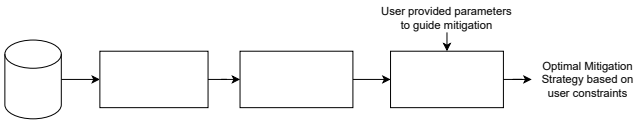

Figure 1: IoTPatch workflow describing the main building blocks of our approach.

2 IOT NETWORK MODEL

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

There are several IoT protocols for communicating over the internet. gRPC [2] is a remote procedure call (RPC) framework based on HTTP/2.0. It encodes data with Protobuf, a serialization library that converts objects to binary streams. MQTT [1] is an open-source protocol standard designed to support communications between two or more separated devices. It has several implementations like RabbitMQ, HiveMQ, and AWS IoT. Since no default serialization method is provided, developers commonly conduct JSON serialization on objects sent in request and response. STOMP [3] is an alternative to MQTT, commonly used in low-energy devices as it provides minimal functionality. Similar to MQTT, developers have full freedom on deciding which serialization methods to use. Other protocols such as Websocket, AMQP, JMS, and CoAP are also commonly used in the IoT ecosystem. However, since they share most patterns of the aforementioned protocols [16], in our experimental evaluation, we only focused on the three protocols listed above. Our side-channel analysis technique is applicable to applications written using any of these communication protocols.

Network packets and network traces. Our representation of a packet is an abstraction of the actual TCP/IP network packets. Network packets consist of layers and headers that contain information to help with the transportation of the packet over the internet. In our abstraction, a packet p has a timestamp field (p.time), denoting the time it was sent, a payload size field (p.size), denoting the size of its content in bytes, and its direction as the source and destination IPs and ports (p.src, p.dst, p.sport, p.dport). The packet contents are encrypted in the UDP/TCP layer and not visible (since we are not considering crypto attacks), therefore we do not use packet payload content in our abstraction.

We represent each packet p as a tuple that consists of packet meta-data: $p = (p.\text{time}, p.\text{size}, p.\text{src}, p.\text{dst}, p.\text{sport}, p.\text{dport})$

A network trace t is a list of packets sorted in the ascending order of packet times. A network trace can also be split into smaller traces based on packet direction or other constraints for analysis. Each trace t consists of an ordered sequence of packets, $t = \langle p_1, p_2, ..., p_{|t|} \rangle$.

We also label the traces based on user interactions such as the action the user is taking or the device type to find the correlation between them. When a developer or an attacker wants to profile the IoT system for analysis, they need to run each possible action multiple times and obtain different traces for each action. The reason for multiple runs are that the network traffic can be influenced by other factors such as packet drops, delays, time-of-day or device update affecting traffic. We can represent the trace list obtained from profiling as $T = (t_{(1)}, t_{(2)}, ..., t_{(|T|)})$, and the corresponding labels as the secret information as a vector $\vec{y} = (y_{(1)}, y_{(2)}, ..., y_{(|T|)})$.

Assumptions about attacker capabilities. In our attack model, we assume that an eavesdropper is able to capture traces of user actions using duplicate IoT devices (which is possible for commercial devices) and train classifiers using the captured traces to identify devices or user actions. To perform the attack, we assume that the attacker is either in the same local area network or on a network switch between the device and server to monitor the victim's network traffic. We also assume that the attacker has a way to identify different device or client application traffic by either observing the MAC addresses of IoT devices if they are in the local area network or

using separate classifiers for device identification, which is feasible as shown in literature [18] and our experiments. The attacker passively observes each trace associated with the action once and uses the classifiers to obtain the secret information (device type/action depending on the task). We assume that observation of the trace associated with the action happens only once because the attacker cannot cause the user to perform an action. This prevents the amplification strategy to circumvent packet padding as the attacker cannot cause the user to perform the same action multiple times knowingly. We also assume that the attacker has no prior information that can impact its information gain such as sleep patterns or job schedules of the users.

3 FEATURE EXTRACTION AND PRIORITIZATION

In this section, we describe the initial steps of our targeted mitigation approach (see Figure 1), the feature extraction and feature prioritization techniques.

3.1 Feature Extraction from Network Traces

To have an accurate information leakage analysis, we have to process the traces obtained from capturing network packets and extract meaningful features from the network packets. Based on our definition of traces and the corresponding secrets in Section 2, we define feature functions $f : \mathbf{T} \rightarrow \mathbb{R}$ (where $\mathbf{T}$ is the domain of traces) which map each trace t to a numerical feature value that can potentially correlate with the corresponding action label y. We extract packet-based features such as size of each packet and inter-arrival time to the consecutive packet. We also extract trace-level features such as the total size and total duration of trace, mean, standard deviation, min and max of packet size and time differences. These feature definitions are based on the leakage sources in the prior work [14, 20]. Our feature function definitions are described in Table 1. For each function f_i, we apply it to the trace set T to obtain the vector of values for that feature, $F_i = \langle f_i(t_{(1)}), f_i(t_{(2)}), \ldots, f_i(t_{(|T|)})\rangle$ which we use to determine the probability distributions.

Table 1: Definition of network trace features.

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

3.2 Feature Prioritization via Leakage Quantification

To target mitigation to specific features, we need to have a metric of importance where mitigating more *important* features would reduce the information leakage more than mitigating less important features. To measure importance of each feature, we use an information theoretic measure, Shannon entropy [23] which quantifies information in terms of amount of bits. If we have n unique secret values, $\log_2 n$ will be the maximum amount of information leakage for that secret according to Shannon entropy. We define a feature vector $F_i = \langle f_i(t_{(1)}), f_i(t_{(2)}), \ldots, f_i(t_{(|T|)})\rangle$ to represent the value of feature i over each trace $t_{(j)}$ in T. Recall that, we use a secret vector $\vec{y} = (y_{(1)}, y_{(2)}, ..., y_{(|T|)})$ which represents the secret value $y_{(j)}$ of the corresponding trace $t_{(j)}$. We use mutual information $I(\mathbf{Y};\mathbf{F}_i)$ to quantify the information leakage for feature i, where $\mathbf{Y}$ is the domain of all secret values and $\mathbf{F}_i$ is the domain of feature values for feature i. The mutual information for feature i, I_i, is defined as

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

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

We do not have the exact probability distributions $p_i(x)$, $p_i(y|x)$. Therefore, we first compute the estimated probability distribution $\hat{p}_i(x|y)$ using the feature vector F_i and the corresponding secret vector $\vec{y}$. We use Kernel Density Estimation (KDE) with k-fold cross validation [12] to estimate the probability distribution $\hat{p}_i(x|y)$ [12, 14]. We assume the secret distribution $p(y)$ to be uniform to represent the attacker has no prior information about the secret. It can be modified in the cases of prior information to reduce the initial amount of information. We compute $\hat{p}_i(x)$ and $\hat{p}_i(y|x)$ using $p(y)$ and $\hat{p}_i(x|y)$ and the Bayes' rule.

After quantifying the information leakage of each feature separately, IoTPatch ranks them from the highest amount of information leaked to the lowest, which we then use in our targeted mitigation strategy. Algorithm 1 describes the feature ranking method where the highest ranked feature f_m is found by the formula: $m = \arg\max_i I_i$.

Algorithm 1 FeaturePrioritization(T, L_f) Given a set of traces T, and a list of feature functions L_f, FeaturePrioritization extracts features over traces, ranks them based on the information leakage per feature and returns a list of feature rankings.

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

4 PARETO OPTIMAL MITIGATION FOR MULTIPLE FEATURES

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

4.1 Cost Models

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

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

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

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

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

These cost metrics represent the average increase in number of bytes transmitted and duration of the traces respectively. Increase in duration of user interaction for the same action or increase in number of bytes transmitted would impact power usage, quality of service and total used bandwidth.

4.2 Tunable Mitigation on Targeted Features

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

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

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

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

We describe our mitigation approach in Algorithm 2. Our approach iterates over the ranked feature list L_f^{ranked}, modifies the traces based on the feature (which we explain in the following subsection), quantifies the information leakage of the modified trace and calculates the objective function value Θ. $\textsc{QuantifyLeakage}(T')$ performs feature extraction and quantification over the modified set of traces T' and returns the highest information leakage measure over all features as described in Sections 3.1 and 3.2 (Lines 3–6). If the objective function is minimizing compared to the previous iteration (Lines 7–10), our approach updates the current minimum value $\Theta_{\min}$ to Θ, updates the trace set T to the modified trace set T' and saves the feature for the revised feature ranking L_f^{revised}. If the modification based on the feature does not improve the objective function (it could be because the modification increases the overhead too much), we exclude the feature from the revised feature ranking. After the execution, our approach returns the revised feature ranking L_f^{revised} which can be used to modify unseen traces.

Algorithm 2 $\textsc{TargetedMitigation}(T, L_f^{\text{ranked}}, \alpha, \beta, \gamma)$ Given a set of traces T, list of ranked features L_f^{ranked}, leakage weight α, space cost weight β and time cost weight γ, TargetedMitigation iterates over the features and shapes the traffic to reduce the information leakage while minimizing the objective function, returning the list of features L_f^{revised} that improve the objective function (where $\Theta_{\min}$ denotes the current minimum value of the objective function).

1: $\Theta_{\min} \leftarrow \infty$
2: $L_f^{\text{revised}} \leftarrow \langle\rangle$
3: **for** $j \leftarrow 1$ to $|L_f^{\text{ranked}}|$ **do**
4: **for** each $t_{(k)}$ in T **do**
5: $t'_{(k)} \leftarrow \textsc{Modify}(t_{(k)}, L_f^{\text{ranked}}[j])$ ▷ Modify trace to reduce leakage from feature $L_f^{\text{ranked}}[j]$ from Section 4.3
6: $\Theta \leftarrow \alpha \times \textsc{QuantifyLeakage}(T') + \beta \times C_{space}(T, T') + \gamma \times C_{time}(T, T')$
7: **if** $\Theta < \Theta_{\min}$ **then**
8: $\Theta_{\min} \leftarrow \Theta$
9: $T \leftarrow T'$
10: L_f^{revised}.append($L_f^{\text{ranked}}[j]$)
11: **return** L_f^{revised} ▷ List of features that improve the objective function

4.3 Targeted Trace Modification

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

Second method, $\textsc{Delay}(t, d_{limit})$, delays each packet based on the uniform delaying up to a certain limit. This helps with mitigating

Algorithm 3 OnlineMitigation(p, L_f^{revised}) Given a packet p and the revised feature set L_f^{revised}, OnlineMitigation modifies the packet based on the targeted features. We do not include dummy packet injections in this description, those packets are sent without processing any packets.

1: $i \leftarrow i+1$ ▷ Packet index counter
2: **for** $j \leftarrow 1$ to $|L_f^{\text{revised}}|$ **do**
3: $p \leftarrow \text{ModifyPacket}(p, i, L_f^{\text{revised}}[j])$ ▷ Modify packet to apply mitigation strategy based on feature $L_f^{\text{revised}}[j]$, described in Section 4.3
4: **return** p ▷ Modified packet based on padding and delays

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

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

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

- For feature $f^{\text{size}}(i)$, we equalize the size of i^{th} packet in each trace to avoid information leakage. For each trace t, we modify the size of i^{th} packet p_i to the maximum size of i^{th} packet over all traces. It can be described as $\forall t' \in T', p'_i.\text{size} \leftarrow \max_{t \in T} f^{size}(t, i)$.
- For feature $f^{\Delta\text{time}}(i)$, we delay the $(i+1)^{\text{th}}$ packet to equalize the delta between them. For each trace t, we modify the response time for the $(i+1)^{\text{th}}$ packet to maximum delay between i^{th} and $(i+1)^{\text{th}}$ packets. It can be described as $p'_{i+1}.\text{time} \leftarrow \max_{t \in T} f^{\Delta\text{time}}(t, i) + p_i.\text{time}$.
- For feature $f^{\text{max-size}}$, for each trace t, we inject a packet to t with size equal to maximum size over all packets to obfuscate this feature. For each t, we modify it to create t' where $t' \leftarrow \text{Inject}(t, 1, \max_{t \in T} f^{\text{max-size}}(t))$.
- For feature $f^{\text{min-size}}$, we pad all packets with size below a threshold to equalize the sizes of minimum packets. It can be described as $\forall p \in t, p.\text{size} < \max_{t \in T} f^{\text{min-size}}(t) : p'.\text{size} \leftarrow \max_{t \in T} f^{\text{min-size}}(t)$.
- For feature $f^{\text{num-pkt}}(k)$, we inject packets to equalize the number of packets with size k. For each t, we pick a random number of packets $n \sim \mathcal{U}(0, 2 \times \max_{t \in T} f^{\text{num-pkt}}(t, k))$ and modify the original trace to create t' where $t' \leftarrow \text{Inject}(t, n, k)$.
- For the size and timing based aggregate features, ($f^{\text{sum-size}}$, $f^{\text{var-size}}$, $f^{\text{avg-size}}$, f^{duration}) we search for a padding strategy to apply to all packets by searching for best parameters D or d_{limit} over a set of parameters for Pad(t, D) and Delay(t, d_{limit}) respectively. We describe the set of parameters used for the search method in Section 5.1.

4.4 Online Mitigation

As shown in the previous subsections, while performing the search in Algorithm 2, trace modification (Modify(t, f)) and leakage quantification (QuantifyLeakage(T')) is done offline and applied to each trace. In real traffic, this is not possible as the traces must be processed and modified as each packet arrives. To address this, using the modified feature set, we implement a packet-based mitigation system that modifies each packet based on the results of the search in Algorithm 2. Using the results of the search, we can synthesize the online mitigation that takes a packet and modifies it based on the features we should modify to reduce the information leakage.

Algorithm 3 describes a method that processes each packet based on the mitigations and returns the modified packet. It takes a packet p, modifies it based on the packet index i and each leaking feature in the revised feature list. Instead of the Modify(t, f) described in Section 4.3, we use a method called ModifyPacket(p, i, f) which extends Modify(t, f) on a packet p_i rather than the full trace t. For Pad and Delay functions, there's no change as both methods apply same modifications to each packet of the trace indiscriminately. For Inject(t, k, s), as we limit it to injecting k packets of size s within a trace t, we obtain the average number of packets per trace from the training set T as $\mu_{\text{num-pkt}}$ and inject a packet for each $\mu_{\text{num-pkt}}/k$ packets.

A router that processes packets can use this method to apply padding and delays to each received packet that is transmitted between IoT device, server and smartphone application. In a real world implementation, this algorithm would need to be optimized to apply the mitigation quickly and send the packet with minimal delays and it can be done as shown in prior work [20] and we will demonstrate it in our experimental evaluation.

5 IMPLEMENTATION AND EXPERIMENTS

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

5.1 Implementation

IoTPatch is implemented in Python. For trace capture, automation of analysis, mutual information calculation, and feature ranking capabilities, we use tools described in [14, 21]. For online mitigation, we used Scapy's network capture, padding and packet sending capabilities [9]. For measuring the effectiveness of the mitigation, we used existing implementations of random forest classifier, k-nearest neighbor, and fully connected neural network algorithms in *scikit-learn* library [17]. For the random forest classifier, we set the number of decision trees to 100 which we obtained after testing the random forest with 50, 100, 150 and 200 and picking the value maximizing accuracy after cross validation.

For the mitigation parameters D and d_{limit} of $\textsc{Pad}(t, D)$ and $\textsc{Delay}(t, d_{limit})$, we use grid search over a set of fixed parameters, picking the parameter that minimizes the objective function. For D, we use uniform distribution $\mathcal{U}(0, k)$ where $k \in \{25, 50, 100, 150, 200, 250\}$ bytes and existing packet padding methods in the related work we compare against in the next section. We also calculate the per packet size difference between the largest and smallest traces and use that value as k for the search. For the delay parameter d_{limit} search, we use the set of parameters {10ms, 20ms, 50ms, 100ms, 200ms, 300ms} to test the impact of low and high delay approaches.

For the experimental evaluation of IoTPatch against related work, we set γ to ∞ and only use α and β parameters to focus on space cost and accuracy in the objective function to have a fair comparison between IoTPatch and most of the prior work which focuses on only packet padding. For quantification, we use the default parameters in [14] implementation which works for estimating distributions of space and time features. We ran each experiment 3 times and averaged the results over 3 runs.

5.2 IoT Benchmarks

5.2.1 IoT protocol benchmarks. To cover the common design and architecture patterns, we used the gRPC, MQTT, and STOMP protocols discussed in Section 2 to create applications with various protocols. We created 10 applications using the protocols representing both smart home and industrial IoT systems such as ovens, air conditioners, smart locks and electrical switches. Details of the applications can be examined in this repository [6].

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

5.2.2 IoT real-world benchmarks. We used an existing IoT application benchmark from previous work, the UNSW device identification dataset [24] which contains network traces collected from a variety of IoT devices in which the task is to identify devices.

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

5.3 Experimental Evaluation

We compare our approach against prior mitigation methods which use a fixed or randomized packet padding strategy for each packet [11, 20, 26, 27] such as linear padding (increasing size to the nearest increment of 128) [11], exponential padding (increasing size to the nearest power of 2) [11], uniform padding (padding with 1-1500 bytes randomly) [11, 27], uniform-255 padding (padding with 1-255 bytes randomly) [11], maximum transmission unit (MTU) padding (increasing size to maximum transmission unit of 1500 bytes for TCP/UDP packets) [11], MTU padding with 0-20 ms delay to each packet (MTU-20ms) [26], mice & elephants padding (increasing size of each packet with size less than 100 to 100, to 1500 bytes otherwise) [11], Level-X where X is 100, 500, 700 or 900 (increasing size of each packet with size less than X to X, pad them randomly otherwise) [20]. None of the aforementioned approaches had a public implementation, therefore we implemented the methods ourselves.

To evaluate IoTPatch against the various related work and compare its performance against advanced attacks, we trained random forest classifiers [13], k-nearest neighbors [5] and fully-connected neural networks [22] on the traces. Random forest classifiers performed the best in terms of accuracy, therefore we used random forest classifiers in our evaluations. Hence, we are evaluating the ability of different mitigation strategies against the best performing classifier's ability to infer the secret from network traces.

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

5.3.1 Comparison of IoTPatch *to prior work.* To compare IoTPatch to prior work, we trained random forest classifiers over the mitigated traffic using various mitigation approaches over all the aforementioned benchmarks. As a baseline, we also include the performance of the classifier when no mitigation and full mitigation (padding all packets to full size, delaying packets to make the transmission like heartbeat and injecting packets to make trace size equal overall) and random guessing probability.

Table 2 shows the average testing accuracy, precision, recall, F_1-score [25] and space cost using the random forest classifier. The results show that when our mitigation approach, IoTPatch, is used, mitigated traces leak less information and induce less overhead compared to the prior work. As seen in the table, IoTPatch with different objective functions represent a trade-off between overhead and accuracy and the user can select the solution fitting their constraints. IoTPatch (Overhead) achieves lowest overhead compared to other works while reducing leakage some amount. IoTPatch (Balanced) has similar overhead compared to linear padding while reducing leakage metrics close to MTU with 20ms delay. IoTPatch (Leakage) has very high overhead while reducing leakage only 1-2% compared to IoTPatch (Balanced). Our method is able to mitigate sources of side-channels by padding a single packet or injecting a few packets which help our methods achieve low overhead when that is prioritized. The search based mitigation minimizes the objective function at each step, enabling IoTPatch to achieve lower overhead, leakage or both. On all of the mitigation approaches, online mitigation adds in average 4% time overhead with packet processing. This processing delay is similar to mitigation time delays in previous work such as [19] which shows the main reason of transmission delay is network congestion rather than packet processing.

Table 2: Average testing accuracy, precision, recall, F_1-score and packet size overhead results of the prior work and IoTPatch with 6 objective functions on all benchmarks, trained on a random forest classifier. Overhead, Balanced and Leakage results of IoTPatch represent cases where the objective function weights α and β are 1 and 1, 1 and 0.1, 1 and 0.01 respectively. Bold values are minimum values among the methods within their category (related work, IoTPatch) separated with bold lines.

Mitigation Method	Accuracy	Precision	Recall	F_1-Score	C_{space}
No mitigation	0.82	0.83	0.82	0.82	**0.00**
Full mitigation	**0.44**	**0.44**	**0.44**	**0.44**	113.39
Random Guess	0.30	0.30	0.30	0.30	N/A
Uniform	0.53	0.52	0.51	0.51	7.51
Uniform255	0.54	0.54	0.53	0.53	1.36
Mice & Elephants	0.55	0.54	0.53	0.53	2.62
Linear	0.55	0.55	0.54	0.54	0.94
Exp	0.55	0.56	0.55	0.54	**0.32**
MTU	0.53	0.53	0.52	0.52	15.01
MTU-20ms	**0.50**	**0.48**	**0.48**	**0.48**	15.01
Level-100	0.56	0.56	0.55	0.55	0.76
Level-500	0.54	0.53	0.52	0.52	4.57
Level-700	0.53	0.53	0.52	0.52	6.62
Level-900	0.54	0.52	0.52	0.52	8.70
IoTPatch (Overhead)	0.58	0.58	0.58	0.58	**0.16**
IoTPatch (Balanced)	0.50	0.49	0.48	0.48	0.80
IoTPatch (Leakage)	**0.49**	**0.48**	**0.47**	**0.46**	8.87
IoTPatch (Overhead w/Time)	0.57	0.57	0.56	0.56	**0.14**
IoTPatch (Balanced w/Time)	0.50	0.50	0.49	0.49	0.44
IoTPatch (Leakage w/Time)	**0.46**	**0.46**	**0.45**	**0.44**	1.32

5.3.2 Pareto optimality of IoTPatch. Figure 2 shows the accuracy (x-axis) and space overhead (y-axis) results for the prior work and IoTPatch with different objective function parameters which weigh overhead and leakage at different levels with simulated and actual traffic results. Both plots show that IoTPatch with various objective functions provides a Pareto optimal solution set, where different objective functions result in different points in the accuracy-overhead space and the simulated results and actual traffic show similar results. The users can run IoTPatch with different objective functions, get the mitigation strategies with various results and pick the one that fits their needs and requirements. For example, in the MyQ garage door example in Figure 2, they can pick the strategy with low overhead where the attacker can guess accurately with 65% accuracy or pick the strategy with higher overhead while reducing the accuracy of the attacker to 50%, making it a random guess.

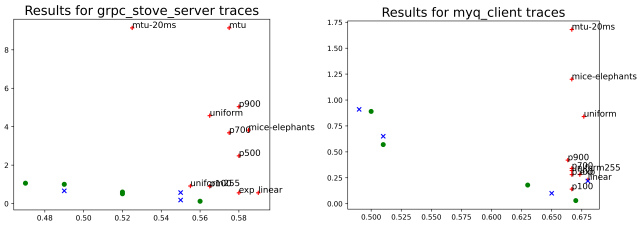

Figure 2: Average accuracy (x-axis) and total overhead (y-axis) results of prior work and IoTPatch using the random forest classifier. Red pluses represent the results of prior approaches, blue crosses and green dots represent the results of IoTPatch with different weights for objective function when online mitigation and simulated mitigation is applied respectively. Lower values are the better results.

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

Table 2 shows the results of IoTPatch with/without the timing mitigation. Compared to IoTPatch without any timing mitigation, timing mitigation either reduces the space overhead while obtaining similar results, reduces information leakage or both. IoTPatch (Leakage w/Time) improves both metrics compared to IoTPatch (Leakage) where modifying both time and space features helps the search process have more options to improve the objective function with only 5% average time overhead. Compared to MTU with 0-20 ms delays which have a 90% average time overhead, our version achieves better results with lower time and space overheads with 5-10% time overhead. Results of both approaches demonstrate the importance of targeting timing side-channels.

5.3.4 Limitations. Our results on feature prioritization and side-channel mitigation depend on the quality of the captured network trace set that contains a variety of user behaviors. If the number of traces are low or they are captured in a way such that some other unrelated event (such as time of day, device updates, etc.) correlates with the action, IoTPatch can try to mitigate the traffic, assuming it leaks information when in fact it does not leak information in the real world. To alleviate validity concerns, in our evaluations we

split the trace sets such that we synthesize the mitigation strategy on seen traces and demonstrate effectiveness of mitigation against attacks on another set of unseen traces.

IoTPatch detects and quantifies the information leakage and it can find the optimum mitigation strategy to reduce leakage, however implementing that strategy is left to the user. We propose a method for the code synthesis for the mitigation strategy in experimental evaluation and it can be implemented similar to the prior works [20, 26] which use software defined networking to manipulate network traffic data

6 RELATED WORK

In the network side-channel mitigation area, there are several proposed mitigation techniques based on packet padding and delaying [11, 20, 26, 27]. We compared our work against the proposed techniques and experimentally demonstrated that our work synthesizes mitigation strategies with better accuracy and overhead due to feature prioritization and refinement on an objective function.

Apthorpe et al.'s work Stochastic Traffic Padding [8] mitigates side-channels caused by a burst of packets (such as a camera uploading a photo, Amazon Echo downloading music files for play, etc.) where the timing of the burst of packets leaks when the event happens. They obfuscate the timing of the event by sending fake bursts of packets over a trace and tune the amount of bursts based on guessing accuracy. Their method is tailored on mitigating a specific type of information leakage based on user taking or not taking an action whereas we try to mitigate information leakages in general with lower overhead. The padding method is not publicly available to use and the authors did not respond to our requests for their implementation.

Liu et al.'s work, SniffMislead [15] aims to obfuscate the correlation between network side channels and user actions by simulating dummy users over the network. They capture traces and use classifiers to identify which packets are relevant to the action and replay those packets over the trace as phantom users to confuse any eavesdroppers. Our approach is more general as it can be used to reduce information leakage for device fingerprinting or action fingerprinting as we evaluate our approach over a variety of benchmarks. We also provide a tunable mitigation strategy for the privacy and overhead constraints of the user.

7 CONCLUSIONS

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

REFERENCES

[1] 2019-03-07. MQTT Version 5.0 OASIS Standard. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[2] 2020-03-18. gRPC Concepts. https://grpc.io/docs/guides/concepts/

[3] 2020-05-06. STOMP Protocol Specification, Version 1.2. https://stomp.github.io/stomp-specification-1.2.html

[4] Fadele Ayotunde Alaba, Mazliza Othman, Ibrahim Abaker Targio Hashem, and Faiz Alotaibi. 2017. Internet of Things security: A survey. *Journal of Network and Computer Applications* 88 (2017), 10–28.

[5] Naomi S Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician* 46, 3 (1992), 175–185.

[6] Anonymous. 2020. IoT benchmark applications. https://anonymous.4open.science/r/30c5353a-0802-446b-82f8-5debf7fc08ec/.

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

[8] Noah J. Apthorpe, Danny Yuxing Huang, Dillon Reisman, Arvind Narayanan, and Nick Feamster. 2019. Keeping the Smart Home Private with Smart(er) IoT Traffic Shaping. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 128–148. https://doi.org/10.2478/popets-2019-0040

[9] Philippe Biondi. 2021-04-19. Scapy: Packet crafting for Python. https://scapy.net/

[10] Yair Censor. 1977. Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization* 4, 1 (1977), 41–59.

[11] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. 2012. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE symposium on security and privacy*. IEEE, 332–346.

[12] Peter Hall, JS Marron, and Byeong U Park. 1992. Smoothed cross-validation. *Probability theory and related fields* 92, 1 (1992), 1–20.

[13] Tin Kam Ho. 1998. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence* 20, 8 (1998), 832–844.

[14] Ismet 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*.

[15] Xuanyu Liu, Qiang Zeng, Xiaojiang Du, Siva Likitha Valluru, Chenglong Fu, Xiao Fu, and Bin Luo. 2021. SniffMislead: Non-Intrusive Privacy Protection against Wireless Packet Sniffers in Smart Homes. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*. 33–47.

[16] Nitin Naik. 2017. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE International Systems Engineering Symposium (ISSE)*. IEEE, Vienna, Austria, 1–7. https://doi.org/10.1109/SysEng.2017.8088251

[17] 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.

[18] Antônio J Pinheiro, Jeandro de M Bezerra, Caio AP Burgardt, and Divanilson R Campelo. 2019. Identifying IoT devices and events based on packet length from encrypted traffic. *Computer Communications* 144 (2019), 8–17.

[19] Antônio J Pinheiro, Jeandro M Bezerra, and Divanilson R Campelo. 2018. Packet padding for improving privacy in consumer IoT. In *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 00925–00929.

[20] Antônio J Pinheiro, Paulo Freitas de Araujo-Filho, Jeandro de M Bezerra, and Divanilson R Campelo. 2020. Adaptive Packet Padding Approach for Smart Home Networks: A Tradeoff Between Privacy and Performance. *IEEE Internet of Things Journal* 8, 5 (2020), 3930–3938.

[21] 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*.

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

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

[24] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. 2018. Classifying IoT devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing* 18, 8 (2018), 1745–1759.

[25] Alaa Tharwat. 2020. Classification assessment methods. *Applied Computing and Informatics* (2020).

[26] Mostafa Uddin, Tamer Nadeem, and Santosh Nukavarapu. 2019. Extreme SDN framework for IoT and mobile applications flexible privacy at the edge. In *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom*. IEEE, 1–11.

[27] Sijie Xiong, Anand D Sarwate, and Narayan B Mandayam. 2018. Defending against packet-size side-channel attacks in IoT networks. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2027–2031.