End-to-end automated cache-timing attack driven by machine learning

Thomas Perianin, Sebastien Carré, Victor Dyseryn, Adrien Facon, Sylvain Guilley

Noname manuscript No.
(will be inserted by the editor)

End-to-end automated cache-timing attack driven by Machine Learning

Thomas Perianin[1] · Sebastien Carré[1,2] · Victor Dyseryn[1] · Adrien Facon[1,3] · Sylvain Guilley[1,2,3]

Received: date / Accepted: date

Abstract Cache-timing attacks are serious security threats that exploit cache memories to steal secret information. We believe that the identification of a sequence of function calls from cache-timing data measurements is not a trivial step when building an attack. We present a recurrent neural network model able to automatically retrieve a sequence of operations from cache-timings. Inspired from natural language processing, our model is able to learn on partially labelled data. We use the model to unfold an end-to-end automated attack on OpenSSL ECDSA on the secp256k1 curve. Our attack is able to extract the 256 bits of the secret key by automatic analysis of about 2400 traces without any human processing.

Keywords side-channel analysis · cache-timing attacks · machine learning · connectionist temporal classification (CTC) · recurrent neural network (RNN)

1 Introduction

1.1 Context and Related Work

In traditional cryptanalysis, some assumptions are made: the attacker is only expected to interact with the system through its regular interfaces. However, with the increasing development of embedded cryptography, new possibilities of attack appear. They exploit physical information leaking from the device such as timing information or electromagnetic emanations. This new generation of attacks, called *side-channel attacks*, has largely held the research world's attention since the first publication in 1996 about timing attacks [20].

Possibilities of attacks are numerous, given the wide variety of signals that can be disclosed by a device during a sensitive computation: power consumption [19, 24], magnetic field [11], temperature [5] or even sound [1]. The reader is referred to [15] for an extensive introduction about side-channel attacks.

This paper focuses on one specific class of side-channel attacks: *cache-timing attacks*. Those attacks are based on timing information leaked from CPU cache memory. Indeed, when the target algorithm is using sensitive information, it loads secret data into the cache memory. An attacker who can make use of a spy program to indirectly check the content of the cache memory can infer which data has been manipulated by the target algorithm.

Cache-timing attacks were first introduced by Tsunoo et *al.* in [35] to break DES. Later, cache-timing information was used to break AES [3], as well as RSA in a popular implementation of a cache-timing attack: Flush+Reload [38]. In the present paper, we will use an improvement of the latter attack: the Flush+Flush attack [14] which is more stealthy and yields more accurate results.

More specifically, we study in this paper a cache-timing attack on the OpenSSL implementation of ECDSA, an Elliptic Curve algorithm for digital signature. OpenSSL [27] is an open source toolkit for the implementation of cryptographic protocols. The library of functions, implemented using C, is often used for the implementation of Secure Sockets Layer and Transport Layer Security protocols and has also been used to implement OpenPGP and other cryptographic standards.

T. Perianin
[1] Secure-IC S.A.S., Think Ahead Business Line, 35510 Cesson-Sévigné, France
E-mail: thomas.perianin@secure-ic.com

[2] LTCI, Telecom-Paris, Institut Polytechnique de Paris, 75013 Paris, France · [3] Département d'informatique de l'ENS, ENS, CNRS, PSL University, 75005 Paris, France

Breaking various ECDSA implementations has been an active topic in the previous years. The vulnerable part is the Elliptic Curve Scalar Multiplication (ECSM), which is computed through different algorithms, depending on the implementing library and on the used elliptic curve.

One of these algorithms is the Montgomery Ladder, which was attacked by Yarom and Berger [37] in a field of characteristic two. Another one is the w-ary non adjacent form ($wNAF$), which is used for the **secp256k1** curve in OpenSSL. It has been shown to leak a few bits per signature in [2]. The authors then used a lattice reduction [25, 26] to fully recover the private key after about 200 ECDSA signatures. This attack was later improved to reduce the number of necessary signatures to 25 [29]. Even more recently, the authors of [10] explained a method to recover the key with as little as 7 signatures for a success rate of 92%. However, the computation time exceeds three hours. Finally, another algorithm is the fixed-window implementation, which is used for the **NIST P-256** curve in OpenSSL. It has been broken in [12]. This attack targeted actually the modular inversion implementation and not the double-and-add sequence.

The focus of this paper will be the $wNAF$ implementation of scalar multiplication over the **secp256k1** curve, just like in [2]. Indeed, this implementation contains a conditional branching depending on a secret variable and leading to a cache-timing vulnerability. We concede that the **secp256k1** curve is a very particular case, and that the attack would not be effective in a more general situation (as it will be explained in Section 7 the target code was patched in recent versions of OpenSSL). However, the main takeaway of this paper should not be the performance of the attack in itself, but rather the principles of automation *via* machine learning in order to expedite the attack.

All the above mentioned attacks use the Flush+Reload leakage extraction technique in order to retrieve a sequence of operations. It can be a double-and-add sequence [2] or a right-shift / subtraction sequence [12]. In state-of-the-art papers on cache-timing attacks, the side-channel acquisition of the said sequence of operations is usually expedited [2]. In [10] the main assumption is that Flush+Reload traces contain no errors. The main contribution of those articles is to present innovative mathematical techniques to recover the private key from perfect sequences. However, lattice reduction techniques tolerate very few errors in their inputs. By experience, reproducing in real-life the attacks of those papers can be difficult, mainly because of two problems. First, computer and processor models are diverse, so a trace that would be perfectly readable in an ideal configuration might be very noisy with another setup. Second, a huge number of traces must sometimes be inspected in order to be translated into enough sequences to yield the private key. The evaluator is expected to spend a lot of time looking at signals to turn them into patterns, and as always with human processing, errors can come up.

In contrast, some approaches such as [7] consider the possibility of errors in automatic sequence extraction and make use of error correction algorithms, which are computationally expensive.

Contrary to previous research, we would like in this paper to address particularly the problem of pattern recognition in side-channel signals. In 2009, the authors of [6] recognized the importance of automated analysis for processing large volumes of cache-timing data over many executions of a given algorithm. They suggested to use Hidden Markov Models (HMMs) [30] to infer a sequence of operations and applied their idea to a modified $wNAF$ algorithm in OpenSSL. Even though we praise this pioneer work in the field of automated attacks, we think that it is less robust than our method which is based on neural networks (see Section 4 for details). Indeed, HMMs were designed to model a process, and diverting it to a pattern recognition tool is less straightforward. Furthermore, neural networks require less assumptions and customization, because they more or less act as a black-box.

Current trends in side-channel analysis consist in end-to-end automatic key extractions. One such example [18] is explained in the context of power analysis, using tools such as Markov chains Monte Carlo. We are interested in cache-timing leakage, where the leakage rate is way smaller, hence new techniques shall be put forward to analyse the leakage.

Machine learning for pattern recognition is obviously a hot topic in the research community, with applications in image [21] and sound [32] processing. Regarding applications to cache-timing attacks, the authors of [39] used Support Vector Machines, a machine learning algorithm, to classify vectors of cache access timings into a sequence of operations of a modular exponentiation. More recently, another pattern recognition technique, hierarchical clustering, has been used in [23] to identify pre-computed multipliers in cache activity traces.

1.2 Contributions and Paper Organization

The main contribution of this paper is to suggest a neural network based machine learning model to retrieve

sequences of operations from a large set of cache-timing data.

To test the accuracy of our model, we replay the side-channel attack of [2], with a stronger focus on the analysis of the generated traces. As a result, we present an end-to-end automated attack on an OpenSSL implementation of ECDSA, comprising the following steps:

- Step 1: A slightly improved cache-timing trace generation through the Flush+Flush attack, which is more robust and stealthy. (Section 3)
- Step 2: A pattern recognition algorithm to derive double-and-add sequences from traces, after an initial training phase. (Section 4)
- Step 3: An attack based on a lattice reduction to retrieve the private key from the identified signature sequences. (Section 5)

The detailed workflow is illustrated on Figure 1. Let us insist on the fact that no human intervention is needed from the moment the traces are measured and given to the trained model until the recovery of the private key. This has the double advantage of sparing evaluator's efforts and reducing the number of human-origin errors.

Another advantage of our method is that it streamlines the three above steps, which are usually distinct in traditional attacks. When a trace is generated it is immediately given to the recognition model and the lattice is then fed with the output sequence to progressively build up information on the private key. As a result, traces and sequences do not need to be stored on disk, reducing the memory footprint of the attack and improving stealthiness. Besides, the full process can be implemented as a *daemon* operating silently in the background.

The next section (Section 2) is a quick reminder about ECDSA and scalar multiplication algorithms. The three following sections (Sections 3, 4 and 5) are about the roll-out of the attack (one section for each step). Section 6 presents the setup, the workflow and the results of the attack. The last section (Section 7) is dedicated to mitigations and future applications for machine learning.

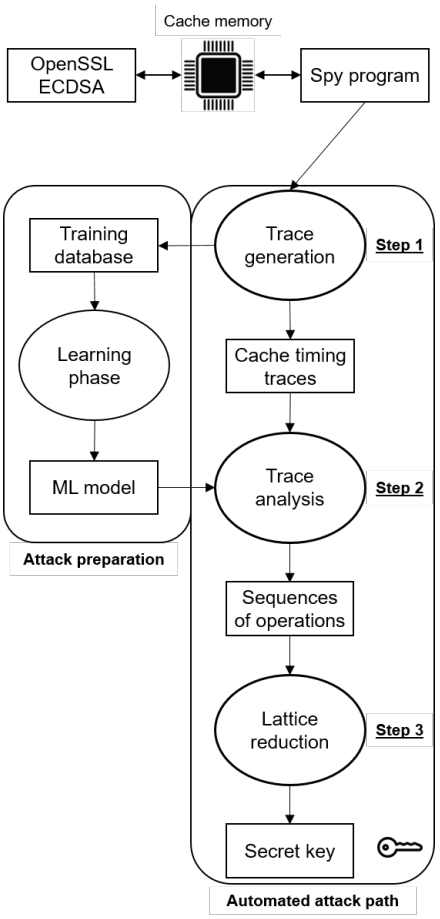


Fig. 1 Workflow of the attack. A machine learning model is manually trained ahead of the actual attack. During the attack, 1. cache-timing traces are measured during ECDSA signature, 2. the model is used to extract sensitive sequences of operations, and 3. these intermediate values are used to recover the secret key via lattice reduction. The three steps are automated.

2 Background

2.1 ECDSA Signature Algorithm

Elliptic Curve Digital Signature Algorithm (ECDSA) is the transposition to elliptic curve cryptography of the NIST-standard Digital Signature Algorithm (DSA) [34, §6]. The digital signature provides message authentication, integrity and non-repudiation (the sender cannot falsely claim that they have not signed the message).

The algorithm is part of asymmetric cryptography; the sender uses a private key to generate the signature, and the receiver uses a public key together with the message to verify the signature.

All users agree on the use of an elliptic curve E defined over a finite field $\mathbb{F}_p$ (in the case of **secp256k1**, p is a large 256-bit prime number defined in [31]). A point $G \in E$ of large prime order n is also chosen and called *generator* of the curve.

The key pair consists of a private key α [1] randomly selected in the interval $\{1, \ldots, n-1\}$; and a public key

[1] To remain consistent with the notations of [2], we chose to write α for the private key instead of d, which is the standard notation.

point $Q := \alpha \cdot G$. We use $\cdot$ to denote elliptic curve scalar multiplication.

Using the previous notations, a message m can be signed through the following steps:

1. Using an approved hash algorithm, compute $e = Hash(m)$.
2. Let h be the l leftmost bits of e, where l is the bit length of the group order n.
3. Select a cryptographically secure random integer k from the interval $\{1, \ldots, n-1\}$.
4. Compute the point $(X, y) = k \cdot G \in E$.
5. Take $r = X \mod n$; if $r = 0$ then return to step 2.
6. Compute $s = k^{-1}(h + r\alpha) \mod n$; if $s = 0$ then return to step 2.
7. The signature is the pair (r, s).

We will not give details over the verification algorithm, which is not useful for the rest of the paper. The reader is referred to [2] for verification. Let us point out the fact that knowledge of the integer k, which is commonly referred to as the *ephemeral scalar*, or *nonce*, leads to knowledge of the private key α when combined with the message and signature pair:

$$\alpha = (sk - h)r^{-1} \mod n \quad (1)$$

It is this equation which we shall exploit in our attack, by making use of secret bits of the nonce k leaking through cache memory usage.

2.2 Scalar multiplication with *wNAF*

Our attack targets step 4 of ECDSA algorithm presented in the previous subsection. In OpenSSL and for the **secp256k1** curve, this elliptic curve scalar multiplication is computed with *wNAF* algorithm (see [16] for details). In this algorithm, the nonce k is converted into a Non-Adjacent Form (NAF) $d_0 \ldots d_{l-1}$, taking the notations of Algorithm 2 in [2]. The multiplication is then computed with the double-and-add method which contains a conditional branching on the secret d_j leading to a point addition, depending on the value of d_j. It is this particular operation which is targeted in the attack. More precisely, the number of doubling operations after the last addition of the sequence reveals a few zero bits of k that can be used to find α. That is why we will focus on recovering accurately the double-and-add sequence during a *wNAF* scalar multiplication.

3 Cache-timing Traces Generation

In this section we shall explain how to use the Flush+Flush method to acquire cache-timing data during a *wNAF* scalar multiplication.

3.1 Overview on processor memories

Instructions and data are usually stored in a main memory. To reduce the average time to access this data, recent CPUs feature an additional memory which is faster and stores copies of frequently used data. This *cache memory* is more expensive in energy, thus is smaller than the main memory. Most modern processors have multiple levels of cache memories. Some levels are split and some are shared between the different cores of the processor.

3.2 Flush+Flush

In this paper, we use the Flush+Flush method of Gruss et al [14]. This attack relies on the execution time of the `clflush` instruction. `clflush` takes a memory address as parameter and evicts its content from the cache memory.

If the cache contains a copy of the content of this memory address, the `clflush` instruction will take longer than if it was not in the cache.

Indeed, for inclusive memories, `clflush` will evict the lower levels of the cache memory in case of a cache hit and thus take a longer execution time. Detailed reasons for this delay can be found in the paper of Gruss et *al.*

This delay opens an attack scenario for a spy program, measuring repetitively the time taken by `clflush` on a memory address `I` of a target program instruction during the execution of that target. If at some point, the time taken by `clflush(I)` is longer than expected, it means that the target instruction at memory address `I` has been copied in the cache during the time between the previous `clflush` and the current `clflush`. Consequently, it means that the target most probably executed the instruction at address `I` during this time interval.

In our attack, we want to reconstruct the double-and-add sequence during a signature. To do so, we select M memory addresses for instructions of the doubling function and M memory addresses for instructions of the addition function (the choice of M and of the addresses is the object of the next subsection). We denote $\left[D_1 \ldots D_M\right]$ (respectively $\left[A_1 \ldots A_M\right]$) the addresses of selected instructions of the double (respectively add) function.

We flush all the selected addresses one by one, and repeat the operation until the target is finished with the signature. For an instruction address I we denote t_i^I the measured time of the i-th iteration of `clflush(I)`. The generated trace can then be seen as a $2M \times T$ matrix

X, where T is the number of iterations of flushing all the selected addresses:

$$X = \begin{bmatrix} t_1^{D_1} & ... & t_i^{D_1} & ... & t_T^{D_1} \\ \vdots & & \vdots & & \vdots \\ t_1^{D_M} & ... & t_i^{D_M} & ... & t_T^{D_M} \\ t_1^{A_1} & ... & t_i^{A_1} & ... & t_T^{A_1} \\ \vdots & & \vdots & & \vdots \\ t_1^{A_M} & ... & t_i^{A_M} & ... & t_T^{A_M} \end{bmatrix}$$

For a given i, if the $t_i^{D_j}$ are significantly higher than usual, one can deduce that a double has been performed by the target between the $(i-1)$-th and the i-th iteration, and *idem* for the $t_i^{A_j}$. In our work, instead of helping to visually deduce double-and-add sequence, the matrix will serve as input of a pattern recognition model, as detailed in Section 4.

As explained in [2], the spy and the victim can run on different cores, because the targeted memory is a level of cache which is shared among the cores of a same physical processor. As a result, the attack does not have the same limitations as other cache-based attacks ([28]).

3.3 Choice of the number of addresses and profiling

The number of addresses to use for probing is not trivial. If the attacker uses a very small value for M (e.g. 1), the cache hits may be difficult to detect in noisy measurements. Therefore it is best to choose several instruction addresses for one target function. On the other hand, because we measure the flushing timings of $2M$ addresses successively, the time to get measurements from all the addresses of one of the two functions will be longer if M is large, and the attacker could fail to detect the use of the other function. Additionally, if M is small, the patterns for each function call will be distant from each other. This makes the measurement easier to read as all patterns are distinct from each other.

For the sake of convenience, the number of addresses for the addition function is taken equal to that in the doubling function, but there is no compelling reason for this arbitrary choice.

As for the addresses values, the debugging symbols are used to map the names of the targeted functions to the correct memory lines. Starting from the first address for each function, we select M addresses with steps of 64 bytes, corresponding to the size of a cache line for our machine. Indeed, because the `clflush` instruction does not evict a single memory cell but rather a whole cache line, probing memory addresses that will be copied in the same cache line would be ineffective.

For example, if $M = 3$ and that the doubling function starts at address `0x7000ff10` and the addition function starts at address `0x7001a000`, the probed memory addresses would be (`0x7000ff10`, `0x7000ff50`, `0x7000ff90`, `0x7001a000`, `0x7001a040`, `0x7001a080`).

The choice of the best M value is crucial for the success of the attack. For $M = 1$ and $M \geq 8$ our attack was unsuccessful. We conducted 10 attacks for $2 \leq M \leq 7$ and computed the mean and standard deviation of the number of traces required to retrieve the key. Figure 4 pictures a summary of these statistics. It is obvious that $M = 2$ or 3 yield the best results. We chose $M = 3$ since it has a lower standard deviation, making our attack more stable.

4 Pattern recognition with Machine Learning

4.1 Motivation for Machine Learning

Using Machine Learning (ML) algorithms to identify operations patterns within the cache-timing traces shows several advantages:

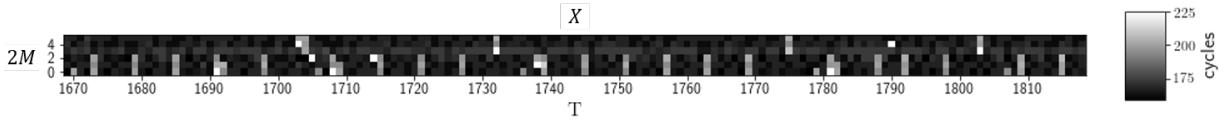
- Using a supervised approach can leverage on comprehensive cache-timing characterization obtained from the target processor.
- It is a practical and generic way to implement pattern recognition.
- Most Machine Learning models provide confidence indicators associated with their predictions, allowing to discard potential false positives and avoid detection errors.

For these reasons, we investigate the use of a Machine Learning methodology to extract the double-and-add sequence from cache measurements.

4.2 Methodology

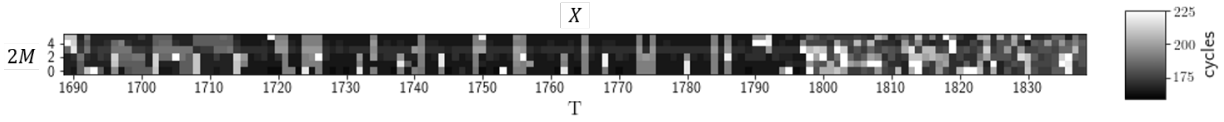
Our global approach is the following: in order to train a pattern recognition model, we generate thousands of cache traces on a processor by recording the cache activity of the doubling and addition functions during the precise execution of the scalar multiplication, in order to observe patterns denoting the utilisation of these functions.

In order to detect more easily the cache hits, the inputs of the model are the columns of the measurement matrices X where each row is a memory address and each column is a period of time during the scalar multiplication.



$$z = (\ldots, D, D, D, D, D, A, D, D, D, D, A, D, D, D, D, D, D, A, D, D, D, D, \underline{A, D, D})$$

Fig. 2 150 rightmost points of a cache-timing matrix X with $M = 3$ and the associated sequence of operations z. High values in the matrix indicate a cache hit and show the use of specific memory addresses by OpenSSL. The M upper (resp. lower) rows match the addition (resp. doubling) function addresses, and therefore correspond to the A (resp. D) operation in the sequence z. The part of z revealing secret zero bits of k is underlined.



$$z = (\ldots, A, D, D, D, D, A, D, D, D, D, A, D, D, D, D, D, D, A, D, D, D, D, \underline{A, D, D, D})$$

Fig. 3 150 rightmost points of a cache-timing matrix with higher noise than the one depicted in Figure 2 and associated sequence of operations z. This figure is difficult to interpret as the noise blurs the cache hits patterns, but our model is able to correctly decode the double-and-add operations needed to the secret recovery (underlined).

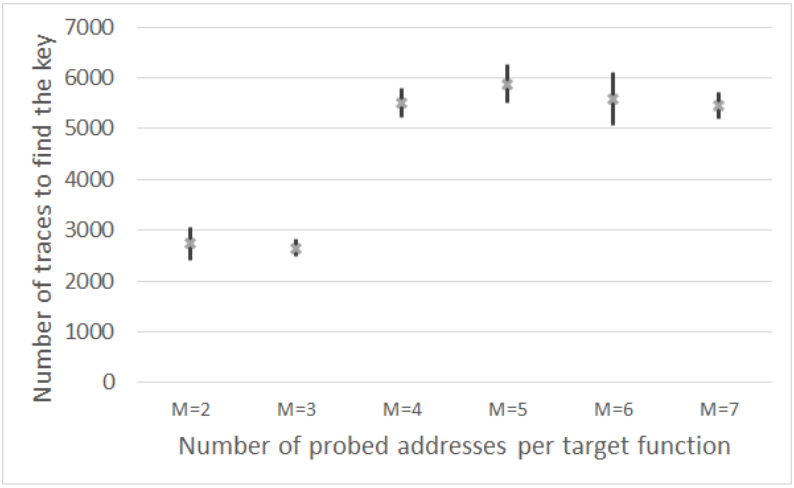


Fig. 4 Statistics over the number of traces required to find the key for different M values. The gray cross marker represents the mean of traces and vertical bars represent an interval of length twice the standard deviation. Statistical moments were computed over 10 experiences.

Additionally, we monitor and record the processor effective function calls, which yields the exact sequence of operations z matching the scalar multiplication. These sequences of operations are used as the ground truth allowing to label our different cache measurements. These labels enable a supervised approach. A sequence $z = (z_1 \ldots z_l)$ is composed of successive operations calls, namely A for addition and D for doubling ($z_i \in \{A, D\}$).

Let a single matrix X, and the associated sequence of operations z, the set of couples (X, z) constitutes our dataset for supervised learning. A sample of this database is shown on Figure 2. On this example, one can clearly identify the patterns and can easily reconstruct the sequence. The clarity of the trace depends of multiple factors like the number of processes running concurrently on the target machine. Those may add noise and make the recognition more difficult. An example of a noisy trace is shown on Figure 3. In our experiments, the model is able to correctly recognize the sequence of operations for that example, whereas it may be challenging for a human being to visualize the patterns.

The pairs (X, z) are the inputs to train the model, which can be deployed in real time to process unseen measurements. Given an input X, the model's task is to reconstruct a sequence z', ideally identical to z. This task is commonly known as **sequence prediction**.

To reconstruct this sequence, the model shall predict, for each column of X, whether an addition ('A' label), a doubling ('D' label), or none ('0' label) amongst the two operations is being executed by the target. Let

us denote by σ a sequence composed of 'A', 'D' and '0' labels. σ is called an alignment, or a path, and is an element of $\{A, D, 0\}^*$.

We finally retrieve the final sequence z' by applying a post-processing function on σ. This post-processing is composed of two steps:

- We merge all adjacent identical labels in the predicted sequence. This is consistent because we choose a small value for M, and therefore we are not expecting to find contiguous patterns.
- Finally, we remove the blank labels to keep only the doubling and addition labels and obtain the final prediction.

For example:

$$[A, 0, D, D, 0, D] \implies [A, 0, D, 0, D] \implies [A, D, D].$$

For a given sequence z, an alignment σ is called a *valid path for* z if the above post-processing function outputs z when applied to σ. The set of all valid paths of length T for z is denoted $\mathcal{A}_T(z)$.

4.3 Supervised task - Misaligned labels

During the training phase, the sequence z is used in association with each trace X, making our problematic a supervised sequence generation task.

However, these sequences miss the 'blank' labels denoting the spaces between patterns because this information cannot be obtained by monitoring function calls. From a Machine Learning perspective, the ground truth is not synchronized with the input data.

In other words, we know the order of the executed operations but we do not know at which precise point of time they occur nor the number of '0' labels between each of them. This number of 'blank' is not predictable. Target sequences are said to be misaligned with the training data. Consequently, each sequence z has a length l smaller than the length of the trace X, and it is therefore not possible to directly train a ML classifier (such as a random forest) with X and z as inputs.

From there on, different alignments are possible to match a matrix X with its associated sequence z. This problematic is common in speech recognition and different approaches exist to solve it. It is sometimes called **sequence generation** with **unsegmented labels**.

In a naive approach, we first tried to segment the matrices X in order to make them match the sequences z. To resolve the misalignment problematic, we assigned the columns of the training data with the correct operations. The realignment was done using a statistical method allowing to generate a fully labeled intermediate dataset, suitable for fully supervised learning. We then trained several classes of ML algorithms and used them to reconstruct the double-and-add sequence. However, this realignment methodology is not generic and not suitable for noisy measurements. The preliminary results from the obtained models were encouraging but we wanted to make a better use of the training data.

We therefore propose a method to process the couples (X, z) without any segmentation or realignment pre-processing.

4.4 Connectionist temporal classification

In order to remove the need for realignment, we investigate a solution based on recurrent neural networks combined with a particular objective function, inspired from natural language processing.

Recurrent neural networks (RNN) [9] are state models designed to learn the temporal characteristics of their inputs: unlike usual networks, the input data is projected into a state vector linked to itself. The values of this context vector at a time i therefore depends on the inputs at time step i but also on the values of the context vector at step $i-1$. These models are therefore able to catch the temporal dependencies between each point of a time series.

As defined in Section 3.2, our representation X of cache measurements can be viewed as stacked time series, and temporal models are particularly suited to process them. Our motivation to use recurrent neural networks is to take advantage of their modelling capacities and their genericity: they require no prior knowledge of the input data.

To train a RNN, an objective function needs to be defined. This function should evaluate the ability of the model to resolve the task it was designed for. It should also be derivable in order to allow the RNN to learn from a training dataset.

However, the usual objective functions to train neural networks must be applied for each time point of the input data, requiring ground truth for each point. As discussed in section 4.3 , our labels are unsegmented sequences and therefore information is missing to directly train a RNN.

To solve this problem, we choose to use a different objective function, the Connectionist Temporal Classification (CTC) function proposed by Graves et al. in [13].

This function has originally been designed to solve problems of sequence generation with unsegmented labels for speech recognition. It is typically used to train

RNNs with pairs of word sequences and audio records of human voices reading the sequences. In that case, the segments of the audio records where each word is pronounced is unknown and the CTC function overcomes this lack of information. The same constraint apply to the pairs (X, z). Therefore, the CTC function can be used in the context of cache-timing measurements and sequences of operations.

Given an observation X of length T (T varies between traces) and a sequence z, the CTC function computes the probability of z conditionally to X as the sum of the probabilities of all the valid paths σ of length T for z:

$$p(z|X) = \sum_{\sigma \in \mathcal{A}_T(z)} p(\sigma|X) \tag{2}$$

The probability of a path σ is given by the product of its labels probabilities for each time step:

$$p(\sigma|X) = \prod_{t=1}^{T} p_t(\sigma_t|X) \tag{3}$$

where $p_t(\sigma_t|X)$ is the probability computed by the RNN for the label $\sigma_t \in \{A, D, 0\}$ at time step t.

From there on, the CTC function can be derived and its log-likelihood can thus be minimized with the gradient back propagation algorithm to optimize the weights of the neural network. The model will then learn to predict the most probable path leading to z given its input X, overcoming the lack of information concerning the true labels position in the measurement. Using CTC removes the need of any preprocessing to align the sequences with the observations.

The final prediction of the model is given by the alignment with the highest probability predicted by the model.

$$\sigma^\star = \underset{\sigma \in \mathcal{L}}{\operatorname{argmax}}\, p(\sigma|X) \tag{4}$$

with $\mathcal{L}$ the space of possible alignments.

Given a sequence z, computing the probabilities of every alignment $\sigma \in \mathcal{L}$ can be expensive, especially when the number of labels increases. The CTC function uses a dynamic programming algorithm to make these operations faster. For more details on the CTC function mechanisms, the reader is referred to [13].

The CTC function can be used with any type of classifiers able to produce an output distribution over $\{A, D, 0\}$ for each column of X. For our attack, we combine the CTC with a simple RNN and use Stochastic Gradient Descent with Momentum [33] to train the neural network.

As explained in section 2, the number of doubling operations after the last addition in the sequence reveals secret bits of k. Therefore, we evaluate the quality of the model by computing the proportion of pairs (X,z) of a test dataset from which the model is able recover this number. In the rest of the paper, we will refer to this evaluation metric as *accuracy*.

4.5 Confidence indicators

Because this attack is an end-to-end attack, it is composed of different bricks. If each brick seems to be functional enough individually, the complete chaining can emphasize error rates which eventually causes the attack to fail.

Moreover, the lattice reduction algorithm involved in the last step of the attack is intolerant to false positives: the attack would not bear any misclassified samples by the ML model.

It is therefore important for the attacker to have the option to discard weak predictions and avoid using false positives as inputs of the equation system involved in the lattice-based key retrieval. Most ML models provide probabilities or distances associated with their predictions, and we need to consider these indicators in order to filter the classifiers outputs, based on a confidence threshold.

To ensure the efficiency of the attack, it is mandatory to calibrate models so that their predictions will respect the following constraints:

- No false positive should be predicted with high confidence.
- There must be enough true positives with high confidence to gather enough information to retrieve the key.

In other words, for the attack to be feasible, the model must not only have a good accuracy but also be able to distinguish clearly between strong and weak predictions. We therefore proceed to a phase of confidence calibration on a validation dataset.

In order to estimate the confidence indicator of a prediction of the CTC-based model, the CTC function is used to compute the probability $p(z'|X)$ of the output sequence z'. However, when the length of a sequence increases, this indicator has a tendency to vanish (i.e., converge towards 0). So, instead of computing a prediction z' with the complete measurement X, only a sub-part of the trace is considered, namely the last N columns of X. The model takes X_N the matrix composed of the N last columns of X as input and predicts z'_N with the confidence $p(z'_N|X_N)$. This prevents the confidence indicator to vanish, as the product computation of the probabilities of paths associated to z'_N considers fewer points (see Equation 3).

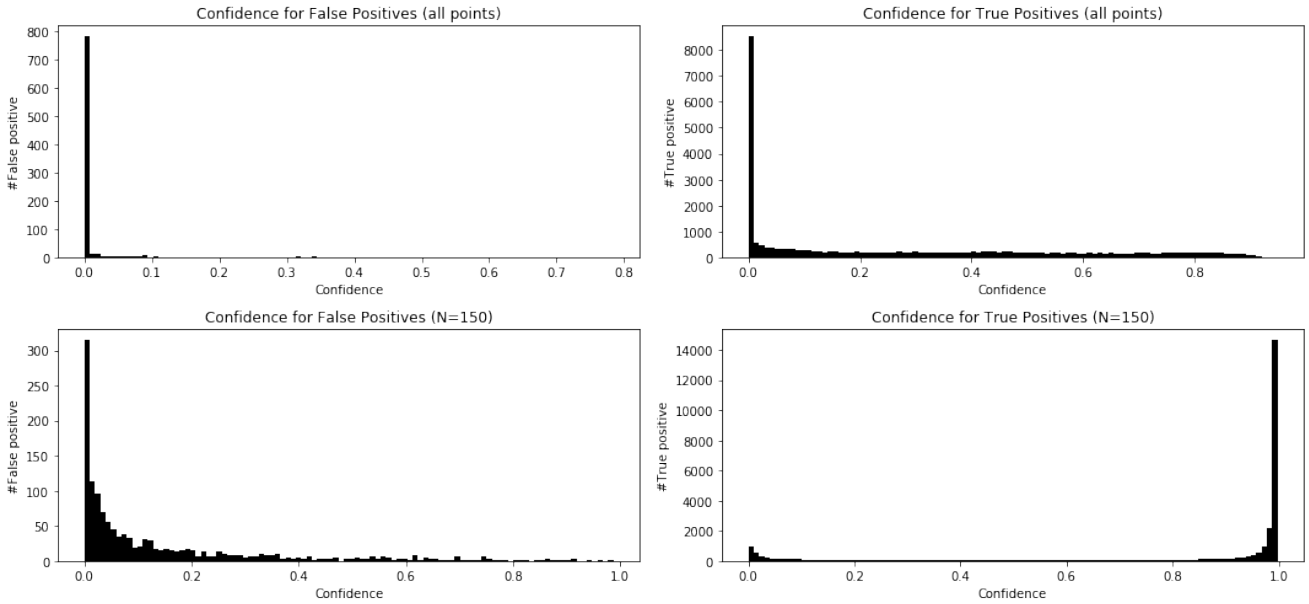


Fig. 5 Histograms of confidence values using all the points to compute the prediction (row 1, normal mode), versus using only the last 150 points (row 2, $N = 150$), on a validation dataset. In normal mode, the accuracy is 0.9684 and when $N = 150$, it is 0.9553.

Considering only the N last points of the measurements lowers slightly the global accuracy (decrease by 2 or 3%), but provides a reliable confidence indicator, as shown on Figure 5 for $N = 150$. Clearly, using the 150 last points splits the distribution of confidence indicators between false positives and true positives. After that, setting a fixed threshold p_{min} allows to reduce the number of false positives with high confidence to 0 on the validation dataset. During the attack, this allows to discard weak predictions and eliminate prediction errors.

4.6 Limitations of Machine Learning

Sequence prediction with Machine Learning is one specific step within the global attack. Moreover, this step is not at the end of the chain. Therefore, the quality of a trained model does not necessarily reflect the efficiency of the attack because the metric used to train this model can not attest the efficiency of the attack.

If we train two different types of models and find out that there is e.g. 5% difference in their accuracy on the testing dataset, we are expecting the one with the higher score to perform a better attack. This is a biased hypothesis because other factors need to be considered: most notably the reliability of the confidence indicators computed by the model.

For this reason, the metrics usually chosen for ML model evaluation are not fit to reflect the quality of the attack. Given two models with different accuracy scores, it is may well happen that the one with a significant lower accuracy performs a better attack.

5 Retrieving the Private Key with Lattice Reduction

The final step of the attack consists on a lattice reduction, using Lenstra-Lenstra-Lovàsz (LLL) [22] to compute the private key α from the extracted bits of the different ephemeral scalars.

We now have a set of d signatures and (hashed) messages (r_i, s_i, h_i) for $i = 1...d$ as well as their double-and-add sequences generated with the model presented in the previous section.

The final parts of the sequences reveal bits of the ephemeral scalar. More precisely, one property of wNAF is that if the sequence of the i-th signature is ending with an addition followed by $l_i - 1$ doubles ($...AD^{l_i-1}$, $l_i \geq 1$) then the binary form of the ephemeral scalar k_i ends with exactly $l_i - 1$ zeros ($...10^{l_i-1}$). Signatures with a number of known bits $l_i \geq \ell$ only are considered. The choice of ℓ affects the success rate of the lattice reduction and will be discussed in the next section.

As presented in [2], the key is retrieved by first constructing the following matrix from the d signatures:

$$B = \begin{pmatrix} 2^{l_1+1}n & 0 & 0 & \dots & 0 & 0 \\ 0 & 2^{l_2+1}n & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 2^{l_d+1}n & 0 & 0 \\ 2^{l_1+1}t_1 & \dots & 2^{l_d+1}t_d & \dots & 1 & 0 \\ 2^{l_1+1}u_1 & \dots & 2^{l_d+1}u_d & \dots & 0 & n \end{pmatrix}$$

where l_i is the number of known least significant bits as explained above, and n is the curve order and:

$$t_i = \frac{r_i}{(s_i.2^{l_i})} \mod n \quad ; \quad u_i = \frac{\alpha_i - \frac{h_i}{s_i}}{2^{l_i}} \mod n$$

Finally, the private key α can be extracted by applying the lattice reduction on B. The reader is referred to [2] for a more comprehensive description of lattice reduction.

6 Setup and Results

We tested the attack on an Intel Core i7-4790 processor (Haswell microarchitecture) running Ubuntu 18.04.2 LTS. The attack was also successfully tested on Skylake and Kaby Lake processors. The requirement of the attack is that the cache memory must be inclusive for the Flush+Flush technic to work, as explained in Section 3.2. The attack was conducted on version 1.1.0f of OpenSSL. The vulnerability has been corrected in latest versions using a cache constant-time multiplication method.

The number of targeted addresses for the Flush+Flush cache-timing extraction was chosen as $2M = 6$, because according to the results presented in Section 3, it is the number that allows a successful attack in a small number of traces with the least variance. We selected the addresses inside each target function `ec_GFp_simple_dbl` and `ec_GFp_simple_add`, according to the method described in Section 3.

The training phase is done before the actual attack and only requires the attacker intervention to set the gradient descent parameters and the calibration parameters (N and p_{min}). The rest of the attack (trace generation, sequence extraction and lattice reduction) is automated and requires no human intervention once the model is trained.

Because ECDSA signature generation is probabilistic, learning does not involve any change in long-term secret α nor in the data to be signed. Given the learning ability of the CTC-based model, a relatively small number of traces is enough to achieve a high test accuracy. However, a larger dataset is required for calibration. We generate 1,000 traces as a training dataset, 1,000 for testing the model, and finally, 10,000 are kept as a validation dataset for calibrating the model confidence indicators. It allowed us to choose $N = 150$ and a threshold $p_{min} = 0.95$, as these parameters allow to discriminate efficiently between true positives and false positives.

The trained model achieve high accuracy, between 0.95 and 0.97 after calibration, depending on their initialization and on the randomness of the stochastic gradient descent.

Once the model is trained, the attack is performed on the fly: ECDSA signatures are successively triggered with random nonces. The measurements are processed by the neural network directly after being generated, and the extracted bits are used to build the lattice matrix in parallel. Once the bits of d signatures have been accumulated, the reduction is computed. This workflow is detailed on Algorithm 1.

Algorithm 1: Attack workflow.

Parameters: d the number of signatures needed for reduction, N the number of points to use for predictions, p_{min} the minimum confidence, ℓ the minimum value for l_i.
Result: Private key α.

```
Initialize empty B with d_b = 0 columns
while d_b < d do
    Sign a message. Generate (r, s, h) and the
     associated measurement X
    Extract z'_N and confidence p(z'_N|X_N) with the
     ML model
    if p(z'_N|X_N) ≥ p_min then
        Get l_i from z'_N
        if l_i ≥ ℓ then
            Add a column to B using r, s, h, and l_i
Compute the lattice reduction
return α
```

Our experimental phase showed that the best value was $d = 200$ for the number of acceptable signatures and $\ell = 4$ for the minimal number of known bits for an acceptable signature. Those parameters ensured a success rate of more than 99 % in predicting the private key in a reasonable timeframe (less than 5 seconds).

Because the bits of the nonce k are randomly distributed, there is a probability $1/2^{\ell-1}$ that a signature is acceptable (the ephemeral scalar must end with at least $\ell - 1$ zeros). Thus the expected total num-

ber of traces needed to recover the private key is $1,600$. In practice, we need on average 1.5 times more traces, because around 33% of signatures are discarded due to a lack of confidence in the model predictions. **The total number of traces required is thus $\mathbf{d} \times \mathbf{2}^{\ell-\mathbf{1}} \times \mathbf{1.5} = \mathbf{2400}$ on average** (circa). For certain iterations of the experimental phase, the key retrieval was achieved with the minimum value of approximately 1600 traces. Of course this is a greater number of signatures than in [2], but we achieve a better success rate in retrieving the private key. This is mainly because we are able to process a larger number of signatures thanks to the automation of the sequence identification.

7 Conclusion and Perspectives

7.1 Mitigation

Several methods for mitigating the attack were suggested in [2]. We will not cover them, especially because at the time this paper is written, the w-NAF implementation is not used anymore, regardless of the curve. Indeed in the latest version (1.1.1c) of OpenSSL, this algorithm is used for multi-ECSM only (an operation that consists in computing $[k_1]P_1 + [k_2]P_2$ where k_1, k_2 are scalars and P_1, P_2 curve points). Multi-ECSM is only used for the verification part of ECDSA signature, therefore is non sensitive. For the generation part of ECDSA signature, a fixed-window algorithm is now used, making the double-and-add sequence independent from any secret.

We decided to realize the attack on a legacy version of OpenSSL for didactic purpose, because our objective is not to disclose a new attack but rather a new generic model for cache-timing traces processing. We also wanted to remain consistent with the original paper of [2].

7.2 Future applications for Machine Learning

In this work, we presented a neural network model for pattern recognition in cache-timing traces. We then utilized the model in a cache-timing attack targeting an elliptic curve scalar multiplication algorithm, on a specific architecture. Synergies could easily be found with state-of-the-art cryptanalysis, such as [10], which requires a small number of traces with no errors. Our model would be useful in the production of such traces in real conditions.

Besides, the modelling capabilities of neural networks are mostly limited by the complexity of their structure and the diversity of their training inputs: an improvement of our methodology would be to build a database with training examples from different architectures, enabling the creation of cross-architecture attack models.

We also believe that the same technique can be generalized to other algorithms, thus improving the quality of a number of other attacks. We were able to successfully reproduce the attack of Bernstein et *al.* [4] on an RSA sliding window modular exponentiation implemented in Libgcrypt. Our model was useful to generate the square-and-multiply sequence. We underline that Ueno et al. [36] also target that goal, but using Dynamic Time Warping (DTW) and not CTC+RNN. Synergies between both approaches might be possible.

Our model of sequence prediction with incomplete information could be applied to other side-channel attacks, not only those capturing cache-timing data. In power analysis, for example, patterns of electrical consumption must be classified. In Template Attacks [8], a lot of traces must be recorded to select *points of interest* in the data. Our model could be used to speed up the identification of relevant points to build the template.

Additionally, our model can be customized as the CTC function can be combined with different types of classifiers. In particular, it is possible to use LSTMs [17], to model more complex temporal dependencies.

Eventually, it could be beneficial to classify traces depending whether the final bits are 0, instead of recognizing a pattern of double and added and subsequently sort for nonces with trailing zeroes.

References

1. Backes, M., Dürmuth, M., Gerling, S., Pinkal, M., Sporleder, C.: Acoustic Side-Channel Attacks on Printers. In: USENIX Security symposium, pp. 307–322 (2010)
2. Benger, N., van de Pol, J., Smart, N.P., Yarom, Y.: "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In: L. Batina, M. Robshaw (eds.) Cryptographic Hardware and Embedded Systems – CHES 2014, pp. 75–92. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
3. Bernstein, D.J.: Cache-timing attacks on AES (2005). `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`
4. Bernstein, D.J., Breitner, J., Genkin, D., Bruinderink, L.G., Heninger, N., Lange, T., van Vredendaal, C., Yarom, Y.: Sliding right into disaster: Left-to-right sliding windows leak. In: International Conference on Cryptographic Hardware and Embedded Systems, pp. 555–576. Springer (2017)
5. Brouchier, J., Kean, T., Marsh, C., Naccache, D.: Temperature attacks. IEEE Security & Privacy **7**(2), 79–82 (2009)
6. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: International Conference on the Theory and

Application of Cryptology and Information Security, pp. 667–684. Springer (2009)
7. Cabrera Aldaya, A., García, C., Alvarez Tapia, L., Brumley, B.: Cache-timing attacks on rsa key generation. pp. 213–242 (2019). DOI 10.13154/tches.v2019.i4.213-242
8. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: International Workshop on Cryptographic Hardware and Embedded Systems, pp. 13–28. Springer (2002)
9. Elman, J.L.: Finding structure in time. Cognitive science **14**(2), 179–211 (1990)
10. Fan, S., Wang, W., Cheng, Q.: Attacking OpenSSL implementation of ECDSA with a few signatures. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1505–1515. ACM (2016)
11. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: Concrete results. In: International workshop on cryptographic hardware and embedded systems, pp. 251–261. Springer (2001)
12. García, C.P., Brumley, B.B.: Constant-Time Callees with Variable-Time Callers. In: E. Kirda, T. Ristenpart (eds.) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017., pp. 83–98. USENIX Association (2017). URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia`
13. Graves, A., Fernández, S., Gomez, F.J., Schmidhuber, J.: Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In: Machine Learning, Proceedings of the 23rd International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006, pp. 369–376 (2006). DOI 10.1145/1143844.1143891. URL `https://doi.org/10.1145/1143844.1143891`
14. Gruss, D., Maurice, C., Wagner, K.: Flush+Flush: A Stealthier Last-Level Cache Attack. CoRR **abs/1511.04594** (2015)
15. Guilley, S., Meynard, O., Nassar, M., Duc, G., Hoogvorst, P., Maghrebi, H., Elaabid, A., Bhasin, S., Souissi, Y., Debande, N., et al.: Vade mecum on side-channels attacks and countermeasures for the designer and the evaluator. In: 2011 6th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS), pp. 1–6. IEEE (2011)
16. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to elliptic curve cryptography. Computing Reviews **46**(1), 13 (2005)
17. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
18. Hwang, J., Yoon, J.W.: An automated end-to-end side channel analysis based on probabilistic model. Applied Sciences **10**(7), 2369 (2020). DOI 10.3390/app10072369. URL `http://dx.doi.org/10.3390/app10072369`
19. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Annual International Cryptology Conference, pp. 388–397. Springer (1999)
20. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Annual International Cryptology Conference, pp. 104–113. Springer (1996)
21. LeCun, Y., Boser, B.E., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W.E., Jackel, L.D.: Handwritten digit recognition with a back-propagation network. In: Advances in neural information processing systems, pp. 396–404 (1990)
22. Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen **261**(4), 515–534 (1982)
23. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE Symposium on Security and Privacy, pp. 605–622. IEEE (2015)
24. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks: Revealing the secrets of smart cards, vol. 31. Springer Science & Business Media (2008)
25. Nguyen, P.Q., Shparlinski, I.E.: The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. Journal of Cryptology **15**(3) (2002)
26. Nguyen, P.Q., Shparlinski, I.E.: The insecurity of the elliptic curve digital signature algorithm with partially known nonces. Designs, codes and cryptography **30**(2), 201–217 (2003)
27. OpenSSL: Cryptography and SSL/TLS Toolkit. `http://www.openssl.com`
28. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: The Case of AES. In: Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology, CT-RSA'06, pp. 1–20. Springer-Verlag, Berlin, Heidelberg (2006). DOI 10.1007/11605805_1. URL `http://dx.doi.org/10.1007/11605805_1`
29. Van de Pol, J., Smart, N.P., Yarom, Y.: Just a little bit more. In: Cryptographers' Track at the RSA Conference, pp. 3–21. Springer (2015)
30. Rabiner, L.R., Juang, B.H.: An introduction to hidden Markov models. IEEE ASSP magazine **3**(1), 4–16 (1986)
31. Research, C.: Recommended elliptic curve domain parameters. Standards for Efficient Cryptography (SEC) 2 (2000)
32. Roy, D.K., Pentland, A.P.: Learning words from sights and sounds: A computational model. Cognitive science **26**(1), 113–146 (2002)
33. Rumelhart, D.E., Hinton, G.E., Williams, R.J., et al.: Learning representations by back-propagating errors. Cognitive modeling **5**(3), 1 (1988)
34. of Standards, N.I., Technology: FIPS PUB 186-4: Digital Signature Standard (2013). DOI: http://dx.doi.org/10.6028/NIST.FIPS.186-4
35. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of DES implemented on computers with cache. In: International Workshop on Cryptographic Hardware and Embedded Systems, pp. 62–76. Springer (2003)
36. Ueno, R., Takahashi, J., Hayashi, Y.I., Homma, N.: Constructing Sliding Windows Leak from Noisy Cache Timing Information of OSS-RSA (2019). 8th International Workshop on Security Proofs for Embedded Systems (PROOFS). Atlanta, GA, USA
37. Yarom, Y., Benger, N.: Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. IACR Cryptology ePrint Archive **2014**, 140 (2014)
38. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 719–732. USENIX Association, San Diego, CA (2014). URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom`
39. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: Proceedings of the 2012 ACM conference on Computer and communications security, pp. 305–316. ACM (2012)