

# Cache-Timing Template Attacks

Billy Bob Brumley\* and Risto M. Hakala

Department of Information and Computer Science,  
Helsinki University of Technology,  
P.O.Box 5400, FI-02015 TKK, Finland,  
`{billy.brumley,risto.m.hakala}@tkk.fi`

**Abstract.** Cache-timing attacks are a serious threat to security-critical software. We show that the combination of vector quantization and hidden Markov model cryptanalysis is a powerful tool for automated analysis of cache-timing data; it can be used to recover critical algorithm state such as key material. We demonstrate its effectiveness by running an attack on the elliptic curve portion of OpenSSL (0.9.8k and under). This involves automated lattice attacks leading to key recovery within hours. We carry out the attack on live cache-timing data without simulating the side channel, showing these attacks are practical and realistic.

**Key words:** cache-timing attacks, side channel attacks, elliptic curve cryptography.

## 1 Introduction

Traditional cryptanalysis views cryptographic systems as mathematical abstractions, which can be attacked using only the input and output data of the system. As opposed to attacks on the formal description of the system, side channel attacks [1, 2] are based on information that is gained from the physical implementation of the system. Side channel leakages might reveal information about the internal state of the system and can be used in conjunction with other cryptanalytic techniques to break the system. Side channel attacks can be based on information obtained from, for example, power consumption, timings, electromagnetic radiation or even sound. Active attacks in which the attacker manipulates the operation of the system by physical means are also considered side channel attacks.

Our focus is on cache-timing attacks in which side channel information is gained by measuring cache access times; these are trace-driven attacks [3]. We place importance on automated analysis for processing large volumes of cache-timing data over many executions of a given algorithm. Hidden Markov models (HMMs) provide a framework, where the relationship between side channel observations and the internal states of the system can be naturally modeled. HMMs for side channel analysis was previously studied by Oswald [4], and models for

---

\* Supported in part by the European Commission's Seventh Framework Programme (FP7) under contract number ICT-2007-216499 (CACE).

key inference given by Karlof and Wagner [5] and Green et al. [6]. While their proposed models make use of an abstract side channel, we are concerned with concrete cache-timing data here.

The analysis additionally makes use of Vector Quantization (VQ) for classification. Cache-timing data is viewed as vectors that are matched to predefined templates, obtained by inducing the algorithm to perform in an unnatural manner. This can often easily be accomplished in software.

Abstractly, it is reasonable to consider the analysis shown here as a form of template attack [7] used in power analysis of symmetric cryptographic primitive implementations, and more recently for asymmetric primitives [8]. Chari et al. [7] formalize exactly what a template is: A precise model for the noise and expected signal for all possible values for part of the key. Their attack is then carried out iteratively to recover successive parts of the key.

It is difficult and not particularly prudent to model cache-timing attacks accordingly. In lieu of such explicit formalization, we borrow from them in name and in spirit: The attacker has some device or code in their possession that they can give input to, program, or modify in some way that forces it to perform in a certain manner, while at the same time obtaining measurements from the side channel.

Using the described analysis method, we carry out an attack on the elliptic curve portion of OpenSSL (0.9.8k). Within hours, we are able to recover the long-term private key in ECDSA by observing cache-timing data, signatures, and messages. Our attack exploits a weakness that stems from the use of a low-weight signed representation for scalars during point multiplication. The algorithm uses a precomputation table of points that are accessed during point addition steps. The lookups are reflected in the cache-timings, leaking critical algorithm state. A significant fraction of ECDSA nonce portions can be determined this way. Given enough such information, we are able to recover the private key using a lattice attack.

The paper is structured as follows. In Sect. 2, we give background on cache architectures and various published cache attacks. In Sect. 3, we review elliptic curve cryptography and the implementation in OpenSSL. Section 4 covers VQ and how to apply it effectively to cache-timing data analysis. In Sect. 5, we discuss HMMs and describe how they are used in our attack, but also how they can be used to facilitate side channel attacks in general. We present our results in Sect. 6, and countermeasures briefly in Sect. 7. We conclude in Sect. 8.

## 2 Cache Attacks

We begin with a brief review of modern CPU cache architectures. This is followed by a selective literature review of cache attacks on cryptosystem implementations.

## 2.1 Data Caches

A CPU has a limited number of working registers to store data. Modern processors are equipped with a data cache to offset the high latency of loading data from main memory into these registers. When the CPU needs to access data, it first looks in the data cache, which is faster but with smaller capacity than main memory. If it finds the data in the cache, it is loaded with minimal latency and this is known as a cache hit; otherwise, a cache miss occurs and the latency is higher as the data is fetched from successive layers of caches or even main memory. Thus access to frequently used data has lower latency. Cache layers L1, L2, and L3 are commonplace, increasing with capacity and latency. We focus on data caches here, but processors often have an instruction cache as well.

The cache replacement policy determines where data from main memory is stored in the cache. At opposite ends of the spectrum are a fully-associative cache and a direct mapped cache. Respectively, these allow data from a given memory location to be stored in any location or one location in the cache. The trade-off is between complexity and latency. A compromise is an  $N$ -way associative cache, where each location in memory can be stored in one of  $N$  different locations in the cache. The cache locations, or lines, then form a number of associative sets or congruency classes.

We give the L1 data cache details for the two example processors under consideration here.

**Intel Atom.** The L1 data cache consists of 384 lines of 64B each for a total of 24KB. It is 6-way associative, thus the lines are divided into 64 associative sets.

**Intel Pentium 4.** The L1 data cache consists of 128 lines of 64B each for a total of 8KB. It is 4-way associative, thus the lines are divided into 32 associative sets.

We focus on these because they implement Intel’s HyperThreading, a form of Simultaneous Multithreading (SMT) that allows active execution of multiple threads concurrently. In a cache-timing attack scenario, this relaxes the need to force context switches since the threads naturally compete for shared resources during execution, such as the data caches. The newly-released (Nov. 2008) Intel i7 also features HyperThreading; it has the same number of associative sets as the Intel Atom.

## 2.2 Published Attacks

Percival [9] demonstrated a cache-timing attack on OpenSSL 0.9.7c (30 Sep. 2003) where a classical sliding window was used twice for exponentiation for two 512-bit exponents in combination with the CRT to carry out a 1024-bit RSA encryption operation. Sliding window exponentiation computes  $\beta^e$  by sliding a width- $w$  window across  $e$  with placement such that the value falling in the window is odd. It then uses a precomputation table  $\beta^i$  for all odd  $1 \leq i < 2^w$ ,

accessed during multiplication steps; this lookup is reflected in the cache-timings, demonstrated on a Pentium 4 with HyperThreading. The sequence of squarings and multiplications yields significant key data: recovery of 200 bits out of each 512-bit exponent, and [9] claimed an additional 110 bits from each exponent due to fixed memory access patterns revealing information about the index to the precomputation table and thus key data. Assuming the absence of errors, [9] reasoned how this allows the RSA modulus to be factored in reasonable time. OpenSSL responded to the vulnerability in 0.9.7h (11 Oct. 2005) by modifying the exponentiation routine.

Hlaváč and Rosa [10] used a similar approach to demonstrate a lattice attack on DSA signatures with known nonce portions. They estimated that after observing 6 authentications to an OpenSSH server, which uses OpenSSL (< 0.9.7h) for DSA signatures, an attacker will have a high success probability when running a lattice attack to recover the private key. They state that the side channel was emulated for the experiments.

The numerous published attacks against secret key implementations are noteworthy. Among others, these include attacks on AES by Bernstein [11] and Osvik et al. [12]. Both papers present key recovery attacks on various implementations.

### 3 Elliptic Curve Cryptography

To demonstrate the effectiveness of the analysis method, we will look at one particular implementation of ECC. We stress that the scope of the analysis is much larger; this is merely one example of how it can be used.

Given a point  $P$  on an elliptic curve and scalar  $k$ , scalar multiplication computes  $kP$ . This operation is the performance benchmark for an elliptic curve cryptosystem. It is normally carried out using a double-and-add approach, of which there are many varieties. We outline a common one later in this section.

Our attack is demonstrated on an implementation of scalar multiplication used by ECDSA signature generation. A signature  $(r, s)$  on a message  $m$  is produced using

$$r = x(kG) \bmod n \tag{1}$$

$$s = k^{-1}(h(m) + rd) \bmod n \tag{2}$$

with point  $G$  of order  $n$ , nonce  $k$  chosen uniformly from  $[1, n)$ ,  $x(P)$  the projection of  $P$  to its  $x$ -coordinate,  $h$  a collision-resistant hash function, and  $d$  the long-term private key corresponding to the public key  $D = dG$ .

#### 3.1 ECC in OpenSSL

OpenSSL treats two cases of elliptic curves over binary and prime fields separately and implements scalar multiplication in two ways accordingly. We consider only the latter case, where a general multi-exponentiation algorithm is used [13,

14]. The algorithm works left-to-right and uses interleaving, where one accumulator is used for the result and point doublings are shared; low-weight signed representations are used for individual scalars.

When only one scalar is passed, as in (1) or when creating a signature using the OpenSSL command line tool, it reduces to a rather textbook version of scalar multiplication, in this case using the modified Non-Adjacent Form  $\text{mNAF}_w$  (see, for example, [15]). This is reflected in the pseudocode below. OpenSSL has the ability to store the precomputed points in memory, so with a fixed  $P$  such as a generator they need not necessarily be recomputed for each invocation.

The representation  $\text{mNAF}_w$  is very similar to the regular windowed  $\text{NAF}_w$ . Each non-zero coefficient is followed by at least  $w - 1$  zero coefficients, except for the most significant digit which is allowed to violate this condition in some cases to reduce the length of the representation by one while still retaining the same weight. Considering the MSBs of  $\text{NAF}_w$ , one applies  $10^{w-1}\delta \mapsto 010^{w-2}\epsilon$  where  $\delta < 0$  and  $\epsilon = 2^{w-1} + \delta$  when possible to obtain  $\text{mNAF}_w$ .

**Algorithm:** Scalar Multiplication

**Input:**  $k \in \mathbb{Z}$ ,  $P \in E(\mathbb{F}_p)$ , width  $w$

**Output:**  $kP$

$(k_{\ell-1} \dots k_0) \leftarrow \text{mNAF}_w(k)$

Precompute  $iP$  for all odd  $0 < i < 2^{w-1}$

$Q \leftarrow k_{\ell-1}P$

**for**  $i \leftarrow \ell - 2$  **to**  $0$  **do**

$Q \leftarrow 2Q$

**if**  $k_i \neq 0$  **then**  $Q \leftarrow Q + k_iP$

**end**

**return**  $Q$

**Algorithm:** Modified  $\text{NAF}_w$

**Input:** window width  $w$ ,  $k \in \mathbb{Z}$

**Output:**  $\text{mNAF}_w(k)$

$i \leftarrow 0$

**while**  $k \geq 1$  **do**

**if**  $k$  *is odd* **then**  $k_i \leftarrow k \bmod 2^w$ ,

$k \leftarrow k - k_i$

**else**  $k_i \leftarrow 0$

$k \leftarrow k/2$ ,  $i \leftarrow i + 1$

**end**

**if**  $k_{i-1} = 1$  **and**  $k_{i-1-w} < 0$  **then**

$k_{i-1-w} \leftarrow k_{i-1-w} + 2^{w-1}$

$k_{i-1} \leftarrow 0$ ,  $k_{i-2} \leftarrow 1$ ,  $i \leftarrow i - 1$

**end**

**return**  $(k_{i-1}, \dots, k_0)$

### 3.2 Cache Attack Vulnerability

Following the description of the  $\text{mNAF}_w$  representation, knowledge of the curve operation sequence corresponds directly to the algorithm state, yielding quite a lot of key data. Point additions take place when a coefficient  $k_i \neq 0$  and these are necessarily followed by  $w$  point doublings due to the scalar representation. From the side channel perspective, consecutive doublings allow inference of zero coefficients, and more than  $w$  point doublings reveals non-trivial zero coefficients.

Without any countermeasures, the above scalar multiplication routine is vulnerable to cache-timing attacks. The points in the precomputation phase are stored in memory; when a point addition takes place, the point to be added is loaded into the cache. An attacker can detect this by concurrently running a spy process [9] that does nothing more than continually load its own data into the cache and measure the time require to read from all cache lines in a cache set,

iterating the process for all cache sets. Fast cache access times indicate cache hits and the scalar multiplication routine has not aggressively accessed those cache locations since the last iteration, which would evict the spy process data from those cache locations, cause a cache miss, and thus slower cache access times for the spy process.

In Fig. 1, we illustrate typical cache timing data obtained from a spy process running on a Pentium 4 (Top) and Atom (Bottom) with OpenSSL 0.9.8k performing an ECDSA signature operation concurrently. The top eight rows of each graph are metadata; the lower half represents the VQ label and the upper half the algorithm state. We show how we obtained the metadata in Sect. 4 and Sect. 5, respectively. The remaining cells are the actual cache-timing data. Each cell in these figures indicates a cache set access time. Technically, time moves within each individual cell, then from bottom to top through all cache sets, then from left to right repeating the measurements. To visualize the data, it is beneficial to consider the data as vectors with length equal to the number of cache sets, and time simply moves left to right.

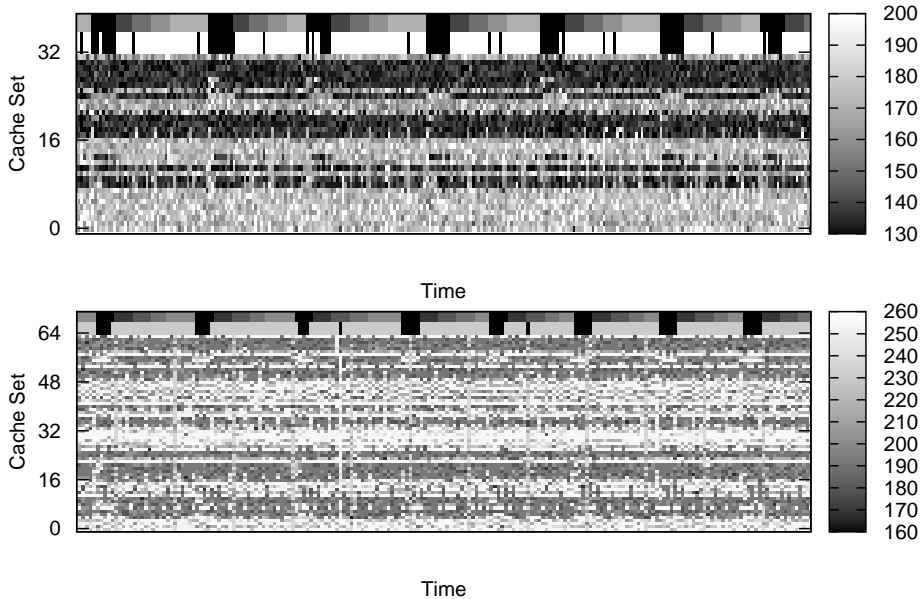
To manually analyze such traces and determine what operations are being performed we look for (dis)similarities between neighboring vectors. These graphs show seven (Top) and eight (Bottom) point additions, with repeated point doublings occurring between each addition. As an attacker, we hope to find correlation between these point additions and the cache access times—which we easily find here. Additions in the top graph are visible at rows 13 and 24, among others; the bottom graph, rows 6, 7, 55, 56. The reader is encouraged to use the vector quantization label to help locate the point additions (black label).

## 4 Vector Quantization

Automated analysis of cache-timing data like that shown in Fig. 1 is not a trivial task. When given just one trace, for simplistic algorithms it is sometimes possible to interpret the data manually. For many traces or complex algorithms this is not feasible. We aim to automate the process; the analysis begins with VQ.

A *vector quantizer* is a map  $V : \mathcal{R}^n \rightarrow \mathcal{C}$  with  $\mathcal{C} \subset \mathcal{R}^n$  where the set  $\mathcal{C} = \{c_1, \dots, c_\alpha\}$  is called the *codebook*. A typical definition is  $V : v \mapsto \arg \min_{c \in \mathcal{C}} D(v, c)$  where  $D$  measures the  $n$ -dimensional Euclidean distance between  $v$  and  $c$ . One also associates a *labelling*  $L : \mathcal{C} \rightarrow \mathcal{L}$  with the codebook vectors; this can be as trivial as  $\mathcal{L} = \{1, \dots, \alpha\}$  depending on the application.

Here, we are particularly interested in VQ classification; input vectors are mapped to the closest vector in the codebook, then applied the corresponding label for that codebook entry. In this manner, input vectors with the same labelling share some user-defined quality and are grouped accordingly. The classification quality depends on how well the codebook vectors approximate input data for their label. We elaborate on building the codebook  $\mathcal{C}$  below.



**Fig. 1.** Cache-timing data from a spy process running concurrently with an OpenSSL 0.9.8k ECDSA signature operation; 160-bit curve, mNAF<sub>4</sub>. Top: Pentium 4 timing data, seven point additions. Bottom: Atom timing data, eight point additions. Repeated point doublings occur between the additions. The top eight rows are metadata; the bottom half the VQ label (Sect. 4) and top half the HMM state (Sect. 5). All other cells are the raw timing data, viewed as column vectors from left to right with time.

#### 4.1 Learning Vector Quantization

To learn the codebook vectors, we employ LVQ [16]. This process begins with a set  $\mathcal{T} = \{(t_1, l_1), \dots, (t_j, l_j)\}$  of training vectors and predetermined corresponding labels, as well as an approximation to  $\mathcal{C}$ . This is commonly derived by taking the  $k$  centroids resulting from  $k$ -means clustering [17] on all  $t_i$  sharing the same label. LVQ in its simplest form then proceeds as follows. For each  $t_i, l_i \in \mathcal{T}$  if  $L(V(t_i)) = l_i$  the classification is correct and the matching codebook vector is pulled closer to  $t_i$ ; otherwise, incorrect and it is pushed away. This process is iterated until an acceptable error rate is achieved.

#### 4.2 Cache-Timing Data Templates

We apply the above techniques to analyze cache-timing data. Taking the working example in Fig. 1, for the Pentium 4 we have  $n = 32$  and Atom  $n = 64$  the dimension of the cache-timing data vectors; this is the number of cache sets. For simplicity we define  $\mathcal{L} = \{D, A, E\}$  to label vectors belonging to respective operations double, addition, or beginning/end.

Next, we build the training data  $\mathcal{T}$ . This is somewhat simplified for an attacker as they can create their own private key and generate signatures to produce training data. Nevertheless, extracting individual vectors by hand proves

quite tedious and error-prone. Also, if the spy process executes multiple times, there is no guarantee where the memory buffer for the timing data will be allocated. From execution to execution, the vectors will likely look quite different.

Inspired by template attacks [7], we instead modify the software in such a way that it performs only a single task we would like to distinguish. For the scalar multiplication routine shown in Sect. 3, we force the algorithm to perform only point doubling (addition) and collect templates to be used as training vectors by running the modified algorithm concurrently with the cache spy process, obtaining the needed cache-timing data. This provides large amounts of training vectors and corresponding labels to define  $\mathcal{T}$  with minimal effort.

One might be tempted to use these vectors in their entirety for  $\mathcal{C}$ . There are a few disadvantages in doing so:

- This would cause VQ to run slower because  $\#\mathcal{C}$  would be sizable and contain many vectors such that  $L(c_i) = L(c_j)$  where  $D(c_i, c_j)$  is needlessly small; codebook redundancy in a sense. In practice we may need to analyze copious volumes of trace data.
- We cannot assume the obtained cache-timing data templates are completely error-free; we strive to curtail the effect of such erroneous vectors.

To circumvent these issues, we partition  $\mathcal{T} = \bigcup_{l \in \mathcal{L}} \{(t_i, l_i) : l_i = l\}$  as subsets of all training vectors corresponding to a single label and subsequently perform  $k$ -means clustering on the vectors in each subset. The resulting centroids are then added to  $\mathcal{C}$ . Finally, with  $\mathcal{C}$  and  $\mathcal{T}$  realized we employ LVQ to refine  $\mathcal{C}$ . This allows experimentation with different values for  $k$  in  $k$ -means to arrive at a suitably compact  $\mathcal{C}$  with small vector classification error rate.

While we expect quality results from VQ classification, errors will nevertheless occur. Furthermore, we are still left with the task of inferring algorithm state. To solve this problem, we turn to hidden Markov models.

## 5 Hidden Markov Models

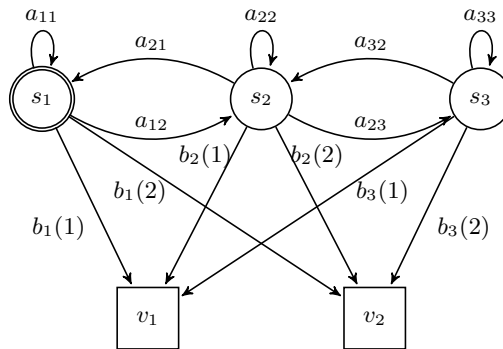
HMMs (see, e.g., [18]) are a common method for modeling discrete-time stochastic processes. An HMM is a statistical model in which the system being modeled is assumed to behave like a probabilistic finite state machine with directly unobservable state. The only way of gaining information about the process is through the observations that are emitted from each state.

HMMs have been successfully used in many real life applications; for example, many modern speech recognition methods are based on HMMs [18]. Their usability is based on the ability to model physical systems and gain information about the hidden causes of emitted observations. Thus, it is not very surprising that HMMs can be employed in side channel cryptanalysis as well: the target system can be viewed as the hidden part of the HMM and the emitted observations as information leaked through the side channel. In the following sections, we give a formal definition of an HMM, discuss the three basic problems for HMMs and describe how HMMs are used in our attack. The methodology should give an idea of how to use HMMs in side channel attacks in general.



### 5.1 Elements of an HMM

An HMM models a discrete-time stochastic process with a finite number of possible states. The state of the process is assumed to be directly unobservable, but information about it can be gained from symbols that are emitted from each state. The process changes its state based on a set of transition probabilities that indicate the probability of moving from one state to another. An observable symbol is emitted from each process state according to a set of emission probabilities. An example of an HMM is illustrated in Fig. 2. This HMM models a system with three internal states, which are denoted by circles in the figure. Denoted by squares are the two symbols, which can be emitted from the internal states. The state transition probabilities and the emission probabilities are denoted by labeled arrows. For example, the probability of moving from state  $s_2$  to  $s_3$  is  $a_{23}$ ; the probability of emitting symbol  $v_2$  from state  $s_3$  is  $b_3(2)$ . In this HMM, the process always starts from  $s_1$ . Generally, however, there may be several possible first states. The initial state distribution defines the probability distribution for the first state over the states of the HMM.



**Fig. 2.** An example of an HMM.

Formally, an HMM is defined by the set of internal states, the set of observation symbols, the transition probabilities between internal states, the emission probabilities for each observable, and the initial state distribution. We denote the set of internal states by  $S = \{s_1, s_2, \dots, s_N\}$  and the state at time  $t$  by  $w_t$ . Correspondingly, the set of observables is denoted by  $V = \{v_1, v_2, \dots, v_M\}$  and the observation emitted at time  $t$  by  $o_t$ . The set of transition probabilities is denoted by  $A = \{a_{ij}\}$ , where

$$a_{ij} = \Pr(w_{t+1} = s_j | w_t = s_i), \quad 1 \leq i, j \leq N,$$

such that  $\sum_{j=1}^N a_{ij} = 1$  for all  $1 \leq i \leq N$ . Whenever  $a_{ij} > 0$ , there is a direct transition from state  $s_i$  to state  $s_j$ ; otherwise, it is not possible to reach  $s_j$  from  $s_i$  in a single step. An arrow in Fig. 3 denotes a positive transition probability. Thus,

$s_3$  cannot be reached from  $s_1$  in a single step. The set of emission probabilities is denoted by  $B = \{b_j(k)\}$ , where

$$b_j(k) = \Pr(o_t = v_k | w_t = s_j), \quad 1 \leq j \leq N, \quad 1 \leq k \leq M.$$

The initial state distribution indicates the probability distribution for the first state  $w_1$ . It is denoted by  $\pi = \{\pi_i\}$ , where

$$\pi_i = \Pr(w_1 = s_i), \quad 1 \leq i \leq N.$$

The first state of the HMM in Fig. 2 is always  $s_1$ , so the initial state distribution for this HMM is defined as  $\pi_1 = 1$  and  $\pi_i = 0$  for all  $i \neq 1$ . The three probability measures  $A$ ,  $B$  and  $\pi$  are called the model parameters. For convenience, we will simply write  $\lambda = (A, B, \pi)$  to indicate the complete parameter set of an HMM.

## 5.2 The Three Basic Problems for HMMs

The usefulness of HMMs is based on the ability to model relationships between internal states and observations. Related to this are the following three problems, which are commonly called the three basic problems for HMMs in literature (e.g., [18]):

**Problem 1** Given an observation sequence  $O = o_1 o_2 \cdots o_T$  and a model  $\lambda = (A, B, \pi)$ , how do we efficiently compute  $\Pr(O|\lambda)$ , the probability of the observation sequence given the model?

**Problem 2** Given an observation sequence  $O = o_1 o_2 \cdots o_T$  and a model  $\lambda$ , what is the most likely state sequence  $W = w_1 w_2 \cdots w_T$  that produced the observations?

**Problem 3** Given an observation sequence  $O = o_1 o_2 \cdots o_T$  and a model  $\lambda$ , how do we adjust the model parameters  $\lambda = (A, B, \pi)$  to maximize  $\Pr(O|\lambda)$ ?

We briefly review the methods used to solve these problems; the reader can refer to [18] for a detailed overview. Problem 1 is sometimes called the evaluation problem since it is concerned with finding the probability of a given sequence  $O$ . This problem is solved by the forward-backward algorithm (see, e.g., [18]), which is able to efficiently compute the probability  $\Pr(O|\lambda)$ . Problem 2 poses a problem that is very relevant to our work. It is the problem of finding the most likely explanation for the given observation sequence. The aim is to infer the most likely state sequence  $W$  that has produced the given observation sequence  $O$ . There are other possible optimality criteria [18], but we are interested in finding  $W$  that maximizes  $\Pr(W|O, \lambda)$ . The problem is known as the decoding problem and it is efficiently solved by the Viterbi algorithm [19]. Another relevant question is posed by Problem 3, which asks how to adjust the model parameters  $\lambda = (A, B, \pi)$  to maximize the probability of the observation sequence  $O$ . Although there is no known analytical method to adjust  $\lambda$  such that  $\Pr(O|\lambda)$  is maximized, the Baum-Welch algorithm [20] provides one method to locally maximize  $\Pr(O|\lambda)$ . The process is often called training the HMM and it typically involves collecting a set of observation sequences from a real physical phenomenon, which are used in training. This problem is known as the learning problem.

### 5.3 Use of HMMs in Side-Channel Attacks

HMMs are also useful tools for side channel analysis [4]. Karlof and Wagner [5] and Green et al. [6] use HMMs for modeling side channel attacks. Their research is concerned with slightly different problems than ours. We outline the differences below.

- They only consider Problem 1 and simulate the side channel. As a result, Problem 3 is not relevant to their work since the artificial side channel actually defines the model that produces the observations. Thus their model parameters are known a priori. This is not the case for our work; Problem 3 is essential.
- They assume one state transition per key digit, in which case the key can be inferred directly from the operation of the algorithm. In our case, the operation sequence does not reveal the entire key, but a significant fraction of the key nevertheless. We use an HMM in which the states correspond only to possible algorithm states.
- They are additionally interested in derivation of the (secret) scalar  $k$  in scalar multiplication when the same scalar is used during several runs using a process called belief propagation. This is not helpful in our case, since (EC)DSA uses nonces.

A practical drawback of the HMM presented by Karlof and Wagner was that a single observable needs to correspond to a single key digit (and internal state). Green et al. presented a model, where this is not required: multiple observables can be emitted from each state. This is a more realistic model as one system state may emit variable length data through the side channel. Our model allows this also, but it is based on a different approach.

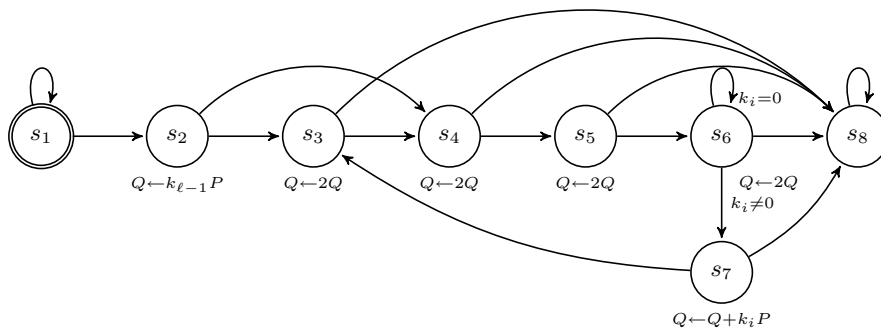
In the following sections, we describe the HMM used for modeling the OpenSSL scalar multiplication algorithm. We use this model in conjunction with VQ to describe the relationship between the states of the algorithm and the side channel observations. We also describe how to perform side channel data analysis using VQ and the HMM. The aim is to find the most likely state sequence for each trace that is obtained from the side channel. The analysis process can be divided into two steps:

1. The VQ codebook is created and the HMM parameters are adjusted according to obtained training sequences.
2. The actual data analysis is performed. When a sequence of observations is obtained from the side channel, we infer the most likely (hidden) state sequence that has emitted these observations using VQ and the HMM.

Since these states correspond to the internal states of the system, we are able to determine a good estimate of what operations have been done. This information allows us to recover the key.

The following sections give a framework for performing side channel attacks on any system. The main requirements are that we know the specification of the system and have access to do experiments with it or are able to accurately model it.

**The HMM for Scalar Multiplication** We construct an HMM where the hidden part models the operation of the algorithm—in this case, scalar multiplication using the modified  $\text{NAF}_w$  representation, which leaks information about the algorithm state through the side channel. An illustration of this part (without the transition probabilities) is presented in Fig. 3. The state set is defined as  $S = \{s_1, \dots, s_8\}$ . Each label denotes the operation that is performed in the corresponding state. In addition, there are separate states to denote the system state preceding and following the execution of the algorithm. These states are denoted by  $s_1$  and  $s_8$ , respectively. OpenSSL uses  $\text{mNAF}_4$  for scalars in the case of the 160-bit curve order we are experimenting with, so each point addition is followed by at least 4 point doublings, except in the beginning or end of the process. The states  $s_3, \dots, s_6$  represent these doublings. The most significant digit is handled by the first addition state  $s_2$ .



**Fig. 3.** An HMM transition model for modified  $\text{NAF}_4$  scalar multiplication.

As can be seen from Fig. 1, the execution of one point doubling or point addition spans several column vectors in the trace. Hence, we should let the internal states emit multiple observations instead of just one. Green et al. [6] solved this problem by introducing an additional variable that counts the cumulative number of emitted observables. This has the drawback of considerably expanding the state space. To avoid this, we solve the problem by introducing substates in each HMM state. One main state consists of a sequence of substates, which are just ordinary HMM states that always emit one observation. Thus, all previously introduced techniques can be used for our HMM.

The set of observables for this HMM is  $V = \{D, A, E\}$ , which is the same set used for labeling cache-timing data vectors in Sect. 4. We assume that the additions emit mainly  $A$ s and the doublings mainly  $D$ s. The  $s_1$  and  $s_8$  states are assumed to emit mainly  $E$ s. These symbols are connected with side channel observations using VQ as described in Sect. 4. Each vector observation is labeled according to which state— $A$ ,  $D$  or  $E$ —they correspond to. When a new side channel observation is obtained, it can be classified as  $A$ ,  $D$  or  $E$  by taking the

label of the closest codebook vector. An example of this is shown in Fig. 1, where the rows directly above the observations represent the quantized values. Symbols  $A$  and  $D$  are indicated using darker and lighter shades, respectively.

**Training of the HMM** Training starts by setting the initial model parameters. These parameters can be rough estimates, since they will be improved during training. To train the model, we obtain a set of sequences in the HMM observation domain. These sequences can be created from the side channel observations as we know how the algorithm operates. The obtained sequences are used for model parameter re-estimation, which is performed using the Baum-Welch algorithm [20]. Next, we create the codebook for VQ as shown in Sect. 4.

**Inference of the State Sequence** Given a set of side channel observation sequences from the real target system, we can infer the most likely hidden state sequence for each of them. The first step is to perform VQ, this is, to tag the observations with the label of the closest codebook vector. Thus, we get a set of sequences in the HMM observation domain. By applying the Viterbi algorithm [19], we finally obtain the most likely state sequence for each observation sequence. These state sequences are actually sequences of substates; the actual operation sequence can be recovered based on the transitions that are taken in each state sequence. An example of this is shown in Fig. 1, where the upper rows represent the main states of the algorithm. Additions are indicated using black; doublings are indicated using lighter shades. For example, the first addition on the top trace in Fig. 1 is followed by five doublings.

The state sequences obtained in this step can be used in conjunction with some other method to mount a key recovery attack. In the simplest case, the state sequence reveals the secret key directly and no other methods are needed. However, with  $\text{mNAF}_4$  this is not the case; we discuss a few practical applications in the next section, as well as give our empirical results.

## 6 Results

Depending on the attack scenario and the number of traces available, there are at least two interesting ways to apply the analysis to the case of  $\text{mNAF}_4$  and OpenSSL. The first assumes access to only a single or similarly small number of traces, while the second assumes access to a signature oracle and corresponding side channel information.

**Solving Discrete Logs** We consider special versions of the baby-step giant-step algorithm for searching restricted exponent spaces; see [21, Sect. 3.6] for a good overview.

The length- $\ell$   $\text{mNAF}_w$  representation has maximum weight  $\ell/w$  and average weight  $\ell/(w+1)$ ; we denote this weight as  $t$ . We assume that the analysis provides us with the position of non-zero coefficients, but not their explicit value or sign;

thus each coefficient gives  $w - 1$  bits of uncertainty. One can then construct a baby-step giant-step algorithm to solve the ECDLP in this restricted key space. The time and space complexity is  $O(2^{(w-1)t/2})$ ; note that this does not directly depend on  $\ell$  (or further, the group order  $n$ ). For the curve under consideration, this gives a worst case of  $O(2^{60})$  and on average  $O(2^{48})$ , whereas the complexity without any such side channel information is  $O(2^{80})$ .

**Lattice Attacks** Despite this reduced complexity, an attacker cannot trivially carry out the attack outlined above on a normal desktop PC. Known results on attacking signature schemes with partial knowledge of nonces include [22, 23]; the approach is a lattice attack. Formally, the attacker obtains tuples  $(r_i, s_i, m_i, \hat{k}_i)$  consisting of a signature (2), message, and partial knowledge of the nonce  $k$  obtained through the timing data analysis. For our experiments, not all such tuples are useful in the lattice attack. Using the formalization of [22], we assume  $\hat{k}_i$  tells us

$$k_i = z'_i + 2^{\alpha_i} z_i + 2^{\beta_i} z''_i$$

with  $z_i$  the only unknown on the right. Our empirical timing data analysis results show that the majority of errors occur when too many or few doubles are placed between an addition; a synchronization error in a sense. So the farther we move towards the MSB, the more likely it is that we have erroneous indexing  $\alpha_i, \beta_i$  and the lattice attack will likely fail.

To mitigate this issue, we instead focus only on the LSBs. We disregard the upper term by setting  $z''_i = 0$  and consider only tuples where  $\hat{k}_i$  indicates that  $z'_i = 0$  and  $\alpha_i \geq 6$ ; that is, the LSBs of  $k_i$  are 000000. For  $k$  chosen uniformly, this should happen with the reasonable probability of  $2^{-6}$ . Our empirical results are in line with those of [22]: For a 160-bit group order, 41 such tuples is usually enough for the lattice attack to succeed in this case.

Lattice attacks have no recourse to compensate for errors. If our analysis determines  $z'_i = 0$  but indeed  $z'_i \neq 0$  for some  $i$ , that instance of the lattice attack will fail. We thus adopt the naïve strategy of taking random samples of size 41 from the set of tuples until the attack succeeds; an attacker can always check the correctness of a guess by calculating the corresponding public key and comparing it to the actual public key. This strategy is only feasible if the ratio of error-free tuples to erroneous tuples is high.

Finally, we present the automated lattice attack results; 8K signatures with messages and traces were obtained in both cases.

**Pentium 4 results.** The analysis yielded 122 tuples indicating  $z'_i = 0$ . The long-term private key  $d$  (2) was recovered after 1007 lattice attack iterations (107 correct, 15 incorrect). The analysis ran in less than an hour on a Core 2 Duo.

**Atom results.** The analysis yielded 147 tuples indicating  $z'_i = 0$ . We recovered  $d$  after a total of 37196 lattice attack iterations (115 correct, 32 incorrect). Our analysis is less accurate in this case, but still accurate enough to recover the key in only a few hours on a Core 2 Duo.

**Summary** We omit strategies for finding correlation between the traces and specific key digits. This can be tremendously helpful in further reducing the search space when trying to solve the ECDLP. As such, given only one or a few traces, this analysis method should be used as a tool in conjunction with other heuristics to trim the search space. The lattice attack given here is proof-of-concept. The results suggest that significantly fewer signatures are needed. In practice one can perform a much more intelligent lattice attack, perhaps even considering lattice attacks that account for key digit reuse [24].

## 7 Countermeasures

An implementation should not rely on any one countermeasure for side channel security, but rather a combination. We briefly discuss countermeasures, with an emphasis on preventing the specific weakness we exploited in OpenSSL.

**Scalar Blinding.** One often-proposed strategy [1, 25–27] is to blind the scalar  $k$  from the point multiplication routine using randomization. One form is  $(k + mn + \tilde{m})P - \tilde{m}P$  with  $m, \tilde{m}$  small (e.g. 32-bit) and random. The calculation is then carried out using multi-exponentiation with interleaving. With such a strategy, it suffices that  $\tilde{m}$  is low weight—not necessarily short.

**Randomized Algorithms.** Use random addition-subtraction chains instead of highly regular double-and-add routines. Oswald [28] gave an example and a subsequent attack [4]. Published algorithms tend to be geared towards hardware or resource restricted devices; see [29] for a good review. In a software package like OpenSSL that normally runs on systems with abundant memory, one does not have to rely on simple randomized recoding and can build more flexible addition-subtraction chains.

**Shared Context.** In OpenSSL’s ECC implementation, the results and illustration in Fig. 1 suggest what is most visible in the traces is not the lookup from the precomputation table, but the dynamic memory for variables in the point addition and doubling functions. OpenSSL is equipped with a shared context [30, pp. 106–107] responsible for allocating memory for curve and finite field arithmetic. Memory from this context should be served up randomly to prevent a clear fixed memory access pattern.

**Operation Balancing.** In addition to the above shared context, coordinate systems and point addition formulae that are balanced in the number and order of operations are also useful; [31] gives an example.

The above countermeasures restrict to the software engineering view. Clearly operating system-level and hardware-level countermeasures are additionally possible. We leave general countermeasures to this type of attack as an open question.

## 8 Conclusion

We summarize our contributions as follows:

- We introduced a method for automated cache-timing data analysis, facilitating discovery of critical algorithm state. This is the first work we are aware of that provides this at a framework level, e.g. not specific to one cryptosystem. Consequentially, it bridges the gap between cache attack vulnerabilities [9] and attacks requiring partial key material [22, 23].
- We showed how to apply HMM cryptanalysis to cache-timing data; to the best of our knowledge, its first published application to real traces. This builds on existing work in the area of abstract side channel analysis using HMMs [4–6], yet departs by tackling practical issues inherent to concrete side channels.
- We demonstrated the method is indeed practical by carrying out an attack on the elliptic curve portion of OpenSSL using live cache-timing data. The attack resulted in complete key recovery, with the analysis running in a matter of hours on a normal desktop PC.

The method works by:

1. Creating cache-timing data vector templates that reflect the algorithm’s cache access behavior.
2. Using VQ to match incoming cache-timing data to these existing templates.
3. Using the output as observation input to an HMM that accurately models the control flow of the algorithm.

The setup phase, including acquiring the templates used to build the VQ codebook vectors and learning the HMM parameters, is the only part by definition requiring any manual work, and the majority of that can in fact be automated by simple modifications to the software under attack. This attack scenario is described for hardware power analysis in [7], but is perhaps even a greater practical threat in this case due to the inherent malleability of software. After the setup phase, cache-timing data analysis is fully automated and requires negligible time.

The analysis given here is not strictly meant for attacking implementations, but for defending them as well. We encourage software developers to analyze their implementations using these methods to discover memory access patterns and apply appropriate countermeasures.

## Future Work

One might think to forego the VQ step and use the cache-timing data directly as the sole input to the HMM. In our experience, this only complicates the model and hampers quality results.

The example we gave was tailored to data caches, in particular the L1 data cache. Other data caches could prove equally fruitful. We also plan to apply the analysis method to instruction caches.

While the attack results we gave were for one particular cryptosystem implementation, the analysis method has a much wider range of applications. We



in fact found a similar vulnerability in the NSS library’s implementation of elliptic curves. Departing from elliptic curves and public key cryptography, we plan to apply the analysis to an assortment of implementations, asymmetric and symmetric primitives alike.

One of the more interesting planned applications is to algorithms with good side channel resistance properties, such as “Montgomery’s ladder”. While this might be an overwhelming challenge for traditional power analysis, the work here emphasizes the fact that cache-timing attacks are about memory access patterns; a fixed sequence of binary operations cannot be assumed sufficient to thwart cache-timing attacks.

**Acknowledgments** We thank the following people for comments and discussions: Dan Bernstein, Kimmo Järvinen, Kaisa Nyberg, and Dan Page.

## References

1. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Koblitz, N., ed.: CRYPTO 1996. Volume 1109 of LNCS. Springer (1996) 104–113
2. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In Wiener, M.J., ed.: CRYPTO 1999. Volume 1666 of LNCS. Springer (1999) 388–397
3. Page, D.: Defending against cache based side-channel attacks. Information Security Technical Report **8**(1) (2003) 30–44
4. Oswald, E.: Enhancing simple power-analysis attacks on elliptic curve cryptosystems. In Kaliski, Jr, B.S., Koç, Ç.K., Paar, C., eds.: CHES 2002. Volume 2523 of LNCS. Springer (2003) 82–97
5. Karlof, C., Wagner, D.: Hidden Markov model cryptanalysis. In Walter, C.D., Koç, Ç.K., Paar, C., eds.: CHES 2003. Volume 2779 of LNCS. Springer (2003) 17–34
6. Green, P.J., Noad, R., Smart, N.P.: Further hidden Markov model cryptanalysis. In Rao, J.R., Sunar, B., eds.: CHES 2005. Volume 3659 of LNCS. Springer (2005) 61–74
7. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In Kaliski, Jr, B.S., Koç, Ç.K., Paar, C., eds.: CHES 2002. Volume 2523 of LNCS. Springer (2003) 13–28
8. Medwed, M., Oswald, E.: Template attacks on ECDSA. In Chung, K.I., Sohn, K., Yung, M., eds.: WISA 2008. Volume 5379 of LNCS. Springer (2008) 14–27
9. Percival, C.: Cache missing for fun and profit. <http://www.daemonology.net/papers/cachemissing.pdf> (2005)
10. Hlaváč, M., Rosa, T.: Extended hidden number problem and its cryptanalytic applications. In Biham, E., Youssef, A.M., eds.: SAC 2006. Volume 4356 of LNCS. Springer (2006) 114–133
11. Bernstein, D.J.: Cache-timing attacks on AES. <http://cr.yp.to/papers.html#cachetiming> (2004)
12. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In Pointcheval, D., ed.: CT-RSA 2006. Volume 3860 of LNCS. Springer (2006) 1–20
13. Möller, B.: Algorithms for multi-exponentiation. In Vaudenay, S., Youssef, A., eds.: SAC 2001. Volume 2259 of LNCS. Springer (2001) 165–180

14. Möller, B.: Improved techniques for fast exponentiation. In Lee, P.J., Lim, C.H., eds.: ICISC 2002. Volume 2587 of LNCS. Springer (2003) 298–312
15. Bosma, W.: Signed bits and fast exponentiation. *Journal de Théorie des Nombres de Bordeaux* **13**(1) (2001) 27–41
16. Kohonen, T.: *Self-Organizing Maps*. Springer (1995)
17. Lloyd, S.: Least squares quantization in PCM. *IEEE Transactions on Information Theory* **28**(2) (1982) 129–137
18. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE* **77**(2) (1989) 257–286
19. Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* **13**(2) (1967) 260–269
20. Baum, L.E., Petrie, T., Soules, G., Weiss, N.: A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics* **41**(1) (1970) 164–171
21. Menezes, A., van Oorschot, P., Vanstone, S.: *Handbook of Applied Cryptography*. 5 edn. CRC Press (2001)
22. Howgrave-Graham, N., Smart, N.P.: Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography* **23**(3) (2001) 283–290
23. Nguyen, P.Q., Shparlinski, I.: The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, Codes and Cryptography* **30**(2) (2003) 201–217
24. Leadbitter, P.J., Page, D., Smart, N.P.: Attacking DSA under a repeated bits assumption. In Joye, M., Quisquater, J.J., eds.: CHES 2004. Volume 3156 of LNCS. Springer (2004) 428–440
25. Coron, J.S.: Resistance against differential power analysis for elliptic curve cryptosystems. In Koç, Ç.K., Paar, C., eds.: CHES 1999. Volume 1717 of LNCS. Springer (1999) 292–302
26. Clavier, C., Joye, M.: Universal exponentiation algorithm: a first step towards provable SPA-resistance. In Koç, Ç.K., Naccache, D., Paar, C., eds.: CHES 2001. Volume 2162 of LNCS. Springer (2001) 300–308
27. Möller, B.: Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks. In Chan, A.H., Gligor, V.D., eds.: ISC 2002. Volume 2433 of LNCS. Springer (2002) 402–413
28. Oswald, E., Aigner, M.: Randomized addition-subtraction chains as a countermeasure against power attacks. In Koç, Ç.K., Naccache, D., Paar, C., eds.: CHES 2001. Volume 2162 of LNCS. Springer (2001) 39–50
29. Walter, C.D.: Randomized exponentiation algorithms. In Koç, Ç.K., ed.: *Cryptographic Engineering*. Springer (2009)
30. Viega, J., Messier, M., Chandra, P.: *Network Security with OpenSSL*. O’Reilly Media, Inc. (2002)
31. Chevallier-Mames, B., Ciet, M., Joye, M.: Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *IEEE Transactions on Computers* **53**(6) (2004) 760–768