Automated Detection of Side Channels in Cryptographic Protocols: DROWN the ROBOTs!

Jan Peter Drees
University of Wuppertal
jan.drees@uni-wuppertal.de

Pritha Gupta
Paderborn University
prithag@mail.upb.de

Eyke Hüllermeier
LMU Munich
eyke@ifi.lmu.de

Tibor Jager
University of Wuppertal
tibor.jager@uni-wuppertal.de

Alexander Konze
achelos GmbH
alexander.konze@achelos.de

Claudia Priesterjahn
achelos GmbH
claudia.priesterjahn@achelos.de

Arunselvan Ramaswamy
Paderborn University
arunr@mail.upb.de

Juraj Somorovsky
Paderborn University
juraj.somorovsky@upb.de

Abstract—**Currently most practical attacks on cryptographic protocols like TLS are based on side channels, such as padding oracles. Some well-known recent examples are DROWN, ROBOT and Raccoon (USENIX Security 2016, 2018, 2021). Such attacks are usually found by careful and time-consuming manual analysis by specialists.**

In this paper, we consider the question of how such attacks can be systematically detected and prevented before (large-scale) deployment. We propose a new, fully automated approach, which uses supervised learning to identify arbitrary *patterns* in network protocol traffic. In contrast to classical scanners, which search for *known* side channels, the detection of general patterns might detect new side channels, even "unexpected" ones, such as those from the ROBOT attack.

To analyze this approach, we develop a tool to detect Bleichenbacher-like padding oracles in TLS server implementations, based on an ensemble of machine learning algorithms. We verify that the approach indeed detects known vulnerabilities successfully and reliably. The tool also provides detailed information about detected patterns to developers, to assist in removing a potential padding oracle. Due to the automation, the approach scales much better than manual analysis and could even be integrated with a CI/CD pipeline of a development environment, for example.

I. Introduction

Many recent attacks on cryptographic protocols deployed in practice do not break the cryptographic algorithms directly. Instead, they are based on *side-channel* information. For instance, this includes many recent attacks on TLS, probably the most widely-used and well-analyzed cryptographic protocol on the Internet, such as DROWN [1], ROBOT [6], Raccoon [42], POODLE [38], and many more.

One particularly important family are *padding oracle attacks*, such as the attacks by Bleichenbacher [5], Manger [34], and Vaudenay [53]. Padding oracle attacks have found countless applications. Since the original publication of Bleichenbacher's attack in 1998, follow-up works have developed many different approaches which construct the padding oracle required for this attack, which then often yields an efficient attack on the considered implementation or application [1, 2, 11, 13, 26, 29, 36, 48, 59].

Particularly surprising constructions of padding oracles were shown in the ROBOT attack by Böck *et al.* [6]. This work demonstrated that padding oracles can appear in very subtle and quite unexpected forms. For example, one would probably expect that the TCP protocol, which is used to transport TLS messages, cannot provide an exploitable padding oracle, because it is merely a transport protocol that operates on a completely different network layer. The protocol is independent of the cryptographic keys used in higher-layer protocols (such as TLS running over TCP), and therefore should not be able to leak any information. However, Böck *et al.* showed, very surprisingly, that certain TLS implementations may terminate a TCP session in different ways, depending on whether a padding error occurred or not. This could be used to construct a new padding oracle. The authors of the ROBOT paper also found new vulnerabilities in commercial products by Cisco, Citrix, F5, Symantec, and Cavium, and even demonstrated a forgery of a valid digital signature using Facebook's RSA certificate that was based on a padding oracle provided by Facebook's custom TLS server implementation.

All padding oracle attacks appearing in the literature so far seem to have been found "manually", that is, via careful analyses of TLS server responses by specialized expert security researchers, who thoroughly analyzed one popular implementation after another. Of course, this approach does not scale well with a growing number of implementations. Furthermore, even for experts it is very difficult to find "unexpected" side channels that go beyond what one is specifically looking for, such as the aforementioned TCP side channel [6], for instance.

Research challenge: The development of techniques that are capable of finding such cryptographic side channels automatically, without the need for time-consuming manual analysis, is a foundational open problem. The main difficulty is that in order to be able to identify even unexpected side channels, it is not sufficient to check against a list of known or typical vulnerabilities. Instead, one has to analyze the behavior of an implementation and efficiently recognize *general patterns* that depend on the validity of the padding, and thus might give rise to a padding oracle.

For example, in the context of TLS implementations, we would like to have a tool that can be executed against any concrete TLS implementation, possibly after every significant code modification or before every release of a new software version. Such a tool should not require any extensive and time-consuming manual analysis or supervision by an expert security researcher. Instead, it should be usable without expert knowledge, ideally in a fully-automated way that allows to run automated tests, possibly in the regression testing phase of a CI/CD pipeline. This would significantly reduce the attack surface of practical applications.

Our contributions: We propose an automated approach to detect side channels such as padding oracles in cryptographic protocol implementations, which uses a variety of classification algorithms from machine learning to automatically detect general *patterns* in network protocol traffic that might give rise to a padding oracle. Our solution does not merely use the predictions of a pre-trained model but relies on the ability to learn patterns of interest.

In order to analyze this general approach, we consider Bleichenbacher-like attacks on TLS as a concrete use case. This class of attacks provides a prime example of a cryptographic side channel attack on the protocol level. A long sequence of research papers appearing on leading academic security conferences [1, 2, 5, 6, 11, 13, 26, 29, 36, 48, 59] showed that such vulnerabilities appear repeatedly in popular open-source software and widely-used commercial products. Hence, this is an ideal reference for the development and analysis of an automated methodology to detect side channel attacks on the protocol level.

In order to minimize the probability that a padding oracle vulnerability remains undetected in the automated analysis, we propose to train an ensemble of machine learning algorithms and aggregate them. Concretely, we consider 10 algorithms from different families, including for instance Logistic Regression, Support Vector Machines, Decision Trees, Random Forest, and Boosting algorithms. Since detection of vulnerabilities is most useful when one can also provide information of the origin of recognized patterns (e. g., which particular protocol message exhibits the pattern), we focus on machine learning (ML) algorithms that are amenable to *feature importance* techniques.

We implement and analyze this approach in a tool which automatically analyses a given TLS server implementation. The tool implements a TLS client to generate training and testing data and then applies the machine learning algorithms to detect potential side channels.

We confirm that the tool is indeed able to detect known vulnerabilities in TLS server implementations reliably. Concretely, the tool correctly identifies the vulnerability identified in [29] in OpenSSL version 0.9.7a, while no vulnerability is identified in version 0.9.7b (which patches this vulnerability). We also confirm that the tool reliably detects all padding oracle vulnerabilities described in the ROBOT paper [6]. Since some of these vulnerabilities were found in proprietary implementations (e. g., Facebook's and Cisco's) which are not publicly available for analysis, we patched an mbedTLS[1] server according to the description of the behavior of these implementations from the ROBOT paper [6] to simulate these padding oracles. Finally, we analyze the most recent versions of 13 different popular open source TLS implementations (cf. Table II), but (as expected) without finding any new vulnerabilities. We conclude that the tool is able to reliably detect known vulnerabilities, with a general and generic approach. We consider this as an indicator that the approach will also work for future, new side channels that exhibit distinguishable patterns on the network layer.

To assist in removing identified potential vulnerabilities, the tool also provides detailed feedback about detected patterns to developers, for which we rely on *feature importance* techniques of the considered ML algorithms.

We hope that the ideas developed here may potentially detect new subtle and complex side channels in the future *before* large-scale deployment of an implementation.

Supplementary material: We make the tool and all data publicly available as open source. The full source code of the tool is available on Github.[2]

II. Related Work

a) Attacks based on Bleichenbacher's: The original padding oracle considered in [5] was based on distinguishable error messages returned by a TLS server implementation.[3] Subsequent protocol versions then required indistinguishable error messages, in order to remove this particular way to construct a padding oracle.

The difficulty of detecting and preventing *all* possible side channels that may give rise to a padding oracle has been demonstrated by many research papers that appeared since the original publication of Bleichenbacher's attack in 1998. Klíma *et al.* [29] introduced a new variant (a "bad version oracle", a special case of a padding oracle) and also used timing as a new way to construct the padding oracle required for Bleichenbacher's attack. Jager *et al.* [26] showed that XML Encryption inherently provides a padding oracle, which is based on application layer properties of Web services and XML. Degabriele *et al.* [11] described attacks on the Europay-Mastercard-Visa (EMV) specification. Bardou *et al.* [2] developed a clever variant of Bleichenbacher's algorithm that may improve the performance of attacks significantly. They also found new padding oracles in several applications, including RSA SecurID tokens and several other hardware tokens, Siemens CardOS smartcards, hardware security modules, and even the cryptography implementation of the Estonian ID card. Meyer *et al.* [36] found several new padding oracles in the Java Secure Socket Extension (JSSE) TLS implementation and in hardware security appliances using the Cavium NITROX SSL accelerator chip. The DROWN attack [1] discovered

[1]https://tls.mbed.org/

[2]Following the submission guidelines, we do not yet provide a Github link here. You can access the source code and datasets for review instead at https://www.dropbox.com/sh/8gwwg0v1vlmlbbt/AAAsNx5Ptb2Kx72WCkdzr9Xea

[3]Early versions of TLS were actually called SSL, and re-named to TLS with the specification of TLS 1.0 by the IETF in 1999. We use the term TLS for all protocols versions.

another new vulnerability in OpenSSL that was present in OpenSSL releases from 1998 to early 2015, which gave rise to extremely efficient Bleichenbacher-style attacks, by leveraging an additional vulnerability in OpenSSL even in less than one minute on a single CPU. In 2018, Felsch *et al.* found new padding oracles in widely used IPSec implementations by Cisco, Huawei, Clavister, and ZyXEL [13]. Zhang *et al.* [59] and Ronen *et al.* [48] considered settings where the attacker is able to run code on the same physical machine as the victim, which circumvents many countermeasures to the aforementioned attacks. Even though this is a strong attacker model, it seems very reasonable in certain applications, such as cloud computing.

b) Automated scanning for vulnerabilities: Recent analyses on new side-channel vulnerabilities come with large-scale evaluations of frequently used servers to estimate the attack impact. Such analyses were performed for ROBOT [6], RACCOON [42], or CBC padding oracle attacks [35]. All these analyses have in common that the used scanners send test vectors to the servers and evaluated the potential side channels based on differences in server responses. Nevertheless, such an approach comes with potential false positives and negatives, resulting from unstable Internet connections and server behaviors; one broken TCP connection or connection timeout can change the server response resulting in a different behavior and thus in a potential side channel report. Merget *et al.* attempted to solve these problems by rescanning vulnerable servers and by careful statistical tests [35]. The results of these approaches were integrated into common TLS scanning tools, such as SSLlabs[4] or testssl.sh.[5] Therefore, the tools are now able to cover a very wide range of *specific* and *known* vulnerabilities.

While the statistical tests developed in [35, 42] are well-suited for precisely finding padding oracle vulnerabilities, the side channels they search for have to be manually defined by the researchers. For example, TLS-Attacker, which was used in [35, 42], explicitly searches for side channels resulting from different messages, message counts, and TCP connection state differences. All these side channels have been defined after careful manual vulnerability assessments performed in the previous years [6, 35, 42]. It is not guaranteed that the list of the side channels TLS-Attacker and other scanners search for is final. Unexpected behavior in the TLS implementation or the underlying TCP stack can reveal new side channels beyond message differences and TCP connection states, which are not explicitly analyzed. This gap is addressed in our research; our tool observes the whole TLS communication and provides it to the machine learning algorithms, which are able to detect side channels *without previous assumptions* and explain the potential vulnerability to the developer.

c) Machine learning in side channel analysis: Previously, machine learning algorithms have been applied to detect side channel attacks on the algorithmic and hardware level (cf. [9, 24, 32, 33], for instance, [20] for a recent survey, as well as [57, 58] for more recent works). To best of our knowledge, ours is the first approach to consider side channel attacks on the cryptographic *protocol* level. A different research direction was proposed by Beck *et al.*, who analyzed the automatic exploitation of adaptive chosen ciphertext attacks [3]. In their work, they assumed a vulnerable implementation allowing an attacker to modify ciphertexts. They concentrated on the automatic exploitation development with SAT and SMT solvers based on the malleability characteristics of the encryption scheme. Our work extends this interdisciplinary research direction by analyzing machine learning (ML) algorithms for detecting new side channels.

[4]https://www.ssllabs.com/ssltest/
[5]https://testssl.sh/

III. Preliminaries

This section gives a brief introduction to Bleichenbacher's attack in the context of TLS in Section III-A. We particularly consider TLS 1.2. We further summarize relevant concepts and terminology from machine learning in Section III-B.

A. Bleichenbacher's Attack on TLS

a) The TLS 1.2 Handshake: The TLS 1.2 [45] handshake, shown in Figure 1, is an essential first step in the establishment of a secure TLS connection. Performing the handshake, client and server agree on which cryptographic algorithms and parameters to use, they exchange the secret keys they later use to encrypt the actual data being transmitted, and the server proves its identity to the client:

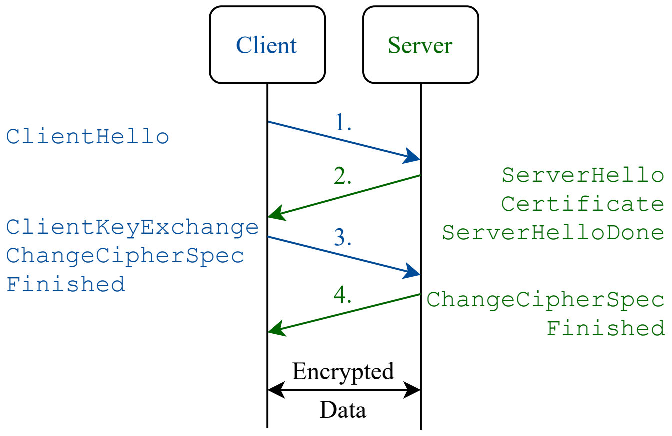

Fig. 1: A typical TLS 1.2 handshake

The TLS Handshake consists of the following sequence of messages.

(1) The client initiates the connection with the `ClientHello` message, and proposes different sets of cryptographic algorithms (so-called "cipher suites") it supports.

(2) The server selects a cipher suite and sends it to the client in the `ServerHello` message. It then proves its identity with a digital signature with respect to a certificate signed by a trusted third party. The `ServerHelloDone` message signals the end of this message flow. In the following, we will

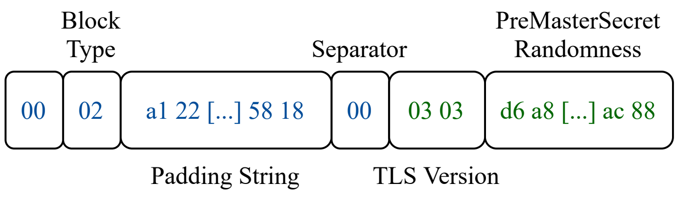

Fig. 2: Padded Premaster Secret PMS.

consider a setting where a cipher suite based on RSA key transport is used.[6]

(3) The client uses the agreed-upon algorithms to perform the actual key exchange with the ClientKeyExchange (CKE). In the case of RSA-based key transport, the client generates a new random key called the *pre-master secret* (PMS) and encrypts it with RSA, such that only the server can decrypt it. It then signals that the remainder of the conversation will be encrypted with this PMS by sending a ChangeCipherSpec (CCS) message. Finally, the Finished (FIN) message contains a cryptographic checksum overall key exchange messages, computed with a key derived from the PMS.

(4) The server decrypts the PMS and uses it to derive several cryptographic keys, which are then used to decrypt and verify the checksum of the FIN message. It concludes the handshake by sending CCS and FIN messages. Thereafter, all communication is protected using the keys derived from the PMS. Note that all cryptographic keys used for a TLS session are derived from PMS and other public values (such as nonces sent in plain text in the ClientHello and ServerHello messages).

b) RSA-PKCS#1v1.5 Encryption: All TLS cipher suites based on RSA key transport use the RSA-PKCS#1 v1.5 encryption scheme [44] in order to transport the PMS from the client to the server. Essentially, this encryption scheme prepends the PMS with constants and random bytes, as defined in Figure 2. The PMS is a random 48-byte string. The leading 00 02, the separator byte 00, and the TLS version number 03 03 (which refers to TLS 1.2) are constants. The padding string is chosen at random from all byte strings that do not contain a 00 byte (to guarantee that the separator byte is uniquely identified), and such that the total length of the padded string (including the PMS and all constants) is equal to the size of the RSA modulus N. The resulting padded string M is then encrypted with the "textbook" RSA encryption function as $c = M^e \bmod N$. We will say that a ciphertext is *PKCS#1 v1.5-conformant* or *has valid padding*, if $M = c^{1/e} \bmod N$ satisfies the padding scheme from Figure 2.

c) Bleichenbacher's Attack: Bleichenbacher's attack assumes that a "padding oracle" is available, which takes as input a ciphertext, and returns whether the ciphertext contains a plaintext with valid PKCS#1 v1.5 padding, with respect to a given target public key (N, e). Such an oracle may be constructed in many different ways, e. g., based on error messages returned by a TLS server implementation. A long sequence of research papers has developed many different ways to concretely construct such an oracle for certain concrete implementations or applications [1, 2, 5, 6, 11, 13, 26, 29, 36, 48, 59].

Bleichenbacher described a seminal algorithm [5] which is able to use such an oracle in order to effciently compute the RSA decryption function $c \mapsto c^{1/e} \bmod N$ with respect to any number $c \bmod N$. Note that this can in particular be used to decrypt RSA PKCS#1 v1.5 ciphertexts, but also to compute valid RSA signatures with respect to the RSA key (N, e) contained in a server's certificate.

Essentially, the idea of Bleichenbacher's algorithm is as follows. Suppose that $c = M^e \bmod N$ be a PKCS#1-conformant ciphertext. This is without loss of generality, because if c is not, then one can use the oracle to "randomize" c by computing $\hat{c} = c\rho^e = (M\rho)^e \bmod N$ for random ρ, until $\hat{c}$ is PKCS#1-conformant, and then continue with $\hat{c}$. Thus, the number $c = M^{1/e} \bmod N$ lies in the interval $[2B, 3B)$, where B denotes the number modulo N whose binary representation is

$$B = \texttt{00 01 ... 00} = 2^{8(\ell-2)}$$

and where ℓ is the byte-length of the RSA modulus N.

Bleichenbacher's algorithm chooses a small integer s, computes

$$c' = (c \cdot s^e) \bmod N = (Ms)^e \bmod N,$$

If the padding oracle reveals that c' has a valid padding, then this implies that $2B \le Ms - rN < 3B$, for some r, which is equivalent to

$$\frac{2B + rN}{s} \le M < \frac{3B + rN}{s}.$$

Thus, M must lie in the interval

$$M \in [\lceil (2B + rN)/s \rceil, \lfloor (3B + rN)/s \rfloor).$$

By repeatedly choosing new s, this yields a set of intervals that narrows down the possible values of M, until only one possibility is left, which has to be the plaintext.

The perfomance of Bleichenbacher's algorithm mainly depends on the provided oracle, and how precisely it checks the validity of the padding. Bardou *et al.* described an improvement to Bleichenbacher's algorithm [2], and also analyzed the concrete efficiency for various types of oracles.

B. Machine Learning

Machine learning (ML) deals with the development and analysis of *learning algorithms* that can automatically improve (their performance), using past experience [37]. ML algorithms build *learning models* using a *training dataset*, in order to make future predictions without being explicitly programmed to do so [30].

[6]For instance, the only cipher suite which is mandatory to implement in TLS 1.2 is of this type (TLS_RSA_WITH_AES_128_CBC_SHA). RSA-based key transport was removed from TLS 1.3, due to the large number of Bleichenbacher-like attacks. However, somewhat counter-intuitively, even TLS 1.3 might be vulnerable to such attacks in the most common practical deployment setting [27].

There are many different approaches in ML, which are broadly divided into three categories based on the type of data provided to the learning algorithms: *supervised learning*, *unsupervised learning* and *reinforcement learning*. Out of these, the simplest and most popular one is *supervised learning*. Here, the learning algorithm is provided with example inputs (training dataset) and their desired outputs (labels). These are used to learn a general mapping from the input space, often represented as d-dimensional real-valued vectors, to the required output, often a real value [37]. The underlying assumption in supervised learning is that the data (input) is generated from an unknown distribution and there exists a mapping from the input space to the desired output space called the *target function*, which in principle is unknown to us. In this paper, we use the *classification* supervised learning approach, in which the goal is to learn a mapping from input to finitely many categories, represented by a natural number called the class-label. For example, the instances can be features of shoes, ($\boldsymbol{x}$ = (price, color, size, healtype)) and the label y could represent categorizes of shoes like ('flat', 'pencil, 'platform') [37]. We call class-labels which are provided by some expert or a user *ground-truth* class-labels.

a) Classification: Let $\mathcal{X} \in \mathbb{R}^d$ be the set of all possible inputs and let $\mathcal{Y}$ be the set of all possible *classes*. For the sake of simplicity, we let $\mathcal{Y} = \{0, 1, \ldots, K-1\} = [K]$, where $2 \leq K < \infty$ represents the number of *classes*, also referred as class-label. The goal of classification is to find a mapping $f : \mathcal{X} \to \mathcal{Y}$, using the available training dataset $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$. It consists of $|\mathcal{D}| = N$ training instances in form of tuples $(\boldsymbol{x}_i, y_i)$, such that $y_i \in \mathcal{Y}$ is the class-label associated with input instance $\boldsymbol{x}_i \in \mathcal{X}$. The learning problem is called *multi-class classification* when the number of classes is $K > 2$ and *binary classification* if $K = 2$ [37]. Formally speaking, the goal in binary classification, is to find a mapping or target function $f : \mathcal{X} \to \mathcal{Y}$, that maximizes the prediction accuracy over the training dataset $\mathcal{D}$. For a given input instance $(\boldsymbol{x}, y) \in \mathcal{D}$, let $y = f(\boldsymbol{x})$ be the corresponding label (ground-truth class-label), and let $\hat{y} = \hat{f}(\boldsymbol{x})$ be the algorithm prediction, such that f is the *target function* and $\hat{f}$ be the *predicted function*.

Accuracy and Error-rate: The average accuracy on the given data is defined as $\mathrm{d}_{\mathrm{bc}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{N}\sum_{i=1}^{N}[\![(\hat{f}(\boldsymbol{x}_i) == f(\boldsymbol{x}_i))]\!]$, where N is the number of training instances; $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are the vectors consisting of ground-truth class-labels and corresponding predicted class-labels, respectively; $[\![(\hat{f}(\boldsymbol{x}_i) == f(\boldsymbol{x}_i))]\!]$ is the indicator function that equals 1 *iff* $\hat{f}(\boldsymbol{x}_i) = f(\boldsymbol{x}_i))$, else it takes value 0. This naturally gives rise to the following definition of training loss called the *error-rate*, $\ell_{\mathrm{bc}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = 1 - \mathrm{d}_{\mathrm{bc}}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ [37]. It may be noted that the predicted function($\hat{f}$) is learned by minimizing the training loss and, $\hat{f}$ is an approximation of the required target function f. For the classification problem, this training loss is the error-rate, ℓ_{bc} calculated on the given training data [37].

Evaluation: Evaluating the performance of $\hat{f}$ is an important problem, since it only minimizes the error-rate on the given dataset. In other words, it minimizes the *in-sample* error (E_{in}). However, this may not be an accurate representation of the general performance of $\hat{f}$, especially on unseen data. Hence, we wish to find a function that not only minimizes E_{in}, but also the error-rate on unseen data, i.e., minimize the *out-of-sample* error (E_{out}). For this, one typically divides the given dataset into training and test data. The predicted function is obtained by minimizing the training loss over the training data, and its performance is evaluated on the test data. To better evaluate the performance of $\hat{f}$, we use the Monte Carlo cross-validation (MCCV) approach, in which the dataset is split into many pairs of train-test datasets, uniformly at random [49, 55]. For a given test set, the remaining dataset constitutes the training set and it is observed that different training and testing sets overlap, i.e. the binary classifier is trained and tested on the same instances in different splits. But one can create a large number of train-test pairs, which in-turn decreases the variance of the mean estimate of the *out-of-sample* performance of $\hat{f}$ [43, 49, 52].

b) Binary Classifiers: A *learning algorithm* used for solving a binary classification task is called a binary classification algorithm and the learning model produced by training on the given dataset is called binary classifier. Each binary classification algorithm has defined a set of functions called the *candidate space*[7] which the algorithm uses to find the *predicted function* that fits the given training dataset most accurately [37]. The candidate space together with the learning algorithm is also referred to as the "Learning Model" [37].

Among the most commonly used binary classifiers proposed in the literature are Perceptron Learning Algorithm (PLA) [16], Logistic Regression (LR) [56], and Ridge Classifier (RC) [21]. These algorithms are limited in that they are only able to solve binary classification problems involving linearly separable data (separable by a hyperplane). Support Vector Machine (SVM) is a popular algorithm that can classify data that is not linearly separable using a *kernel trick*, i.e. its *candidate space* also contains non-linear functions [10, 41]. Decision Tree (DT) is another important approach that learns a set of rules which can be used to classify the input $\boldsymbol{x} \in \mathcal{X}$ [47].

Given the different properties of the described algorithms, it is a common approach to combine more than one classifier to solve a classification problem. This constitutes the *ensemble* based approach, in which multiple classifiers are trained on the given dataset and aggregated to obtain the final predicted function [46]. The first ensemble-based approach is *bagging*, in which each classifier is trained on a sub-sample of the given training dataset. The trained binary classifier is called the *base learner*. There are two popular bagging approaches with DT as a base learner, Extra Tree (ET) (mean aggregation) [18] and Random Forest (RF) (majority voting aggregation) [8].

Another way to build an ensemble is to use *boosting*, in which a set of weak learners are combined to create a single strong learner [46]. The weak learners are successively trained

[7]In the literature, this set is referred to as the "Hypothesis Space". Since we use the term hypothesis for explaining the hypothesis test, we use "candidate space" instead.

and at each round more weights are given to the wrongly classified instances. Three popular approaches of this category are Ada Boost (AB) [15], Gradient Boosting (GB) (uses better optimization) [17] and Histogram Gradient Boosting (HGB) (faster than GB) [28]. All ensemble approaches mentioned here take DT as the base classifier or weak learner to build the complete binary classifier. These classifiers are explained in more detail in Section A.

Hyperparameter Optimization (HPO): Hyperparameters are used by the learning algorithm to control the learning process. ML algorithms, especially tree-based and boosting approaches have different *hypeparameters* apart from the model-parameters, that should be set for an algorithm, such as for RF the number of estimators, maximum depth of each DT, for linear models the learning rate of the LR, PLA and so on. The hypeparameters, define the structure of a learning model, and the value of each hypeparameter has a huge impact on the performance of algorithms and these parameters are dependent on each [22]. This motivates us to find the right combination of their values which can help us to achieve a learned model that produces minimum loss or maximum accuracy. The class of optimization approaches used to solve the task of choosing a set of optimal hypeparameters for a learning algorithm is called HPO and the approach itself is called Hyperparameter Optimizer (HPOR) [22]. Typically, an Hyperparameter Optimizer (HPOR), defines a range of possible values each hypeparameter can take, based on the given algorithm and it runs for a given number of maximum iterations. For each iteration, it applies one combination of hypeparameter values from this defined range and evaluates its performance using a loss function. Generally, we set aside a validation dataset sampled from the training dataset, for performance evaluation the loss-function or metric used is called the *validation loss* or *validation accuracy*. For binary classification the error-rate is used as the validation loss, as defined above as ℓ_{acc}. In this paper, we use Bayesian Optimization techniques with the Gaussian Processes to perform the HPO for different binary classifiers [14].

c) Hypothesis Tests: Comparing the performance of two algorithms is a common problem in ML. In our approach, we want to compare the binary classifiers listed in the previous section with the Random Guessing (RG) baseline. For a given dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i), \ldots\}$, RG generates each class-label ($y_i \in \{0, 1\}$) uniformly at random without using the input $\boldsymbol{x}_i$. Assuming that the proportion of class-label 0 and 1 is equal in the given dataset, the accuracy of approximately 50% implies that the input features $\boldsymbol{x}$ are not used to predict corresponding class-labels $\hat{y}$. For $K = 2$ the RG produces an estimated average E_{out} or accuracy of 50%. So, one can say that if a binary classifier is using input features $\boldsymbol{x}$ to predict the corresponding class-labels $\hat{y}$, its estimated out-of-sample accuracy would be greater than $50\,\%$ ($E_{out} > 0.5$). A binary classifier can only produce accuracy lower than $50\,\%$ if there exists noise in the dataset, i.e. if our testing dataset is not large enough or the number of accuracy estimates n_s is very low. Therefore, we use the MCCV approach to produce a large number of ($n_s = 30$) estimates, with 30% of the data used for testing ($ts = 0.7$) [4].

One way to imply that the binary classifier can learn something about the class-labels using the input vectors $\boldsymbol{x}$ is to consider the differences of the mean accuracies of RG and the given binary classifier. However, this approach can be misleading as it is hard to know whether the difference between the mean accuracies ($1 - E_{out}$) is real or a result of a statistical fluke. For this reason, well-established techniques compare two resulting populations, rather than just their mean differences. For comparing the performance of two classifiers, there are two statistical tests which are proposed in the literature, the paired t-test [12] and the Wilconson-Signed Rank test [54].

Paired t-test: The paired t-test is used to study if there is a statistical difference between two samples observed from two populations. Mathematically, it approaches the problem by assuming a null hypothesis $H_0(a_j == a_{rg})$. After applying the t-test, if the null hypothesis $H_0(a_j == a_{rg})$ is rejected, it indicates that the groups are different with high probability. Here, $a_j, \forall j \in \{1, \ldots, 10\}$ represents the population of the out-of-sample accuracy estimates of the binary classifier j and a_{rg} represents the population out-of-sample accuracy estimates of RG [7, 12].

The t-statistic value is used with the Student-t distribution with $N - 1$ degrees of freedom to quantify the p-value by calculating the area under the t-distribution curve at value t, which is computed as

$$\mu = \frac{1}{N}\sum_{i=1}^{N}(d_i = a_i - rg_i), \quad \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}\frac{(\mu - d_i)^2}{N-1}, \quad t = \frac{\mu}{\sigma}. \tag{1}$$

The p-value p_j is the probability of obtaining test results at least as extreme as the results observed during the test, assuming that the null hypothesis is correct [12, 31]. The critical value α of a statistical test defines the boundaries of the acceptance region of the test [12, 31], i.e. if we nhave p-value $p_j < \alpha$ for binary classifier c_j, then the null hypothesis $H_0(c_j == c_{rg})$ is rejected and we can say that c_j is significantly different/better than RG.

Corrected Paired t-test: A problem with using the Paired Student's t-test (and Wilconson-Signed Rank) is that the accuracy estimates of the binary classifiers are not independent. This is because the same data is used to train the model multiple times. This lack of independence in the evaluation means that the Paired Student's t-Test is optimistically biased. Nadeo and Bengio [39] showed that the violation of independence in the paired t-test might lead to an underestimation of the variance of differences. To solve this problem with the paired Student's t-test, they propose to correct the variance estimate by taking this dependency into account. The corrected variance is given by $\sigma^2_{Cor} = \sigma(\frac{1}{N} + \frac{ts}{1 - ts})$, where ts is fraction of training datasets used by MCCV [4, 39]. The t-statistic is calculated in the similar manner as in Equation (1), such that $t = \frac{\mu}{\sigma_{Cor}}$.

Holm-Bonferroni: In statistics, this method is used to aggregate the the p-values of multiple hypothesis tests [23]. To describe the process, we assume that we compared J classifiers with RG and produced p-values $p_1, \dots, p_J$ and the corresponding hypothesis are $H_1, \dots, H_J$. The significance level is defined for complete family, $\alpha = 0.01$. For each p-value, test whether $p_j < \dfrac{\alpha}{J+1-j}$, If so, reject H_j and the index j identifies the first p-value that is not low enough to validate rejection. Then the rejected hypotheses are $H_1, \dots, H_{j-1}$ and the accepted hypotheses are $H_j, \dots, H_J$. For our experiments, $j \geq 2$ (i.e., if at least one hypothesis is rejected) implies that performance significantly better than RG was achieved. In this case, we conclude that there exists a mapping between input space $\boldsymbol{x}$ to class-labels y, and reject the null hypothesis, for the family of classifiers. If $j = 1$ then no p-values were low enough for rejection, therefore no null hypotheses $H_1, \dots, H_J$ are rejected (i.e., all null hypotheses are accepted). In this case, the analysis result will be given as "vulnerable", while the analysis concludes with "non-vulnerable" otherwise.

d) Feature Importance: Feature importance refers to techniques that assign a score to each input feature based on how useful they are for predicting the class-label $\hat{y}$. Generally, tree-based binary classifiers like, DT are used to calculate feature importance. For instance, it can be used in real-world applications like credit card fraud detection to identify persons that are more likely to commit fraud. The DT quantifies feature importance based on the depth of the node on which the feature was used to split the data. The lower the depth, the higher the importance. See Appendix A for details. RF uses many individual DT for training and each DT calculates the feature importance. Then we take the mean of the feature importances. The advantage of taking the mean value is to reduce the variance and noise in the calculation.

IV. Implementation of Automated Side Channel Detection

In this section, we describe how we implement our proposed approach in a software tool. The tool consists of the components shown in Figure 3, running after each other in four discrete stages:

Stage 1: The manipulated TLS client connects to the TLS server which is to be tested. The client then executes a pre-configured number of requests with manipulated padding while a network tap records all the exchanged messages. We describe this client in Section IV-A. We treat the server as a black box and run it in a docker container. For the network tapping, we record all traffic on the respective docker interface with tcpdump.

Stage 2: The raw data recorded in Stage 1 is transformed into a dataset that is suitable for machine learning. The feature extractor achieves this by extracting real-valued features from the messages contained in the network trace. It also matches these handshakes with the manipulation information from the manipulated TLS client, labeling each handshake with the associated padding manipulation. This is described in depth in Section IV-B.

Stage 3: The dataset is used to train and test classifiers. After training, the learned models are executed on the test data sets, testing the accuracy of their predictions. This process is repeated for different splits into test and training sets and the overall accuracy is determined. This process is presented in Section IV-C.

Stage 4: The results of the machine learning process are evaluated. The performance of each machine learning model is compared to a simple Random Guessing (RG) algorithm, applying the Holm-Bonferroni test for significance. The final report then contains information on whether a model was able to significantly outperform RG, which would indicate the presence of a padding side channel. Feature importance is also included in the report as feedback to the software developer. This is shown in Section IV-D.

A. Manipulated TLS Client

For Bleichenbacher's attack, we need to recognize patterns in protocol network traffic that make it possible to distinguish messages with incorrect padding from messages with correct padding. Therefore we implement a TLS client that executes handshakes with valid and invalid paddings. The client builds upon TLS-Attacker [50],[8] a Java-based tool that allows for sending arbitrary TLS protocol messages with flexible modifications. It has already been used to detect new Bleichenbacher side channels [50] and implements several modified protocol flows. We use TLS-Attacker in our tool to execute n TLS handshakes (where n is a parameter, we will use later: $n \in \{500, 50000\}$), using a TLS Cipher Suite based on RSA key exchange.

[8]https://github.com/tls-attacker/TLS-Attacker

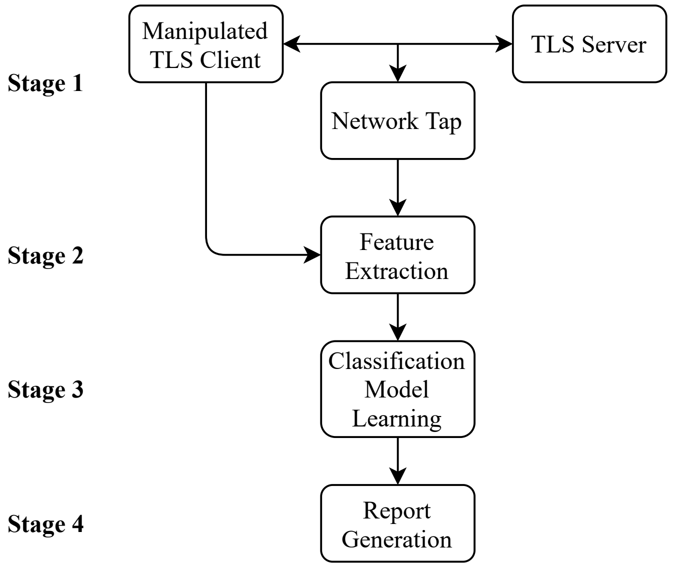

Fig. 3: Components of the tool

The structure of the PKCS#1 v1.5 padded `ClientKeyExchange` (`CKE`) message offers several distinct ways in which a handshake may not conform to the specification. We call each of these deviations a *manipulation*. TLS-Attacker already supports sending `CKE` messages with various padding manipulations. These manipulations include all attack vectors presented by Böck et al. in the ROBOT paper [6], extended with attack vectors by Meyer et al. [36] and Klíma et al. [29]. We use the following manipulations for our experiments:

- Correctly formatted PKCS#1 message: Standard-compliant message, the real *pre-master secret* (`PMS`) replaced with a random string of appropriate length
- Incorrect first byte: Replacing the first byte of the message, which should be 0x00, with a non-zero value (we chose a constant, 0x17)
- Incorrect Second Byte: Replacing the second byte of the message (block type), which should be 0x02, with a different constant (0x17)
- Invalid TLS version in PMS: Setting the TLS version bytes in the payload to an incorrect constant (0x42 0x42, a non-existing version)
- No 0x00 separator byte: Except for the first byte, all other bytes in the padded string that are 0x00 (particularly the separator byte) are replaced with 0x01
- 0x00 in PKCS#1 padding: The second byte of the padding string, which should be non-zero, is replaced with 0x00
- 0x00 in PKCS#1 padding: Replacing the ninth byte of the padding string, which should be non-zero, with 0x00
- `PMS` is the empty string: Placing the 0x00 separator at the last byte, creating a payload of length 0.
- 0x00 On the last-but-one: Placing the 0x00 separator at the last-but-one byte, creating a payload of size 1.
- Correctly formatted, but short $|\texttt{PMS}| = 47$: Valid padding for a 47 bytes `PMS`, which should be 48 bytes long
- Correctly formatted, but 1 byte shorter: An otherwise correctly padded message, but with the the total length being one byte too short (not matching the RSA modulus size)

This aims to cover the majority of classes of padding errors one might expect. Of course, completeness cannot be guaranteed, but the set of considered manipulations can be easily extended, if considered useful in a particular context.

The client chooses at random whether and which manipulation to apply to a ciphertext. It then executes the handshake with the (manipulated) ciphertext. This process is repeated n times. The client logs which padding manipulation from the list above was applied to each handshake, which will be used in Stage 2 to match the observed network traffic to the manipulation. For example, when choosing "Incorrect first byte", the client manipulates the padding by replacing the first byte (which should be `0x00`) with a different constant. For the handshake message corresponding to a correct padding, the `PMS` is replaced with randomness, in order to ensure that the handshake will still fail as soon as the `Finished` (`FIN`) message is processed. This corresponds to a step in the Bleichenbacher attack where the decrypted message has correct padding by chance, but where the actual `PMS` contained has not been found yet. This way of randomly selecting a manipulation for each new handshake eliminates information leakage from the order of execution of the handshakes while producing balanced datasets for sufficiently large dataset sizes.

Our client also supports several different "workflows" of the TLS handshake. The first workflow we use is that of a "regular" handshake, consisting of `CKE`, `ChangeCipherSpec` (`CCS`), and `FIN` messages. It is also necessary to test with a shortened workflow of a single `CKE`, without a `CCS` or `FIN` message, to trigger some vulnerabilities discovered in ROBOT [6]. The workflow (full or shortened) and the padding manipulation are selected at random when executing a handshake.

Note that the missing `CCS` and `FIN` messages can result in server connection timeouts. Because of the timeout caused by the shortened workflow waiting for a potential response from the server, this becomes a limiting factor in client throughput. Consequently, the timeout for the client disconnecting from an idle session needs to be set high enough to not miss any messages from the server. We used a timeout of one second for experiments in a local environment and three seconds for remote servers, which turned out to be appropriate for the respective network delays and gave the analyzed TLS libraries enough time for their responses.

B. Feature Extraction

The "padding oracle" in Bleichenbacher's attack is essentially an abstraction of a way that enables the attacker to efficiently distinguish whether the tested server acts differently on validly or invalidly padded ciphertexts. We consider an attacker that is able to observe and record the entire network traffic. Therefore we need to ensure that the same information is available to the machine learning (ML) algorithms, including the labeled padding modifications performed by the TLS client.

The data obtained in Stage 1 is the traffic exchanged between the manipulated TLS client and the TLS server, recorded using tcpdump. This results in a .pcap file containing all messages in their original binary representation, as well as their metadata. The feature extractor transforms the raw data into a feature representation that is applicable for training a classifier. A standard approach is to use datasets consisting of labeled real-valued vectors, where the label contains the class the particular vector (called "instance") belongs to.

In our case, each handshake corresponds to a single instance in the dataset, with the padding manipulation used as the class label for the instance. Thus, handshakes with the same manipulation end up as instances of the same class in the dataset. An instance has to be represented by an n-dimensional real-valued vector, where each dimension corresponds to a feature. Hence, we have to reduce all network messages belonging to a handshake to a single vector. This is necessary because these messages are intrinsically linked with each other through the handshake process and have to be treated as a single entity in ML, in order to be able to capture patterns exhibited by combinations of messages.

To achieve this transformation, we build upon the popular network analysis tool Wireshark, which our tool automatically interacts with using the Python library pyshark. By taking all the protocol fields from the Wireshark output, we can transform the messages into real-valued feature vectors compatible with ML.

The result of this transformation is a vector of high dimensionality. We need to make sure its dimensionality is not too high, as this affects the performance of most ML algorithms negatively due to a phenomenon often called the "curse of dimensionality" or the "peaking phenomenon" [25, 51]. The peaking phenomenon states that the predictive power of a learning model (classifier) first increases with the number of features in the dataset, but after a certain number it starts deteriorating instead of improving steadily [51]. One way to mitigate this problem is by increasing the size of the dataset to at least 5 training instances for each dimension in the dataset, or by reducing the dimensionality of the feature vectors [51].

The dimensionality of our dataset is higher, if the feature vector contains more messages or more network protocol fields. We can discard all messages sent by the server before it receives the manipulated Client Key Exchange itself, as they cannot be influenced by the padding manipulation. This reduces the number of messages contained in the vector reducing the dimensionality at the same time.

We also chose to only export data on the TCP and TLS layer in our experiments. Our decision was based on the previous works on side-channel attacks [6, 35], which only detected behavioral differences on the TCP layer and above. Our reductions dramatically reduce the feature dimensionality, making the experiments feasible with limited resources.

Note that in this first step of the development of our approach, we do not yet consider timing of messages, even though several attacks construct a padding oracle using timing. The treatment of timing information requires further consideration. We decided to configure the current feature extractor to filter out all timing-related features in our experiments to prevent accidental leak of padding status from the client side. A non-constant-time client implementation (like TLS-Attacker) could inadvertently leak information about the used padding manipulation into the timing features, causing false positives.

Finally, because we are using supervised machine learning, we label the vector with the padding manipulation applied by the client in this handshake.

C. Classification Model Learning

In this section, we discuss the steps taken by the ML component in order to determine the existence of a pattern in the dataset, which would imply the existence of a side channel. This process is illustrated in Figure 4.

a) Multi-class to binary conversion: The dataset obtained in Stage 2 consists of handshakes, involving either a correctly padded message or one of 10 plaintext manipulations. When using a multi-label classifier on the dataset, we observed that the performance was poor. This was because of the fact that many different manipulations may lead to the same server behavior. In its stead we formulate K binary classification problems, since we are interested in distinguishing any of the K manipulations from the correct padding, as this indicates information leakage. Recall that every instance $\boldsymbol{x}$ is associated with a discrete valued class-label $y \in \{0, 1, \ldots, K\} = [K]$. It takes value 0 when there is no manipulation, and values between 1 and K are directly related to the types of manipulation. Our binary classifier considers manipulation k, where $1 \leq k \leq K$ and tries to distinguish it from the correct plaintext for all K manipulations individually.

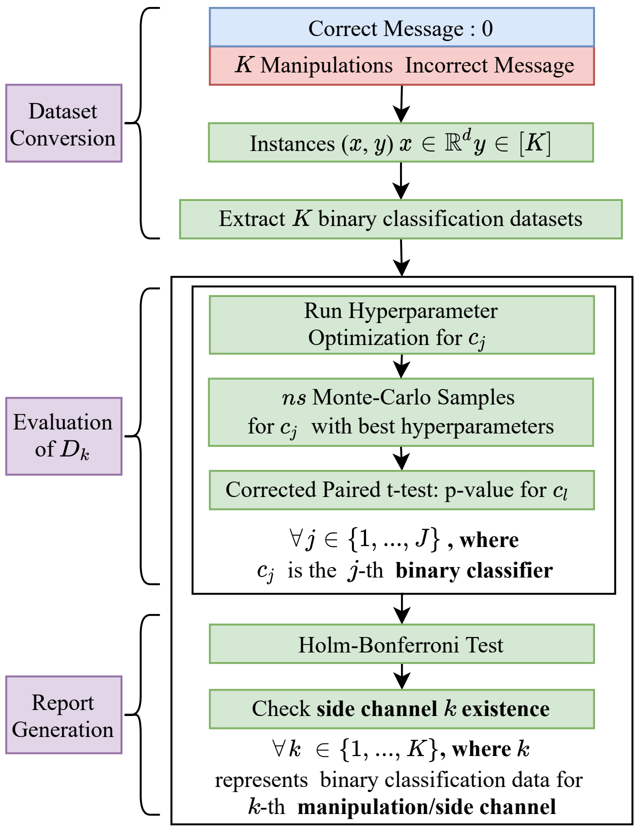
Dataset Conversion
Correct Message : 0
K Manipulations Incorrect Message
Instances (x, y) $x \in \mathbb{R}^d$ $y \in [K]$
Extract K binary classification datasets
Evaluation of D_k
Run Hyperparameter Optimization for c_j
ns Monte-Carlo Samples for c_j with best hyperparameters
Corrected Paired t-test: p-value for c_l
$\forall j \in \{1, ..., J\}$, where c_j is the j-th binary classifier
Report Generation
Holm-Bonferroni Test
Check side channel k existence
$\forall k \in \{1, ..., K\}$, where k represents binary classification data for k-th manipulation/side channel

Fig. 4: Classification Model Learning. As an input into our algorithm, we use a d-dimensional real-valued feature vector $\boldsymbol{x} \in \mathbb{R}^d$ generated with TLS handshakes with one correct and $K = 10$ incorrect messages. The algorithm detects a side channel based on the results from the used binary classifiers.

b) Evaluation: At the core of the process described in Figure 4 is a familiy of binary classifiers that are trained to distinguish between correct padding and a specific manipulation. Evaluating the performance of the classifiers outside the given dataset is pertinent, since we do not want the classifiers to overfit the given dataset. Within our framework, overfitting refers to the problem of finding a pattern within the dataset that does not exist in general. We use Monte

Carlo cross-validation (MCCV) technique for this purpose, see Section III-B for details. Another key step in training a binary classifier successfully is to tune its hyperparameters, for this we use Hyperparameter Optimization (HPO), see Section III-B for details. Finally, note that we train multiple classifiers independently, to increase the robustness of our verdict.

Since the binary classifier needs to be significantly better than the RG to imply information leakage, we apply the corrected paired t-test from statistics, see Section III-B for details. This test provides a t-score and a p-value. The t-score quantifies the performance difference between the binary classifier and RG. The associated p-value gives us the confidence that the t-score represents the true performance difference. Note that a better performance than RG typically implies a high classification accuracy. To summarize, we return the p-values of the family of binary classifiers for each manipulation, which are subsequently interpreted in stage 4. In order to conduct the experiments in a fair and unbiased way, we use a family of classifiers listed in Section III-B. We use the scikit-learn [40] library, which implements the required ML algorithms, cross-validation (CV) techniques and evaluation measures. For HPO we also use the scikit-optimize [19] library, which implements different Hyperparameter Optimizers (HPORs) compatible with the binary classifiers implemented in scikit-learn [40].

D. Error Correction and Report Generation

As explained in the previous section, in order to increase the test robustness, we use an ensemble of binary classifiers. Because it is not known in advance which classifier might perform particularly well on a new, unknown side channel, trying several in an ensemble increases the chance that a suitable one is among them.

a) Family-wise error rate: Since we have multiple classifiers, the false positive rate when combining them is higher than if only a single one were used, as a single classifier outperforming the RG is already sufficient to conclude that implementation is vulnerable. The p-values obtained from the multiple independent tests of each classifier, therefore, need to be adjusted and then aggregated to give us a final verdict on the vulnerability of the server with respect to the manipulation. We use the Holm-Bonferroni test [23] as described in Section III-B to correct for this error.

b) Feedback: We aim to provide detailed information about any detected side channels to support software developers in removing them. The first information we provide is which manipulations were distinguishable from correct padding. This indicates which padding check is at fault. Another important part of the report is whether this side channel was detectable with the full (`CKE`, `CCS`, `FIN`) handshake or the shortened one (`CKE` only). The random forest classifier additionally provides information about the importance of features in the dataset. This can provide valuable feedback for the software developer about which parts of a network trace need to be investigated and what the cause of the potential padding oracle vulnerability could be. For example, as explained in the analysis below, a different TLS alert message in response to a padding failure can be easily pointed out, since this is the single most important feature in the dataset.

V. Analysis

We explore the capabilities of our proposed methodology by applying the tool described in Section IV to various TLS server implementations. In Section V-A, we explain our experimental setup. In Section V-B, we describe the results of a first basic validation of the approach, which executes our tool against a completely insecure implementation, and one implementation which is considered secure. The test confirms that the vulnerabilities are found and that the secure implementation does not exhibit any noticeable patterns. In Section V-C we continue the analysis by testing the ability to spot diverse real-world side channels. To this end, we investigate the side channels discovered in ROBOT [6] and the side channel in OpenSSL Version 0.9.7a from [29]. Again, the analysis confirms that the tool can successfully and reliably detect all of these side channels. Finally, in Section V-E we also present the results of an analysis of the most recent versions of a large number of popular open-source implementations, which does not yield any new unknown vulnerabilities.

A. Test Setup

The tool performs and captures $n \in \{500,\ 50000\}$ handshakes. Usually we run 50,000 handshakes, since machine learning (ML) algorithms often tend to classify more reliably based on larger datasets. For the ROBOT attack, we experimented with 500 handshakes, as this size is better suited for a large-scale scan of public servers and more appropriate for a quick scan as part of automated regression testing. Depending on the network conditions and timeout settings, the 500 handshakes can be completed in as few as fifteen seconds, with the feature extraction taking about ten seconds. The training process of the classifiers is by far the most time-consuming stage of the process, but for a dataset of this size it remains under three minutes. In comparison, the overall running time for the big 50,000 handshake datasets is just under two hours.

Our analysis showed that this is a reasonably large number of requests to provide a sufficient confidence in the analysis result. Detailed analysis of the proper dataset size is covered in Appendix B. All handshakes are executed sequentially without any waiting time.

The servers are running on the same machine as the rest of the setup, isolated in docker containers. The network traffic is captured on the docker virtual network interface, which ensures there is no interfering traffic. Since both server and client run on the same physical machine, we introduce an artificial $2\,\mathrm{ms}$ delay on the interface. Otherwise the very low delay could cause aggressive TCP retransmissions, which appear as noise in the collected dataset and make the automated detection of patterns more difficult and less reliable. The $2\,\mathrm{ms}$ delay produces TCP behavior which is closer to the real world, while still being low enough not to negatively influence the timeout of the client, which is set to $50\,\mathrm{ms}$, or overall performance.

The tool is executed on an otherwise idle workstation with an AMD Ryzen 9 3950X 16-core CPU, 64GB RAM and an Nvidia RTX 2080 Super GPU, running Debian 10.7 and Python 3.7.3.

When executing the machine learning component, we get 30 accuracy estimates samples using $ns = 30$ Monte Carlo cross-validation (MCCV) train-test datasets. This sample size allows applying the Holm-Bonferroni test at a significance level of $\alpha = 0.01$, which is shown as a horizontal bar in the figures. This 99% confidence threshold appears to be a good tradeoff between false positives and false negatives.

B. Basic Approach Validation

We use the newest OpenSSL version 1.1.1i as our baseline for a non-vulnerable server implementation, as it is considered to be secure against Bleichenbacher padding oracle attacks, based upon the scrutiny of both the open source community as well as the attention of security researchers. Our baseline for a vulnerable server implementation is DamnVulnerableOpenSSL[9], a patched TLS implementation which intentionally contains a padding oracle vulnerability for experimental and educational purposes.

Figure 5 shows how the different classifiers perform on our two baseline datasets. For the non-vulnerable OpenSSL 1.1.1i, all classifiers achieve an accuracy of approximately 50%. Consequently, none of the p-values exceeds the threshold for acceptance, i.e., to be considered potentially vulnerable. For DamnVulnerableOpenSSL, some classifiers perform quite well while some others, like Perceptron Learning Algorithm (PLA) or Support Vector Machine (SVM), do not outperform Random Guessing (RG). This is also reflected in the associated p-values. The performance of the classifiers Decision Tree (DT) and Extra Tree (ET) is somewhat higher than 50%. However, their p-values indicate that we cannot rule out that this happened by chance. The performance of other classifiers, e.g Random Forest (RF), which significantly outperform RG when applying the Holm-Bonferroni correction, means that we can still conclude this server is vulnerable. We consider this as support of the approach of using many different machine learning algorithms in parallel.

Table I shows the most important features in the random forest algorithm for DamnVulnerableOpenSSL. This is the feedback that the tool provides to a software developer on the detected side channel. The highest importance is associated to the TLS alert message. Since DamnVulnerableOpenSSL was specifically crafted to return a different TLS alert for incorrect paddings (`handshake failure` instead of `bad record mac`), this confirms that the tool provides correct feedback.

Our approach was also able to spot another subtle change in the error handling; in case of a padding failure, the server disconnects the TCP connection slightly differently, by sending a TCP reset right after a TCP finished message, instead of waiting for the client to acknowledge the TCP finished. This causes the `RST` TCP flag on the second disconnect message to be set only when processing incorrect padding, which consequently appears as the third most important feature in telling the two classes apart. This behavior also causes a shift in the acknowledgement numbers, which was detected and used by the random forest algorithm, as indicated by the second-highest feature importance score. Hence, the approach is also capable of identifying such subtle side channels.

Feature Name	**Importance**
Description of first TLS Alert	1.0
TCP ACK number of 2nd TCP Disconnect	0.93
TCP RST of the 2nd TCP Disconnect	0.90

TABLE I: Most important features leading to a side channel in DamnVulnerableOpenSSL extracted automatically using the random forest algorithm.

C. Detecting Klíma-Pokorný-Rosa Side Channels

As a next step, we analyzed the detection of real-world side channels in old implementations. According to the OpenSSL changelog,[10] prior versions of this implementation exposed several side channels. The first changes were applied prior to version 0.9.5, where Bleichenbacher's attack was fixed after publication of the original paper in 1998. As mentioned in the release notes, this fix was not sufficient, as the error caused by a padding failure was not properly ignored. Even worse, the countermeasures were accidentally removed in version 0.9.5. This is explained in the release notes for version 0.9.6b (released in 2001), claiming that this version then contained the first working protection against Bleichenbacher's attack. Unfortunately, the source code of these versions is no longer available for download.

Versions 0.9.6j and 0.9.7b (released in 2003) then contained another change, this time to address the recently published Klíma-Pokorný-Rosa bad version oracle attack [29]. Version 0.9.7a (vulnerable) and 0.9.7b (not vulnerable to the Klíma-Pokorný-Rosa attack) are available and can be compiled, so we use them to analyze the approach when faced with a bad version side channel.

TLS handshakes with manipulated `ClientKeyExchange` (`CKE`) messages containing an incorrect TLS version resulted in different server behavior of version 0.9.7a. This was correctly detected by our tool. The classifiers were able to significantly outperform RG. The tool consequently detected the Klíma-Pokorný-Rosa bad version side channel present in 0.9.7a. We then tested 0.9.7b and no classifier was able to outperform RG, with the tool returning a "not vulnerable" result. This indicates that the applied countermeasures are successful in preventing these side channels.

[9]https://github.com/tls-attacker/DamnVulnerableOpenSSL

[10]https://www.openssl.org/news/changelog.html

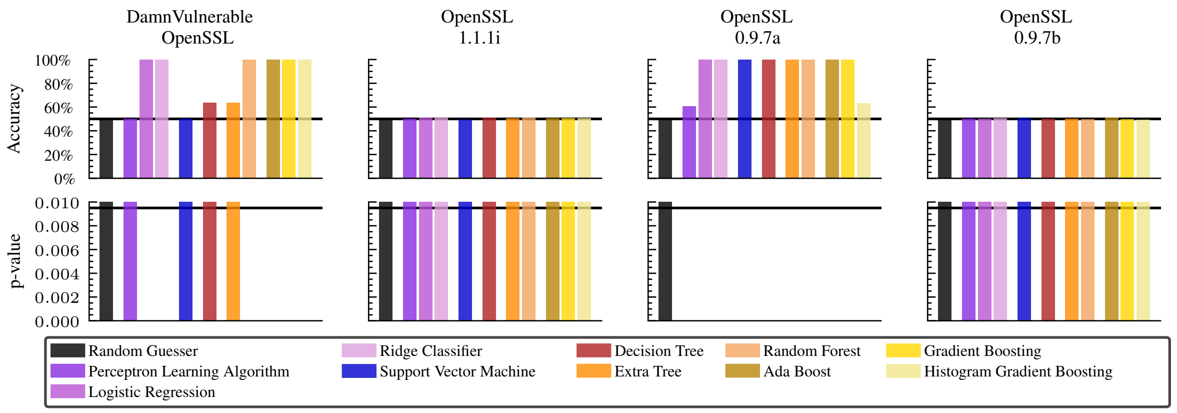

Fig. 5: Performance of different classifiers. The two plots on the left are obtained in the basic approach validation, considering DamnVulnerableOpenSSL and OpenSSL 1.1.1i and DamnVulnerableOpenSSL. The two plots on the right are obtained in the analysis of the Klíma-Pokorný-Rosa attack [29].

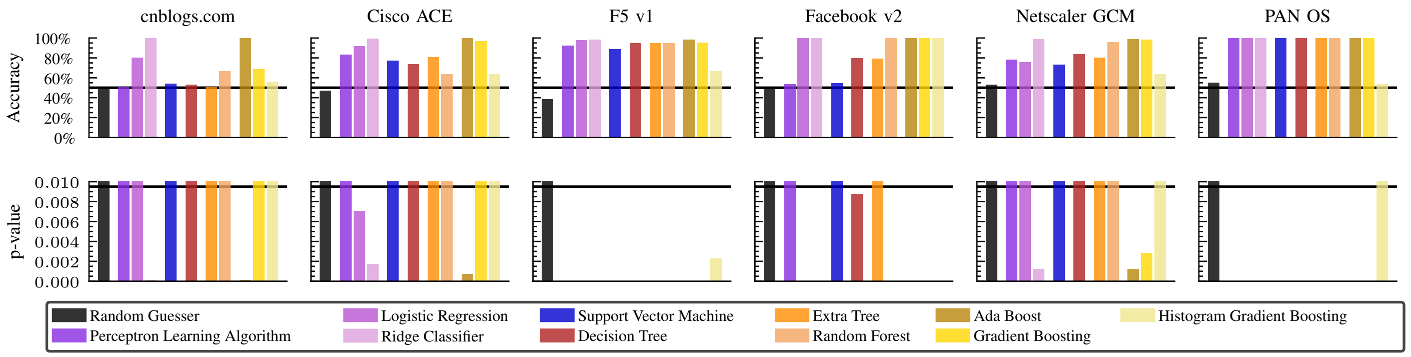

Fig. 6: Performance of different classifiers for the ROBOT servers generated with manipulated `ClientKeyExchange` (CKE) messages containing 0x17 instead of 0x00 as the first byte of the PKCS#1 v1.5 padding. Each server was successfully labeled as vulnerable by at least one of the used classifiers.

D. Detecting ROBOT Side Channels

Another range of real-world side channels to apply our tool on was presented in the ROBOT paper [6], where Böck, Somorovsky and Young found numerous side channels in public web servers. Most of these were found in closed source TLS server implementations. The authors informed the affected vendors, most of which published updated software versions for their devices. Consequently, we expect these side channels to be no longer present in most web-facing TLS servers.

To verify our approach is also suitable for large-scale internet scans, we ran our tool on the domains in the Alexa Top 500 web page ranking. Because this scan necessitates connecting to machines outside of our control, the workstation used for the scans serves an informational web page with contact information for concerned server operations that want to opt out of the scans. The tool was set up to execute an independent test consisting of 500 handshakes with each of the domains in the Alexa Top 500 ranking. For 5 domains the hypothesis test rejected, indicating a possible vulnerability. To investigate further, a larger test with 50000 handshakes was executed with these servers. This revealed that 4 of the 5 were false positives. The fifth, cnblogs.com, is a genuinely vulnerable server, with the result of the initial 500-handshake scan shown in Figure 6. This was confirmed using the established ssllabs[11], tls-scanner[12] and testssl.sh[13] TLS scanners, which were able to detect the side channel as well. We then reached out to the server operators and notified them of the issue.

This confirms that our approach scales to hundreds of web servers, albeit at the cost of an increased running time compared to other ROBOT scanners. On the other hand, our approach requires less assumptions about the nature of the side channel. Additionally, this test also proves that the tool can be applied outside of lab conditions in the real world, where network behavior has a bigger influence on the recorded traffic traces.

We have thus determined that almost all of the side channels covered in ROBOT have since been removed from high-profile web servers. To evaluate if our tool would be able to detect

[11]https://www.ssllabs.com/ssltest
[12]https://github.com/tls-attacker/TLS-Scanner
[13]https://testssl.sh/

Name	**Version**
BearSSL	0.6
BoringSSL	commit 3743aafd
Botan	2.17.3
Bouncy Castle	1.64
GnuTLS	3.7.0
LibreSSL	3.2.3
MatrixSSL	4.3.0
Mbed TLS	2.25.0
OCaml-TLS	0.12.8
OpenSSL	1.1.1i
s2n	0.10.25
tlslite-ng	0.8.0-alpha40
wolfSSL	4.4.0

TABLE II: Open source TLS servers tested

these side channels nonetheless, we deliberately recreated them by imitating their behavior in modified versions of mbedTLS. We decided to imitate five ROBOT vulnerabilities that cover a representative set of server behaviors:

- When F5 v1 (CVE-2017-6168) and Facebook v2 are tested with the reduced workflow of a single `CKE` message, this should result in a TCP timeout caused by the server waiting for the `ChangeCipherSpec` (`CCS`) and `Finished` (`FIN`) message. For incorrect paddings, these servers do not wait but abort prematurely with a TLS alert or a TCP disconnect (TCP finished message).
- Cisco ACE (CVE-2017-17428) responds with a TLS alert 47 instead of alert 20 if the padding check fails.
- Citrix Netscaler (CVE-2017-17382) gives a TLS alert 51 for correct padding, but does not send any data for incorrect padding, causing the TCP session to time out.
- PAN OS (CVE-2017-17841) sends the same TLS alert 40 in both cases, but also sends a duplicate of the alert in case of a padding failure.

Again, we performed 500 handshakes to generate the datasets as we did in the Alexa 500 scan. As can be seen in Figure 6, the tool was able to correctly classify all servers as vulnerable. This was confirmed by a Holm-Bonferroni test, with at least one classifier significantly outperforming RG in each experiment.

E. Testing open-source implementations

After establishing that our method is indeed able to detect known side channels, we applied it to up-to-date open source TLS server implementations. Table II shows all software versions we investigated. For this experiment, we executed 50.000 handshakes each.

After applying the Holm-Bonferroni correction, we concluded that no classifier significantly outperformed random guessing for any of the servers we tested. We therefore conclude not a single of these TLS servers is vulnerable to a padding oracle attack.

VI. Conclusions and Open Problems

We propose, implement, and analyze an approach to automatically detect protocol-level side channels in implementations of cryptographic protocols such as TLS. We consider Bleichenbacher's attack as a concrete use case, due to its repeated appearance in many popular open-source implementations and widely-used commercial products.

A major advantage of our approach is that the side channel vulnerability detection is fully automated, robust, and therefore scales much better than manual analysis. In particular, it could be applied as a standard test before every release of a new version of an implementation, possibly automatically in a CI/CD pipeline. This would also prevent the accidental removal of countermeasures, as happened in version 0.9.5 of OpenSSL, for example.

Our analysis confirms that this approach, despite being fully automated, is able to reliably recognize the patterns in network traffic on which the padding oracles identified by Klíma, Pokorný, and Rosa [29] and ROBOT [6] are based. We did not yet find new weaknesses in the popular TLS implementations that we have analyzed with our tool. However, we were able to reliably detect the aforementioned padding oracles with a general and generic approach, which provides confidence that it will also work for future, new side channels that exhibit distinguishable patterns on the network layer.

In our analyses, we showed that a single binary classification algorithm is not reliable enough to detect every side channel; the used classification algorithms performed differently based on the tested TLS server side channels. The robustness of our approach was achieved by the usage of an ensemble of machine learning algorithms for the task of detecting vulnerabilities. This is further reinforced by the use of a family-wise statistical test to greatly reduce the chance that a vulnerability found by our ensemble is an accident. This technical knowledge can be used by designing future approaches for side channel analyses.

We have intentionally focused on machine learning (ML) algorithms amenable to *feature importance* techniques, because we consider an automated tool particularly useful, if it is able to provide concrete feedback on potential vulnerabilities. An interesting future direction could be to investigate other learning methods, such as deep learning (DL) methods, where the *explainability* of detected side channels is an open problem. Handling imbalanced datasets is another challenge that warrants further investigation. Since our tool is based on scikit-learn and will be made publicly available, it provides a good basis for such extensions.

We believe that the techniques developed in this paper are applicable more generally, beyond Bleichenbacher's attack on the network layer. Possible future extensions could additionally take timing or local states of implementations, as in [48, 59], into account by using them as additional features. Furthermore, one could consider other types of padding oracle attacks, such as Vaudenay's [53], or even completely different cryptographic side channel attacks beyond padding oracles.

References

[1] Nimrod Aviram et al. "DROWN: Breaking TLS Using SSLv2". In: *25th USENIX Security Symposium,*

USENIX Security 16, Austin, TX, USA, August 10-12, 2016. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 689–706.

[2] Romain Bardou et al. “Efficient Padding Oracle Attacks on Cryptographic Hardware”. In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 608–625. DOI: 10.1007/978-3-642-32009-5_36.

[3] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. “Automating the Development of Chosen Ciphertext Attacks”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1821–1837.

[4] Yoshua Bengio and Yves Grandvalet. “No Unbiased Estimator of the Variance of K-Fold Cross-Validation”. In: 5 (Dec. 2004), pp. 1089–1105.

[5] Daniel Bleichenbacher. “Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1”. In: *Advances in Cryptology — CRYPTO ’98*. Ed. by Gerhard Goos et al. Vol. 1462. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–12. DOI: 10.1007/BFb0055716.

[6] Hanno Bock, Juraj Somorovsky, and Craig Young. “Return Of Bleichenbacher’s Oracle Threat (ROBOT)”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, p. 17.

[7] Remco R. Bouckaert and Eibe Frank. “Evaluating the Replicability of Significance Tests for Comparing Learning Algorithms”. In: *Advances in Knowledge Discovery and Data Mining*. Ed. by Honghua Dai, Ramakrishnan Srikant, and Chengqi Zhang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 3–12.

[8] Leo Breiman. “Random Forests”. In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32. DOI: 10.1023/A:1010933404324.

[9] Mathieu Carbone et al. “Deep Learning to Evaluate Secure RSA Implementations”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 132–161. DOI: 10.13154/tches.v2019.i2.132-161.

[10] Corinna Cortes and Vladimir Vapnik. “Support-vector networks”. In: *Machine Learning* 20.3 (Sept. 1995), pp. 273–297. DOI: 10.1007/BF00994018.

[11] Jean Paul Degabriele et al. “On the Joint Security of Encryption and Signature in EMV”. In: *Topics in Cryptology – CT-RSA 2012*. Ed. by Orr Dunkelman. Vol. 7178. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 116–135. DOI: 10.1007/978-3-642-27954-6_8.

[12] Janez Demšar. “Statistical Comparisons of Classifiers over Multiple Data Sets”. In: *J. Mach. Learn. Res.* 7 (Dec. 2006), pp. 1–30. DOI: 10.5555/1248547.1248548.

[13] Dennis Felsch et al. “The Dangers of Key Reuse: Practical Attacks on IPsec IKE”. In: (2018), p. 18.

[14] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. “Initializing Bayesian Hyperparameter Optimization via Meta-Learning”. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI’15. Austin, Texas: AAAI Press, 2015, pp. 1128–1135.

[15] Yoav Freund and Robert E Schapire. “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”. In: *J. Comput. Syst. Sci.* 55.1 (Aug. 1997), pp. 119–139. DOI: 10.1006/jcss.1997.1504.

[16] Yoav Freund and Robert E. Schapire. “Large Margin Classification Using the Perceptron Algorithm”. In: *Machine Learning* 37.3 (Dec. 1999), pp. 277–296. DOI: 10.1023/A:1007662407062.

[17] Jerome H Friedman. “Greedy function approximation: a gradient boosting machine”. In: *Annals of statistics* 29 (Nov. 2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451.

[18] Pierre Geurts, Damien Ernst, and Louis Wehenkel. “Extremely Randomized Trees”. In: *Mach. Learn.* 63.1 (Apr. 2006), pp. 3–42. DOI: 10.1007/s10994-006-6226-1.

[19] Tim Head et al. *scikit-optimize/scikit-optimize*. Version v0.8.1. Sept. 2020. DOI: 10.5281/zenodo.4014775.

[20] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. “Applications of machine learning techniques in side-channel attacks: a survey”. In: *Journal of Cryptographic Engineering* (Apr. 2019). DOI: 10.1007/s13389-019-00212-8.

[21] Arthur E. Hoerl and Robert W. Kennard. “Ridge Regression: Biased Estimation for Nonorthogonal Problems”. In: *Technometrics* 12.1 (Feb. 1970), pp. 55–67. DOI: 10.1080/00401706.1970.10488634.

[22] Matthew Hoffman, Eric Brochu, and Nando de Freitas. “Portfolio Allocation for Bayesian Optimization”. In: *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*. UAI’11. Barcelona, Spain: AUAI Press, 2011, pp. 327–336.

[23] Sture Holm. “A Simple Sequentially Rejective Multiple Test Procedure”. In: *Scandinavian Journal of Statistics* 6.2 (1979). Full publication date: 1979, pp. 65–70. DOI: 10.2307/4615733.

[24] Gabriel Hospodar et al. “Machine learning in side-channel analysis: a first study”. In: *J. Cryptogr. Eng.* 1.4 (2011), pp. 293–302. DOI: 10.1007/s13389-011-0023-x.

[25] G. Hughes. “On the mean accuracy of statistical pattern recognizers”. In: *IEEE Transactions on Information Theory* 14.1 (1968), pp. 55–63. DOI: 10.1109/TIT.1968.1054102.

[26] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. “Bleichenbacher’s Attack Strikes again: Breaking PKCS#1 v1.5 in XML Encryption”. In: *Computer Security – ESORICS 2012*. Ed. by David Hutchison et al. Vol. 7459. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 752–769. DOI: 10.1007/978-3-642-33167-1_43.

[27] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM, 2015, pp. 1185–1196. DOI: 10.1145/2810103.2813657.

[28] Guolin Ke et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Vol. 30. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 3149–3157. DOI: 10.5555/3294996.3295074.

[29] Vlastimil Klíma, Ondrej Pokorný, and Tomáš Rosa. "Attacking RSA-Based Sessions in SSL/TLS". In: *Cryptographic Hardware and Embedded Systems - CHES 2003*. Ed. by Gerhard Goos et al. Vol. 2779. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 426–440. DOI: 10.1007/978-3-540-45238-6_33.

[30] John R. Koza, Forrest H. Bennett, and Martin A. Andre Davidand Keane. "Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming". In: *Artificial Intelligence in Design '96*. Ed. by John S. Gero and Fay Sudweeks. Springer Netherlands, 1996, pp. 151–170. DOI: 10.1007/978-94-009-0279-4_9.

[31] Erich L Lehmann and Joseph P Romano. *Testing statistical hypotheses*. Springer Science & Business Media, 2006.

[32] Liran Lerman et al. "Template Attacks vs. Machine Learning Revisited (and the Curse of Dimensionality in Side-Channel Analysis)". In: *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*. Ed. by Stefan Mangard and Axel Y. Poschmann. Vol. 9064. Lecture Notes in Computer Science. Springer, 2015, pp. 20–33. DOI: 10.1007/978-3-319-21476-4_2.

[33] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. "Breaking Cryptographic Implementations Using Deep Learning Techniques". In: *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*. Ed. by Claude Carlet, M. Anwar Hasan, and Vishal Saraswat. Vol. 10076. Lecture Notes in Computer Science. Springer, 2016, pp. 3–26. DOI: 10.1007/978-3-319-49445-6_1.

[34] James Manger. "A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 230–238. DOI: 10.1007/3-540-44647-8_14.

[35] Robert Merget et al. "Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1029–1046.

[36] Christopher Meyer et al. "Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks". In: (2014), p. 17.

[37] Tom M. Mitchell. *Machine learning, International Edition*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[38] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. *This POODLE Bites: Exploiting The SSL 3.0 Fallback*.

[39] Claude Nadeau and Yoshua Bengio. "Inference for the Generalization Error". In: *Machine Learning* 52.3 (Sept. 2003), pp. 239–281. DOI: 10.1023/A:1024068626366.

[40] Fabian Pedregosa et al. "Scikit-Learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12.null (Nov. 2011), pp. 2825–2830.

[41] John C. Platt. "Fast Training of Support Vector Machines Using Sequential Minimal Optimization". In: *Advances in Kernel Methods: Support Vector Learning*. Cambridge, MA, USA: MIT Press, 1999, pp. 185–208. DOI: 10.5555/299094.299105.

[42] "Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)". In: *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B.C.: USENIX Association, Aug. 2021.

[43] Payam Refaeilzadeh, Lei Tang, and Huan Liu. "Cross-Validation". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 532–538. DOI: 10.1007/978-0-387-39940-9_565.

[44] B. Kaliski. *PKCS #1: RSA Encryption Version 1.5*. RFC 2313 (Informational). RFC. Obsoleted by RFC 2437. Fremont, CA, USA: RFC Editor, Mar. 1998. DOI: 10.17487/RFC2313.

[45] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). RFC. Obsoleted by RFC 8446, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919, 8447. Fremont, CA, USA: RFC Editor, Aug. 2008. DOI: 10.17487/RFC5246.

[46] Lior Rokach. "Ensemble-based classifiers". In: *Artificial Intelligence Review* 33.1 (Feb. 2010), pp. 1–39. DOI: 10.1007/s10462-009-9124-7.

[47] Lior Rokach and Oded Maimon. "Decision Trees". In: *Data Mining and Knowledge Discovery Handbook*. Ed. by Oded Maimon and Lior Rokach. Boston, MA: Springer US, 2005, pp. 165–192. DOI: 10.1007/0-387-25465-X_9.

[48] Eyal Ronen et al. "The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations". In: *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2019, pp. 435–452. DOI: 10.1109/SP.2019.00062.

[49] Padhraic Smyth. “Clustering Using Monte Carlo Cross-Validation”. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD’96. Portland, Oregon: AAAI Press, 1996, pp. 126–133.

[50] Juraj Somorovsky. “Systematic Fuzzing and Testing of TLS Libraries”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1492–1504. DOI: 10.1145/2976749.2978411.

[51] Sergios Theodoridis and Konstantinos Koutroumbas. “Chapter 5 - Feature Selection”. In: *Pattern Recognition (Fourth Edition)*. Ed. by Sergios Theodoridis and Konstantinos Koutroumbas. Fourth Edition. Boston: Academic Press, 2009, pp. 261–322. DOI: https://doi.org/10.1016/B978-1-59749-272-0.50007-4.

[52] Gitte Vanwinckelen and Hendrik Blockeel. “On estimating model accuracy with repeated cross-validation”. In: *BeneLearn 2012: Proceedings of the 21st Belgian-Dutch conference on machine learning*. 2012, pp. 39–44.

[53] Serge Vaudenay. “Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS...” In: *Advances in Cryptology — EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 534–545. DOI: 10.1007/3-540-46035-7_35.

[54] Frank Wilcoxon. “Individual Comparisons by Ranking Methods”. In: *Breakthroughs in Statistics: Methodology and Distribution*. Ed. by Samuel Kotz and Norman L. Johnson. New York, NY: Springer New York, 1992, pp. 196–202. DOI: 10.1007/978-1-4612-4380-9_16.

[55] Qing-Song Xu and Yi-Zeng Liang. “Monte Carlo cross validation”. In: *Chemometrics and Intelligent Laboratory Systems* 56.1 (2001), pp. 1–11. DOI: https://doi.org/10.1016/S0169-7439(00)00122-2.

[56] Hsiang-Fu Yu, Fang-Lan Huang, and Chih-Jen Lin. “Dual Coordinate Descent Methods for Logistic Regression and Maximum Entropy Models”. In: *Machine Learning* 85.1–2 (Oct. 2011), pp. 41–75. DOI: 10.1007/s10994-010-5221-8.

[57] Gabriel Zaid et al. “Ranking Loss: Maximizing the Success Rate in Deep Learning Side-Channel Analysis”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.1 (2021), pp. 25–55. DOI: 10.46586/tches.v2021.i1.25-55.

[58] Jiajia Zhang et al. “A Novel Evaluation Metric for Deep Learning-Based Side Channel Analysis and Its Extended Application to Imbalanced Data”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 73–96. DOI: 10.13154/tches.v2020.i3.73-96.

[59] Yinqian Zhang et al. “Cross-Tenant Side-Channel Attacks in PaaS Clouds”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS ’14*. Scottsdale, Arizona, USA: ACM Press, 2014, pp. 990–1003. DOI: 10.1145/2660267.2660356.

Appendix A
Binary Classifiers

A binary classifier in our application considers two categories: 0 for secure implementation, and 1 for the considered side channel, cf. Figure 4. For our training dataset, we extract real-valued features from the network traffic ($\boldsymbol{x}$) and label each instance with the class-label, which represents the side channel corresponding to the type of error in the message (1) or if the message passed was correct. If there is an information leak or existence of a side-channel in the server under test (SUT), then we can say that a classification algorithm can be used to learn an appropriate function/mapping between the network data and class-label. In particular, each data point is a two-tuple $(\boldsymbol{x}, y)$, where $\boldsymbol{x}$ is the real-valued d-dimensional feature vector extracted from network traffic, and y is its *ground-truth class-label*, see Section III-B. The label y, takes one of two values which is determined by the property of the ciphertext. It is 0 when the padding is correct, and 1 when the padding is incorrect. We say that the SUT is vulnerable when the classification algorithm is able to learn on the binary classification data for the given manipulation, with high accuracy. This in turn implies an ability to distinguish between server reactions for the correct and incorrectly padded plaintext Section III. Hence, when a server is secure, then all binary classifiers will not be able to label the instances $\boldsymbol{x}$ correctly, i.e. the accuracy should be very low. Using this concept, we determine the existence of the side channel with respect to a particular manipulation.

To ensure the robustness of our system, we employ multiple binary classification algorithms and aggregate the corresponding binary classifiers in order to decide on the vulnerability of the SUT. Recall that solving a binary classification task is akin to finding an unknown *target function*. As explained in Section III-B each binary classification algorithm is associated with a *candidate space* for the aforementioned target function. By employing a large set of binary classification algorithms, we are increasing the chance of finding mapping that is very close to the target function. Then we will be searching for the target function within in the union of the *candidate space*. We obtain a set of binary classification models by training the set of binary classification algorithms [37]. Each binary classification algorithm can therefore be thought of as searching its candidate space, for a *predicted function* that fits the given training dataset most accurately [37]. Then we aggregate these learned models to make our final prediction.

a) Linear Models: The first category of algorithms that we use are the linear models. These models assume that the d-dimensional input vector $\boldsymbol{x}$ is linearly dependent on the label, thus falling under the category of linear models. It learns the model-parameters in form of a weight-vector $\boldsymbol{w} \in \mathbb{R}^d$. such that $\boldsymbol{w}\dot{\boldsymbol{x}}$, gives us a real-value which is used to classify the instances as positive or negative. Based on the approach used to learn the weight-vector and interpretation of the obtained real-value, different algorithms are proposed in the literature. Out of them we chose the following algorithms Perceptron Learning Algorithm (PLA) [16], Logistic Regression (LR) [56] and Ridge Classifier (RC) [21].

b) Support Vector Machine (SVM): Since, the underlying function can also be non-linear, we use the more powerful technique like SVM, which uses the kernel methods to learn non-linear objective function [10, 41]. This means that the relationship between the model-parameters $\boldsymbol{w}$ and input instances can be considered as non-linear, for, eg. polynomial. SVM is also robust in the manner that apart from finding a plane separating the positive from negative instances like *linear models*, it tries to increase the margin between the positive and negative instances closer to the plane.

c) Decision Trees: Decision Trees (DTs) are more powerful classification approaches than the linear models and could learn even more complicated functions [47]. In this approach, the goal is to learn a set of simple decision rules inferred from the data features that predict the final class of the instance. DT learns a tree, by splitting the dataset into smaller subsets using a feature at every node. For. e.g, if we are given a set of people, we can split them based on their gender, age, or income. The condition of the split is represented by leaf nodes of the tree and possible outcomes by the branches. This splitting process continues until all the instances are classified with maximum accuracy or maximum depth of the tree is reached. The number of instances that are successfully split by the feature, divided by the total number of instances gives us information gain (IG) for that feature, which can also be seen as the decrease in entropy [47]. IG is used to decide which feature has the highest gain, for splitting the data at every node. Naturally, this would mean that the feature which the highest importance would be nearer to the root node [47]. *Feature importance* is calculated as the increase in the IG divided by the probability of reaching that node. The node probability is the number of instances that reach the node divided by the total number of instances [47].

d) Ensemble Methods: These methods are inspired by the approach of solving a real-life problem by asking multiple experts since it is highly unlikely that each expert will make the same mistake. Based on this the *ensemble methods* were designed, in which a set of learned models $\hat{\mathcal{F}} = \{\hat{f}_1, ..., \hat{f}_M\}$ are used trained on the same data [46]. This set of learned models $\hat{\mathcal{F}}$ is called an "ensemble" and each learner is referred as a *base learner* [46]. At the time of prediction, each members of the "ensemble" is queried with an instance $\boldsymbol{x}_i$, and the final prediction is acquired by aggregating the predictions from these models $\hat{f}(\boldsymbol{x}_i) = AGG(\hat{f}_1(\boldsymbol{x}_i), ..., \hat{f}_M(\boldsymbol{x}_i))$.

Now in principle, these learning models should be independent of each other, but this is a non-trivial issue in machine learning (ML) because the models are eventually trained on the same data. Even then, these methods have been proven to increase the accuracy with large margins [46]. The high accuracy was achieved by introducing the diversification amongst the learned models, by either modifying the learning process or the data for each learner [46].

- ***Bagging*** The data can be modified, by the method of sub-sampling, i.e. by providing each learner a sub-sample of a given training dataset called the *bootstrap*. In bagging, we use a binary classifier as a *base learner* and each one of them is trained on a *bootstrap* dataset. The final prediction of a *Bagging* model is acquired by providing the query $\boldsymbol{x}$ to these learned model $\hat{f}_i, i \in [M]$ and then aggregating the predictions of these models, using a defined aggregation operator $AGG()$ [46]. There are two popular algorithms under this, the Extra Tree (ET) [18] which used mean voting aggregation method to acquire the prediction and Random Forest (RF) which used majority voting [8] and base learner used for learning on each bootstrap is DT.
- ***Boosting*** The main idea behind *boosting*, is to combine a set of weak learners to create a single strong learner [46]. This is achieved by training the first learner on the training dataset, then creating a second classifier that attempts to rectify the errors produced by the first model. The weak learners are added until the perfect performance is achieved on the training set or a maximum number of models are added [46]. These approaches are more susceptible to noise but can learn complex functions [46]. The requirements for the weak learners in boosting are that it should be able to accept weighted training examples (most algorithms can be generalized correspondingly), e.g. shallow DTs (e.g., decision stumps), and linear models. A popular algorithm under this category is Ada Boost (AB), which fits a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases (instances closer to the boundary) [15]. Gradient Boosting (GB) is also similar to AB and it allows for the optimization of arbitrary differentiable loss functions [17]. Histogram Gradient Boosting (HGB) approach is much faster than GB for big datasets and can handle missing values in a dataset [28].

APPENDIX B
MINIMUM DATASET SIZES

For a given learning model, in-sample error (or accuracy) E_{in} is its performance on the training dataset, and the out-of-sample error (or accuracy) E_{out} is its performance on the test data. The plot consisting of the evolution of the two error/accuracy scores as the size of the training set increases are called learning curves. For small n, the in-sample accuracy E_{in} might be much lower since it's quite easy to perfectly fit fewer data points, however, the out-of-sample error E_{out} will be very large. The reason behind this might be that the learning model is built around a small data, and it almost certainly won't be able to generalize accurately on data the learner hasn't seen before. Generally, for the large training set size, the learning model cannot fit perfectly anymore the training

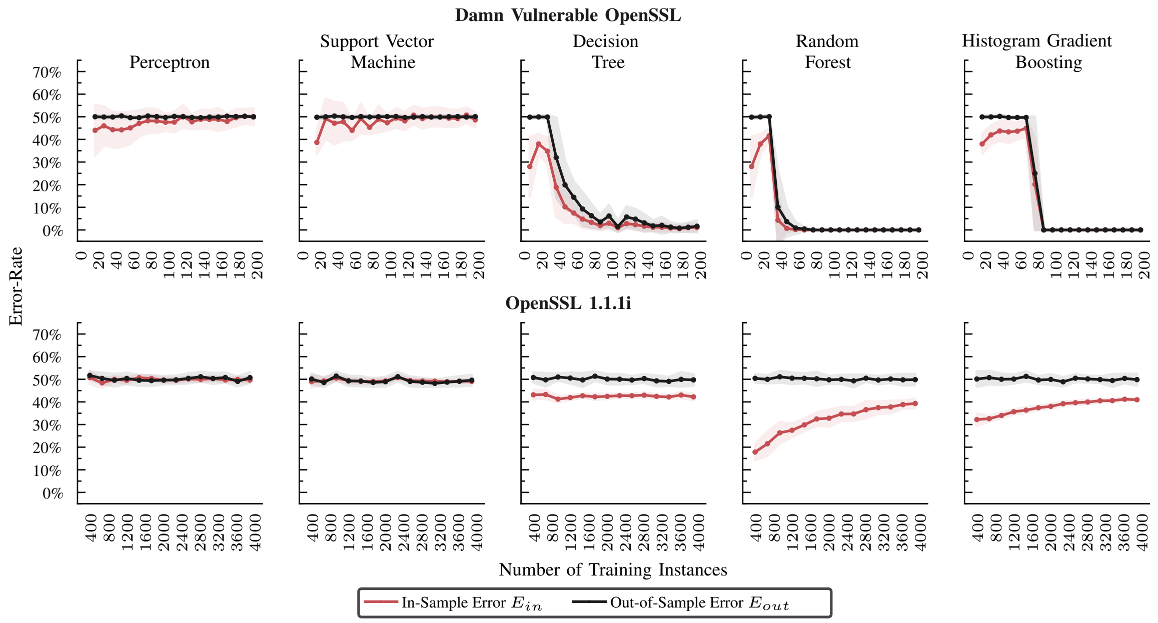

Fig. 7: Learning curves of different classifiers for class-label "Wrong First Byte (0X00 Set To 0X17)" for Damn Vulnerable OpenSSL and Slope 1.1.1g

set and the training error becomes larger, while the out-of-sample error E_{out} decreases, because the model is trained on more data, so it manages to fit the test dataset better. After a certain number of training instances N, the E_{out} stays roughly the same, which implies that adding more training instances won't produce significantly better models. According to the VC-dimension theory, this N represents the sample complexity of an algorithm, i.e. the number of training examples that are required to successfully learn a target function. More precisely, the sample complexity is the number of training-samples that we need to supply to the algorithm, so that the function returned by the algorithm is within an arbitrarily small error of the best possible function, with probability arbitrarily close to 1.

We plot different the learning curves of the subset of binary classifiers, PLA from linear models, DT, SVM, RF from bagging and HGB from boosting models, as shown in Figure 7. As seen in Figure 7, most of the binary classifiers show no significant decrease in the out-of-sample error after a certain number of instances. We can use this concept to determine the number of samples that should be generated by the SUT for each class-label. For OpenSSL 1.1.1i, we can see that the PLAs and SVM require less than 200 instances to converge, i.e. to reach minimum possible error-rate i.e. 0.5 same as Random Guessing (RG). While more complex binary classifiers like DT, RF and HGB are not able to converge even with 3600 instances. So, we cannot say with confidence if the system is not vulnerable, since it can be the case that the vulnerability exists but the existing models are not able to capture it yet. While for the DamnVulnerable OpenSSL server, almost all complex models like DT, bagging and boosting techniques require less then 50 instances to converge, i.e. to reach minimum possible error-rate i.e. 0.0. The learning curves for PLA is interesting as it requires very more 400. As we have seen in Section IV-C, even if one of the classifiers can significantly perform better than RG, after applying the Holm-Bonferroni adjustment, the SUT will be marked as vulnerable to the given class-label. Summarizing these curves, we can imply that if there exists a vulnerability or Side-Channel in the SUT, our approach should be able to detect it for the countable number of instances with high probability. But, we cannot imply non-vulnerability or absence of Side-Channel in the SUT since it might be the case that the side-channel exists but the current models are not able to identify it yet.