A preliminary version of this paper appears in the proceedings of CT-RSA 2015. This is the full version.

# Cold Boot Attacks in the Discrete Logarithm Setting

Bertram Poettering[1] and Dale L. Sibborn[2]

[1] Ruhr University Bochum
[2] Royal Holloway, University of London

**Abstract.** In a cold boot attack a cryptosystem is compromised by analysing a noisy version of its internal state. For instance, if a computer is rebooted the memory contents are rarely fully reset; instead, after the reboot an adversary might recover a noisy image of the old memory contents and use it as a stepping stone for reconstructing secret keys. While such attacks were known for a long time, they recently experienced a revival in the academic literature. Here, typically either RSA-based schemes or blockciphers are targeted.
We observe that essentially no work on cold boot attacks on schemes defined in the discrete logarithm setting (DL) and particularly for elliptic curve cryptography (ECC) has been conducted. In this paper we hence consider cold boot attacks on selected wide-spread implementations of DL-based cryptography. We first introduce a generic framework to analyse cold boot settings and construct corresponding key-recovery algorithms. We then study common in-memory encodings of secret keys (in particular those of the wNAF-based and comb-based ECC implementations used in OpenSSL and PolarSSL, respectively), identify how redundancies can be exploited to make cold boot attacks effective, and develop efficient dedicated key-recovery algorithms. We complete our work by providing theoretical bounds for the success probability of our attacks.

## 1 Introduction

*Cold boot attacks.* Since they were reported in the literature by Halderman *et al.* in 2008 [4], cold boot attacks have received a great deal of attention. The attacks rely on the fact that computer memory (in particular when consisting of DRAM cells) typically retains information when going through a power-down power-up cycle; this might allow an adversary to get access to confidential information such as cryptographic keys after a system reboot. Unfortunately (for such an attacker), while the power is cut the bits in memory will decay over time, which means that any information obtained is likely to be 'noisy'. The focus of cold boot attacks resides in modelling quality and quantity of noise and applying intelligent algorithms to extracted memory images in order to fully reconstruct keys.

The amount of time for which information is retained without power depends on the particular memory type (modern technologies imply a quicker decay) and the environment temperature (information degrades more quickly at higher temperatures). The experiments of [4] demonstrate that, at normal operating temperatures, there is little corruption within the first few seconds, which is then followed by rapid decay. The period of low corruption can be drastically prolonged by cooling the memory chips. For instance, according to [4], in an experiment at $-50\,^{\circ}\mathrm{C}$ ($-58\,^{\circ}\mathrm{F}$) (which can be achieved by spraying compressed air onto the memory chips) less than 0.1% of bits decay within sixty seconds. At temperatures of approximately $-196\,^{\circ}\mathrm{C}$ ($-321\,^{\circ}\mathrm{F}$) (via the use of liquid nitrogen) less than 0.17% of bits decay within one hour without power.

After power is switched off, the decay proceeds in a quite predictable pattern [3,21]. More precisely, memory is partitioned into regions, and each region has a 'ground state' which is either 0 or 1. In a 0 ground state, the 1 bits will eventually decay to a 0. The probability of a 0 bit decaying to a 1 is very small, but not vanishing (a typical probability is 0.001 [4]). When the ground state is 1, the opposite is true.

*Previous cold boot key-recovery algorithms.* The general possibility of using cold boot attacks to recover data from memory chips has been known since at least the 1970s. However, in the academic literature it was not until 2008 that Halderman *et al.* became the first to focus on reconstructing cryptographic private keys from information obtained via this type of attack. There is now an abundance of literature concerning the recovery of private keys in this manner. RSA key-recovery algorithms are without doubt the most popular [4,8,9,12,14,15,18,20], whilst symmetric-key cryptographic schemes have received a

little less attention [1,4,11,23]. One area that remains comparatively unexplored is the discrete logarithm setting. As far as we are aware, this issue has only been discussed in [15].

Published cold boot analyses almost ubiquitously assume that attackers can obtain a (noisy) copy of a private key that was stored with some form of redundancy. For instance, in the case of RSA, while in principle it is only necessary for the private key to contain the prime factors $p$ and $q$, the PKCS#1 standard [10] suggests storing several extra values (such as $d, d_p, d_q$, and $q_p^{-1}$) in order to increase the efficiency of decryption and signing operations. It is this redundancy that was exploited by previous authors to recover private keys even when they were subjected to very high noise levels. In contrast, the discrete logarithm analysis of Lee *et al.* [15] assumes that an attacker only has access to the public key $X = g^x$ and a decayed version of the private key $x$. Consequently, given that there is no further redundancy, their proposed algorithm would be unable to efficiently recover keys that were affected by high noise levels.

*Limitations of previous work.* There has previously only been one paper that considers key-recovery in the discrete logarithm setting [15]. We believe that this paper delivers only a small advantage over brute-force attacks. Furthermore, the analysis seems to be flawed: in an execution of a cold boot attack, the authors assume that the number of key bits that flip is upper-bounded by the expected number of bits that will flip. When the number of bit flips exceeds the expected amount, the algorithm will fail to recover the private key, but this is not accounted for in the analysis. Furthermore, [15] does not explicitly cover the case of asymmetric errors that appear in physical cold boot attacks, despite the fact that this is the motivation for the paper. Instead, the focus is on an idealised setting in which only 1 or 0 bits flip, but not both: however, in practice both 0 and 1 bits have the potential to flip, as described above.

*Our contributions.* Given the above discussion on the results of [15] it is natural to ask whether in practical discrete logarithm-based software implementations there are any private key representations that contain redundancy that can be used to improve cold boot key-recovery algorithms. It turns out that such cases are indeed the rule, and they will form the basis of this paper. The scenarios we consider are taken from two wide-spread ECC implementations found in TLS libraries: the windowed non-adjacent form (wNAF) representation used in OpenSSL, and the PolarSSL comb-based approach. By exploiting redundancies in the respective in-memory representations of private keys we are able to vastly improve upon the results from [15].

Our techniques are based on a novel statistical test that allows a trade-off between success rate and execution speed. We stress that this test is not only applicable to the discrete logarithm setting, but is applicable to all types of key. In particular, it complements the framework of Paterson *et al.* [18] for the RSA setting. We observe that the statistical test proposed in [18] has a bounded running-time, but no lower bound on the success of the algorithm is provided in the scenario where keys are subjected to asymmetric errors. In contrast, for our algorithm we succeed in lower-bounding the success rate. Although we provide no bound on the running time of our primary algorithm, we note that various modifications allow an attacker to seek her own compromise between a preferred success rate and a desired running-time.

On a side note, we provide what we believe to be the first explanation in the academic literature of the point multiplication routine deployed in PolarSSL.

## 2 Multinomial distributions and the multinomial test

The general strategy behind key-recovery procedures for cold boot attacks is to only consider small parts of the targeted key at a time. For instance, RSA-based reconstruction procedures usually start with the least significant bits (LSB) [8,9,13,14,15,18,20], but it is also possible to begin with the most significant bits (MSB) [19]. It is typical to use an iterative process to guess a couple of bits of the key and assess the plausibility of the guess on the basis of both a model of the decay process and the available redundancy in the encoding. Previous cold boot papers have proposed various methods by which the plausibility of the guess is ascertained. Examples are the Hamming distance approach of [8] and the maximum-likelihood method of [18]. The theoretical success of the algorithm is usually based on assumptions that are typically only true for a specific key being considered, and are possibly not easy to generalise. In this section we propose a general statistical test that can be used in various scenarios. The test is based on multinomial

distributions and works well for scenarios when the distribution of private key bits is known (such as RSA), but can also be modified to work even when the attacker knows nothing of the distribution of the private key.

We will now study the multinomial distribution and its associated test. Multinomial distributions are a generalisation of the binomial distribution. The distribution has $k$ mutually exclusive events with probabilities $p = (p_1, \ldots, p_k)$, where $\sum_{i=1}^{k} p_i = 1$ and for all $i$ we have $p_i \neq 0$. If there are $n$ trials, we let the random variables $X_1, \ldots, X_k$ denote the amount of times the $i$th event occurs and say that $X = (X_1, \ldots, X_k)$ follows a multinomial distribution with parameters $n$ and $p$. Given a set of observed values, $x = (x_1, \ldots, x_k)$, we can use a multinomial test to see if these values are consistent with the probability vector $p$ (which is the null hypothesis, denoted $H_0$). The alternative hypothesis (denoted $H_1$) for the probability vector is $\pi = (x_1/n, \ldots, x_k/n)$, where each component is the maximum-likelihood estimate for each probability. The two hypotheses can be compared via the calculation $-2\sum_{i=1}^{k} x_i \ln(p_i/\pi_i)$, which is called the *multinomial test statistic*. When the null hypothesis is true, the distribution of this statistic converges to the chi-squared distribution with $k-1$ degrees of freedom as $n \to \infty$.

We will now see how the multinomial test statistic may be applied in cold boot key recovery algorithms. Let $s_i$ denote a (partial) candidate solution for the private key (including the redundant representation) across a (partial) section of bits. When comparing a partial candidate solution $s_i$ to the noisy information $r$ we define $n_{01}^i$ to be the number of positions at which there is a 0 in the candidate solution and a 1 in the corresponding position in the noisy information $r$. We define $n_{00}^i$, $n_{10}^i$, and $n_{11}^i$ correspondingly, so $n = n_{00} + n_{01} + n_{11} + n_{10}$. Crucially, this count only considers the newly-guessed bits generated at the relevant phase of the algorithm, while all previous bits are ignored. It is clear that these counts follow a multinomial distribution. Let $\alpha := \mathbb{P}(0 \to 1)$ denote the probability that a 0 bit flips to a 1 in the execution of the cold boot attack, and let $\beta := \mathbb{P}(1 \to 0)$ denote the probability that a 1 flips to a 0. For the correct candidate solution, $s_c$, the probability of observing each of the four values $(n_{00}^c, n_{01}^c, n_{11}^c, n_{10}^c)$ is precisely $H_0 : p = (p_0(1-\alpha), p_0\alpha, p_1(1-\beta), p_1\beta)$, where $p_b$, $b \in \{0, 1\}$, is the probability of a $b$-bit appearing in the correct candidate solution. Notice that we require $\alpha, \beta \neq 0$ since each component of the probability vector must be non-zero. The test may be modified to cover the case when $\alpha$ or $\beta$ is zero, but we defer this discussion to the appendix. For each candidate solution we could use the previous set of probabilities as the null hypothesis of the multinomial test. We would like to test whether our guessed candidate solution is consistent with this probability vector. The alternative hypothesis is that the set of probabilities for the four bit-pairs is equal to the maximum-likelihood estimates for each category. That is, $H_1 : p = (n_{00}^i/n, n_{01}^i/n, n_{11}^i/n, n_{10}^i/n)$ for each candidate $i$. We define our first statistical test, which we call Correlate$'$, to be

$$\begin{aligned} \text{Correlate}'(s_i, r) := &-2n_{00}^i \ln\left(\frac{np_0(1-\alpha)}{n_{00}^i}\right) - 2n_{01}^i \ln\left(\frac{np_0\alpha}{n_{01}^i}\right) \\ &-2n_{11}^i \ln\left(\frac{np_1(1-\beta)}{n_{11}^i}\right) - 2n_{10}^i \ln\left(\frac{np_1\beta}{n_{10}^i}\right) \ , \end{aligned} \tag{1}$$

where the values in brackets are the null hypothesis values divided by the alternative hypothesis values. Correlate$'$ outputs a numerical value ($\geq 0$) for each candidate. We now need to discuss when we consider this test to pass or fail. It is well known that when the null hypothesis is correct the distribution of the right-hand side of the equation (1) converges to a chi-squared distribution with $k-1$ degrees of freedom as $n \to \infty$. In our analysis we have $k = 4$, hence the test statistic converges to a chi-squared distribution with three degrees of freedom. We can therefore set a threshold $C$ such that any candidate whose test statistic is less than $C$ is retained, otherwise the candidate is discarded. We therefore define

$$\text{Correlate}_C(s_i, r) = \text{pass} \quad \Leftrightarrow \quad \text{Correlate}'(s_i, r) < C \ ,$$

where $C$ would be an additional (user-chosen) input to the algorithm. The chi-squared distribution can tell us how to set the threshold $C$ to achieve any desired success rate. If we set the threshold $C$ such that $\int_0^C \chi_3^2(x)\mathrm{d}x = \gamma$, we know that, asymptotically, the probability that the correct candidate's correlation value Correlate$'(s_i, r)$ is less than $C$ is equal to $\gamma$. Recall that the Correlate$'$ test only considers the newly generated bits at each stage of the algorithm, and all previous bits are ignored. This eases the success analysis of the algorithm since the probability of passing each Correlate test is independent in this case. Therefore, if the private key has been parsed into $m$ distinct parts, and the attacker applies a Correlate

test to each of the $m$ parts, the probability that the correct private key is recovered is $\gamma^m$, assuming the same threshold $C$ was used for each Correlate test.

The only issue yet to be addressed is specifying the values that $p_0$ and $p_1$ should take. If the distribution of the private key is known, then it is easy to assign values to these parameters. For example, in the RSA setting, the analyses of [9,18] assume that the entire private key would have approximately an equal number of zeros and ones. Therefore, if we were to use the Correlate$'$ test (equation (1)) in the RSA setting we would set $p_0 = p_1 = 1/2$. Notice that this immediately gives us a threshold-based approach for recovering noisy RSA private keys that have been degraded according to an asymmetric binary channel (i.e., $\alpha \neq \beta$), and such an approach is currently lacking in the literature.

In other settings it may not be possible to accurately assign values to these parameters. The approach we use to overcome this issue is to conduct two separate multinomial tests: one for the 0 bits and another for the 1s. The advantage of using two separate tests is that we do not need to estimate the values of $p_0$ and $p_1$, and hence our algorithm's success will not be harmed by a poor estimation of these parameters. For the correct solution, each 0 can flip to a 1 with probability $\alpha$ or it can remain a 0 with probability $1-\alpha$. Hence, if there are $n_0$ zeros in the correct solution, then $(n_{01}, n_{00})$ follows a multinomial distribution with parameters $n_0$ and $p = (\alpha, 1-\alpha)$. Similarly, if there are $n_1$ ones in the correct solution, then $(n_{10}, n_{11})$ follows a multinomial distribution with parameters $n_1$ and $p = (\beta, 1-\beta)$. We may now use the multinomial test to examine each candidate solution without having to estimate $p_0$ and $p_1$. Specifically, we define

$$\text{Correlate}^0(s_i, r) := -2n_{00}^i \ln\left(\frac{n_0(1-\alpha)}{n_{00}^i}\right) - 2n_{01}^i \ln\left(\frac{n_0\alpha}{n_{01}^i}\right) \tag{2}$$

and

$$\text{Correlate}^1(s_i, r) := -2n_{11}^i \ln\left(\frac{n_1(1-\beta)}{n_{11}^i}\right) - 2n_{10}^i \ln\left(\frac{n_1\beta}{n_{10}^i}\right) \ . \tag{3}$$

Then we define

$$\text{Correlate}_C(s_i, r) := \text{pass} \quad \Leftrightarrow \quad \text{Correlate}^0(s_i, r) < C \wedge \text{Correlate}^1(s_i, r) < C \ . \tag{4}$$

Notice that Correlate$^0$ and Correlate$^1$ are functions with one degree of freedom. Therefore the probability that Correlate$^0 < C$ is $\gamma = \int_0^C \chi_1^2(x)\mathrm{d}x$. The same holds for the probability that Correlate$^1 < C$.

### 2.1 Convergence of the multinomial test

One issue with the approach proposed in the previous section is that the Correlate function will only consider a small number of bits at a time, but the convergence of the test to the chi-squared distribution is an asymptotic result. Hence, given the small sample sizes, there will be some discrepancy between what we observe in practice and what is predicted by the chi-squared statistic. However, for small sample sizes it is known that the multinomial test converges to the chi-squared distribution from above. Therefore, in practice the success rate will be greater than that predicted by the chi-squared distribution. As a result, the chi-squared test essentially gives us a lower bound for success. There are various modifications that can be made to the multinomial test in order to force a better agreement with the chi-squared test [22,24], but we will not explore these avenues in this paper.

## 3 Exponentiation algorithms

As all discrete log keys considered in this paper are defined over elliptic curves we use the additive notation $Q = aP$ to denote a public key $Q$, where $P$ is the base point and scalar $a$ is the private key. We write $\mathcal{O}$ for the point at infinity, i.e., the neutral element in that group.

A core part of any DLP-based cryptosystem realised in the elliptic curve setting is a point multiplication routine. Here, a curve point $P$, also refered to as base point, is multiplied with a scalar $a \in \mathbb{N}$ to obtain another curve point $Q = aP$. The overall performance of this operation depends on various factors, including the representation of field elements (e.g., 'pseudo-Mersenne' vs. 'Montgomery-friendly' moduli for prime fields), the availability of optimised formulas for basic group operations like point addition and doubling (e.g., 'Weierstrass' vs. 'Edwards' curves), the representation of curve points (e.g., 'affine' vs. 'projective'), and the scheduler that specifies how the basic group operations are combined to

achieve a full point multiplication algorithm (see [2] for a recent survey on available options and tradeoffs in all these categories). In the context of cold boot attacks particularly the scheduler seems to be an interesting target to analyse: in ECC-based cryptosystems, secret keys typically correspond with scalars, i.e., with precisely the information the scheduler works with. In the following we give a brief overview over the most relevant such algorithms [5]. We analyse the resilience of specific instances against cold book attacks in later sections of this paper.

### 3.1 (Windowed) Double-and-Add

The textbook method for performing point multiplication is the *double-and-add* algorithm[3]. Given scalar $a \in \mathbb{N}$ and an appropriate length parameter $\ell \in \mathbb{N}$, it requires that $a$ is represented by its *binary expansion* $[a]_1 = (a_\ell, \ldots, a_0)$, where $a = \sum_{i=0}^{\ell} a_i 2^i$ and $a_\ell, \ldots, a_0 \in \{0, 1\}$. Given $[a]_1$, and denoting 'right-shifting' $a$ by $k$ positions with $a \gg k$, we observe

$$aP = \left(\sum_{i=0}^{\ell} a_i 2^i\right) P = 2\left(\sum_{i=0}^{\ell-1} a_{i+1} 2^i\right) P + a_0 P = 2(a \gg 1)P + a_0 P \ .$$

This recursion can be unrolled to

$$aP = 2(2(2(\ldots + a_3 P) + a_2 P) + a_1 P) + a_0 P \ . \tag{5}$$

The double-and-add algorithm for computing $Q = aP$ is now immediate: it initializes $Q$ with $\mathcal{O}$ and iteratively updates $Q \leftarrow 2Q + a_i P$, where the $a_i$ are considered 'left-to-right' (i.e., $i$ counts backwards from $\ell$ down to 0). The whole procedure takes approximately $\ell/2$ additions and $\ell$ doublings per point multiplication, if uniform exponents are assumed.

A common approach to improve the efficiency of this algorithm is to decrease the number of required additions by applying a windowing technique. More precisely, for fixed *window size* $w$, we define the notion of *windowed binary expansion* as above, but this time relaxing the requirement on the $a_i$ to $a_\ell, \ldots, a_0 \in [0 \,..\, 2^w - 1]$ and using notation $[a]_w = (a_\ell, \ldots, a_0)$. While such an encoding is in general not unique, it can be shown to uniquely exist if additionally either $a_i = 0$ for all $i \not\equiv 0 \pmod{w}$ is required (*fixed window*), or it is required that all non-zero $a_i$ be odd and all $w$-length subsequences of $[a]_w$ contain at most one such element (*sliding window*).

Observe that, if we assume the fixed-window case and that points $P, 2P, \ldots, (2^w - 1)P$ are precomputed, then $w - 1$ out of $w$ additions vanish from equation (5). Even more additions potentially vanish in the sliding-window case; further, as here all non-zero coefficients $a_i$ are odd, fewer points have to be precomputed.

*Example 1.* For $a = 30$ and $\ell = 6$ we have $[a]_1 = (0, 0, 1, 1, 1, 1, 0)$. Windowed binary expansions for $w = 2$ are $(0, 0, 1, 0, 3, 0, 2)$ (fixed window) and $(0, 0, 0, 3, 0, 3, 0)$ (sliding window).

### 3.2 (Windowed) Signed-digit representations

Many different ways to represent elliptic curve points have been proposed [5,2]; a common property of all these encodings is that group negation is a cheap operation. For instance, for curves in Weierstrass form that are defined over prime fields, e.g., the five 'prime curves' standardized by NIST, the negative of a point $(x, y)$ is simply $(x, -y)$. A general consequence of this is that point subtraction performs as efficient as point addition. This is exploited in point multiplication algorithms that are based on the *signed-digit representation* of scalars.

Formally, for fixed window size $w$ we denote with $[a]_{\pm w} = (a_\ell, \ldots, a_0)$ any decomposition of $a \in \mathbb{N}$ such that $a = \sum_{i=0}^{\ell} a_i 2^i$ and $a_i \in [-2^{w-1} \,..\, 2^{w-1} - 1]$. As equation (5) still holds if some of the coefficients $a_i$ are negative, a 'double-and-add-or-subtract' algorithm that operates on such signed-digit representations is readily derived. The key idea is that the extra freedom obtained by allowing coefficients to be negative will make it possible to find particularly sparse scalar representations, i.e., representations for which only a minimum number of group additions/subtractions is required.

[3] This is known as the square-and-multiply algorithm if the group is written multiplicatively.

We describe three common signed-digit normal forms for representing scalars $a \in \mathbb{N}$. The first one, *non-adjacent form* (NAF), limits the digit set to $\{0, \pm 1\}$ and requires that no two consecutive coefficients are non-zero. The second and third are defined in respect to a window size $w$. Specifically, while the *fixed-window NAF* is an encoding of the form $[a]_{\pm w}$ that requires $a_i = 0$ for all $i \not\equiv 0 \pmod{w}$, the *sliding-window NAF* (wNAF) ensures that all non-zero $a_i$ are odd and all $w$-length subsequences of $[a]_{\pm w}$ contain at most one such element. All three types of encoding are unique. Note that in the $w = 1$ case the notions of NAF and wNAF coincide. Observe also that storing a NAF or wNAF might require one extra digit over the plain binary expansion. For an example of the latter consider that the binary expansion of the decimal number 15 is the sequence $(1, 1, 1, 1)$, while its NAF is $(1, 0, 0, 0, \bar{1})$, where we write $\bar{1}$ for $-1$.

*Example 2.* The NAF of $a = 30$ is $(0, 1, 0, 0, 0, \bar{1}, 0)$. For window size $w = 2$ the fixed-window NAF is $(1, 0, \bar{2}, 0, 0, 0, \bar{2})$ and the wNAF is $(0, 1, 0, 0, 0, \bar{1}, 0)$.

Algorithm 1 gives instructions on how to derive the wNAF of a scalar $a \in \mathbb{N}$. Observe that the computation is conducted in a greedy right-to-left fashion, with a $(w-1)$-look-ahead. As the latter property will become relevant in our later analyses, we state it formally.

**Fact 1 (Suffix property of wNAF)** *Fix a scalar $a \in \mathbb{N}$ and a window size $w$. Denote $a$'s binary expansion with $(a_\ell, \ldots, a_0)$ and its wNAF with $(b_\ell, \ldots, b_0)$, for an appropriate length parameter $\ell$. Then for all $0 \leq t \leq \ell - w + 1$ it holds that $(b_t, \ldots, b_0)$ is fully determined by $(a_{t+w-1}, \ldots, a_0)$.*

**Algorithm 1** Textbook wNAF encoding

**Input:** scalar $a$, length parameter $\ell$, window size $w$
**Output:** wNAF $(b_\ell, \ldots, b_0)$

**for** $i \leftarrow 0$ **to** $\ell$ **do**
  **if** $a$ is odd **then**
    $b_i \leftarrow a \operatorname{smod} 2^w$
  **else**
    $b_i \leftarrow 0$
  $a \leftarrow (a - b_i) \gg 1$
**return** $(b_\ell, \ldots, b_0)$

**Algorithm 2** Textbook comb encoding

**Input:** scalar $a$, parameters $w, d$
**Output:** coefficients $K^{d-1}, \ldots, K^0$

**for** $i \leftarrow 0$ **to** $d - 1$ **do**
  $K^i \leftarrow (a_{i+(w-1)d}, \ldots, a_{i+d}, a_i)$
**return** $K^{d-1}, \ldots, K^0$

Fig. 1: In Algorithm 1, operator 'smod' computes signed remainders of integer divisions by powers of two. Precisely, for integers $a, b$ we have $b = a \operatorname{smod} 2^w$ iff $\exists k : a = k2^w + b \ \wedge \ b \in [-2^{w-1} .. 2^{w-1} - 1]$.

### 3.3 Point multiplication in OpenSSL

We give details about the elliptic curve point multiplication routine used in OpenSSL. Specifically, we studied the code from file `crypto/ec/ec_mult.c` of OpenSSL version 1.0.1h from March 2012, which is the latest stable release. Relevant for this work is particularly the function `compute_wNAF` defined in line 193, which computes a so-called *modified wNAF*. In brief, while a regular wNAF requires every $w$-length subsequence of digits to contain at most one non-zero element, in modified wNAFs [17] this requirement is relaxed for the most significant non-zero position, in order to potentially allow saving a final doubling operation. For instance, in case $w = 2$ this relaxation allows to encode decimal number 11 as $(1, 1, 0, \bar{1})$, while the corresponding (strict) wNAF would be $(1, 0, \bar{1}, 0, \bar{1})$. OpenSSL's `compute_wNAF` function computes the modified wNAF following Algorithm 3, with default window size $w = 4$ (see line 816). The resulting coefficients $b_i \in [-2^{w-1} .. 2^{w-1} - 1]$ are encoded into an array of octets (data type '`signed char`'), using a standard two-complement in-memory representation. For instance, we have

$$-3 \mapsto \texttt{11111101} \quad -1 \mapsto \texttt{11111111} \quad 0 \mapsto \texttt{00000000} \quad +1 \mapsto \texttt{00000001} \quad +3 \mapsto \texttt{00000011} \ .$$

We confirmed that the OpenSSL forks LibreSSL[4] (version 2.0.3 from July 2014) and BoringSSL[5] (version from July 2014) use precisely the same exponent encoding as described above.

**Algorithm 3** OpenSSL's wNAF encoding

**Input:** scalar $a$, length parameter $\ell$, window size $w$
**Output:** modified wNAF $(b_\ell, \ldots, b_0)$

compute $b = (b_\ell, \ldots, b_0)$ using Algorithm 1
**if** $b$ has prefix $0^*10^{w-1}\beta$ with $\beta < 0$ **then**
    $\bar{\beta} \leftarrow 2^{w-1} + \beta$
    in $b$, replace substring $10^{w-1}\beta$ by $010^{w-2}\bar{\beta}$
**return** $b$

**Algorithm 4** PolarSSL's comb encoding

**Input:** odd scalar $a$, parameters $w, d$
**Output:** coefficients $K^d, (\sigma^{d-1}, K^{d-1}), \ldots, (\sigma^0, K^0)$

compute $(\bar{K}^{d-1}, \ldots, \bar{K}^0)$ using Algorithm 2
$K^0 \leftarrow \bar{K}^0$
$c \leftarrow (0, \ldots, 0)$
**for** $i \leftarrow 1$ **to** $d-1$ **do**
    **if** $\bar{K}^i_0 = 0$ **then**
        $(c, K^i) \leftarrow \bar{K}^i \boxplus K^{i-1} \boxplus c$
        $\sigma^{i-1} \leftarrow -1$
    **else**
        $(c, K^i) \leftarrow \bar{K}^i \boxplus c$
        $\sigma^{i-1} \leftarrow +1$
**return** $c, (+1, K^{d-1}), \ldots, (\sigma^0, K^0)$

Fig. 2: In Algorithm 4, for same-size bit-vectors $\alpha, \beta, \gamma, \delta, \epsilon$ we write $(\alpha, \beta) = \gamma \boxplus \delta$ iff $2\alpha_i + \beta_i = \gamma_i + \delta_i$ for all $i$. Correspondingly we write $(\alpha, \beta) = \gamma \boxplus \delta \boxplus \epsilon$ iff $2\alpha_i + \beta_i = \gamma_i + \delta_i + \epsilon_i$. That is, the addition is bit-wise and the sum is stored in $\beta_i$, with $\alpha_i$ taking the carry.

### 3.4 Comb-based methods

The various methods for point multiplication that we studied in the preceding sections aimed at requiring less point additions than the basic double-and-add technique; the number of doubling operations, however, remained invariant (or was even increased). In contrast, comb-based methods [16] get along with significantly fewer doublings—at the expense of some precomputation dependent on the base point. In the following we give a rudimentary introduction to comb-based multiplication techniques. See [5] for further details.

Fix a base point $P$ and parameters $w, d \in \mathbb{N}$. For any scalar $a \in \mathbb{N}$ with $0 \le a < 2^{wd}$ let $[a]_1 = (a_{wd-1}, \ldots, a_0)$ denote its binary expansion. For all $i \in [0\,..\,d-1]$ let $K^i = (K^i_{w-1}, \ldots, K^i_0)$ where $K^i_j = a_{i+jd}$, as formalized by Algorithm 2 and illustrated in Figure 3. That is, as values $K^i_j \in \{0, 1\}$ are assigned such that

$$a = \sum_{i=0}^{wd-1} 2^i a_i = \sum_{i=0}^{d-1} \sum_{j=0}^{w-1} 2^{i+jd} K^i_j = \sum_{i=0}^{d-1} 2^i \sum_{j=0}^{w-1} 2^{jd} K^i_j \ ,$$

we have that

$$aP = \sum_{i=0}^{d-1} 2^i T(K^i_{w-1}, \ldots, K^i_0) \qquad \text{where} \qquad T\colon (k_{w-1}, \ldots, k_0) \mapsto \sum_{j=0}^{w-1} 2^{jd} k_j P \ .$$

The fundamental idea behind comb-based point multiplication is to precompute table $T$; as we have seen, the remaining part of the operation can then be conducted with not more than $d$ additions and doublings.

As first observed by Hedabou *et al.* [6], implementations of the described point multiplication method might offer only limited resilience against side-channel attacks based on simple power analysis (SPA). This comes from the fact that any vector $K^i = (K^i_{w-1}, \ldots, K^i_0)$ is equal to $(0, \ldots, 0)$ with probability about $2^{-w}$ and that, in the multiplication process, this condition implies adding neutral element

[4] http://www.libressl.org/, see file `crypto/ec/ec_mult.c`
[5] https://boringssl.googlesource.com/, see file `crypto/ec/wnaf.c`

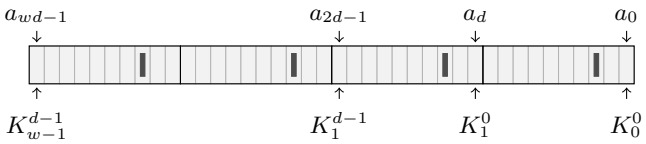


Fig. 3: Visualization of the comb method, for parameters $(w, d) = (4, 10)$. The cells represent the bits of the scalar, the bold rectangles mark the prongs of a comb positioned at offset $i = 2$.

$T(0, \ldots, 0) = \mathcal{O}$ to the current accumulator—an event that is likely recognizable by analysing power traces.

To mitigate the threat, [6] proposes a comb-based scheduler where situation $K^i = (0, \ldots, 0)$ does not occur. In a nutshell, it (a) considers only odd scalars (this guarantees $K^0 \neq (0, \ldots, 0)$), (b) introduces for each $i \in [0\,..\,d-1]$ a flag $\sigma^i \in \{\pm 1\}$ that defaults to $\sigma^i = +1$ and indicates whether the corresponding $K^i$ should be considered 'positive' or 'negative', and (c) examines vectors $K^1, \ldots, K^{d-1}$ (in that order) and for each particular $K^i$ that is equal to $(0, \ldots, 0)$ it updates $K^i \leftarrow K^{i-1}$ and $\sigma^{i-1} \leftarrow -1$. Observe that restriction (a) does not impose a real limitation in groups of prime order $q$ because $aP = -(-aP) = -(q-a)P$ and either $a$ or $q-a$ is odd. Observe also that the steps introduced in (c) do not affect the overall outcome of the point multiplication as for all integers $x$ we have $x = 2 \cdot x + (-1) \cdot x$.

In [7], the same authors improve on their proposal by first encoding (odd) scalars $a = \sum a_i 2^i$ using only signed binary digits $a_i \in \{\pm 1\}$, and then computing vectors $K^i$ from these coefficients. This not only avoids the $K^i = (0, \ldots, 0)$ situation but also reduces the size of the precomputed table by a factor of two.

### 3.5 Point multiplication in PolarSSL

We analysed the source code of the point multiplication routine deployed in PolarSSL[6] version 1.3.8, published on July 11 2014. The scheduler (function `ecp_comb_fixed` in file `library/ecp.c`) is comb-based, and comments in the code give explicit credit to the results of [6]. However, as a matter of fact the actually implemented algorithm significantly improves on the referred-to work, as we detail below. We believe that this is the first description of this point multiplication method in the academic literature.

PolarSSL borrows from [6] both the restriction to handle only odd scalars and the introduction of flags $\sigma^i \in \{\pm 1\}$ that indicate whether corresponding $K^i$ are considered 'positive' or 'negative'. Novel is that the iteration over $K^1, \ldots, K^{d-1}$ that before was solely concerned about fixing the $K^i = (0, \ldots, 0)$ condition is now replaced by an iteration over the same values where action is taken roughly every second time, namely whenever $K^i_0 = 0$. Concretely, in this case the algorithm sets $\sigma^{i-1} \leftarrow -1$ (similarly to [6]) and replaces $K^i$ by $K^i \boxplus K^{i-1}$, where addition '$\boxplus$' is understood position-wise, carrying over into $K^{i+1}$. This method ensures that all $K^i$ have $K^i_0 = 1$ (as is easily shown by an inductive argument), and effectively makes precomputed table $T$ half-size. On the downside, for recording the carries of the final '$\boxplus$' step, vector $K^{d-1}, \ldots, K^0$ has to be extended by an auxiliary component $K^d$. The details on the procedure are given in Algorithm 4.

We conclude by describing how resulting sequence $K^d, (\sigma^{d-1}, K^{d-1}), \ldots, (\sigma^0, K^0)$ is encoded in computer memory. PolarSSL imposes the requirement $w \in [2\,..\,7]$ (in practice $w \in \{4, 5\}$ is used, see line 1382 of `ecp.c`) and can hence store each $K^i$ in a separate octet (data type '`unsigned char`'). The remaining eighth bit is used to store the corresponding sign indicator; precisely, $\sigma^i = +1$ and $\sigma^i = -1$ are encoded as `0` and `1`, respectively. For example, if $w = 3$ and $\sigma^i = -1$ and $K^i = (1, 0, 1)$, the in-memory representation is `10000101`.

Similarly to Fact 1 we can state a suffix property for this encoding.

**Fact 2 (Suffix property of PolarSSL's comb encoding)** *Fix a scalar $a \in \mathbb{N}$ and parameters $w, d$. Denote $a$'s binary expansion with $(a_{wd-1}, \ldots, a_0)$, its (textbook) comb encoding with $(\bar{K}^{d-1}, \ldots, \bar{K}^0)$ where $\bar{K}^i_j = a_{i+jd}$, and its PolarSSL comb encoding with $(K^d, \sigma^{d-1}, K^{d-1}, \ldots, \sigma^0, K^0)$. Then it holds for all $1 \leq t \leq d$ that $(K^{t-1}, \sigma^{t-2}, K^{t-2}, \ldots, \sigma^0, K^0)$ is fully determined by $(\bar{K}^{t-1}, \ldots, \bar{K}^0)$.*

[6] Available at `https://polarssl.org`.

## 4 General procedures for recovering noisy keys

We present next our algorithms that recover the private keys of DL-based cryptosystems from noisy memory images. Separate algorithms are proposed for OpenSSL and PolarSSL and, thus, each will have its own analysis of success probability. We start with specifying the attack model.

### 4.1 Attack model

Both in OpenSSL and PolarSSL, discrete log secret keys and their NAF or comb encodings reside in computer memory simultaneously, at least for a short period of time. Our cold boot attack model hence assumes that the adversary can obtain noisy versions of the original private key and its encoding, and aims at recovering the private key. We assume that a 0 bit will flip with probability $\alpha = \mathbb{P}(0 \to 1)$ and a 1 bit will flip with probability $\beta = \mathbb{P}(1 \to 0)$. Furthermore, we assume that the attacker knows the values of $\alpha$ and $\beta$. Such an assumption is possible because an adversary can easily estimate using an analysis similar to [8]. We refer the reader to that paper for the details.

### 4.2 NAF encodings

Algorithm 5 attempts to recover a key that has been encoded with either the textbook wNAF or the modified NAF of OpenSSL (from Algorithms 1 and 3, respectively). It takes several inputs: the public key, $Q = aP$; the noisy memory image, $\mathcal{M}^*$; the length of the private key, $\ell$; the window size, $w$; a variable parameter, $t$; a constant $k$.

**Algorithm 5** Generic key-recovery algorithm for textbook and OpenSSL wNAF.

**Input:** noisy memory image $\mathcal{M}^*$, reference public key $Q = aP$, parameters $\ell, w, t, k$; use $k = 0$ for textbook wNAF recovery, and $k > 0$ otherwise.
**Output:** secret key $a$ or $\perp$

1: $\mathcal{C}and\mathcal{S}et \leftarrow \emptyset$
2: **for all** $x \in \{0,1\}^{t+w-1}$ **do**
3: calculate partial representation $\mathcal{M}_x$ of $x$
4: **if** Correlate$(\mathcal{M}_x, \mathcal{M}^*) =$ pass **then**
5: add $x$ to $\mathcal{C}and\mathcal{S}et$
6: **for** $i \leftarrow 2$ **to** $\lfloor(\ell - k + 1 - w)/t\rfloor$ **do**
7: $\mathcal{C}and\mathcal{S}et' \leftarrow \emptyset$
8: **for all** $x \in \{0,1\}^t \times \mathcal{C}and\mathcal{S}et$ **do**
9: calculate partial representation $\mathcal{M}_x$ of $x$
10: **if** Correlate$(\mathcal{M}_x, \mathcal{M}^*) =$ pass **then**
11: add $x$ to $\mathcal{C}and\mathcal{S}et'$
12: $\mathcal{C}and\mathcal{S}et \leftarrow \mathcal{C}and\mathcal{S}et'$
13: **for all** $x \in \{0,1\}^{k+(\ell-k-w+1 \bmod t)} \times \mathcal{C}and\mathcal{S}et$ **do**
14: $a \leftarrow \sum_{i=0}^{\ell-1} 2^i x_i$
15: **if** $Q = aP$ **then**
16: **return** $a$
17: **return** $\perp$

We first discuss the textbook NAF, for which $k = 0$. The algorithm will output either $a$ (the private key) or $\perp$, which represents failure. The recovery procedure begins by initialising a set $\mathcal{C}and\mathcal{S}et$ to be empty. The set $\mathcal{C}and\mathcal{S}et$ will store (partial) candidate solutions for the private key $a$. At each stage of the algorithm we wish to compute $t$ new wNAF digits for each candidate solution. To be certain of outputting the first $t$ signed digits of the wNAF, the algorithm requires knowledge of the least $t + w - 1$ bits of the binary representation (cf. Fact 1). Hence, the first stage of the algorithm (cf. lines 1–5) takes

all bit strings of length $t + w - 1$ (giving us the ability to calculate the least $t$ signed digits of the wNAF), converts them to integers, then calculates their corresponding wNAFs for positions $b_{t-1}, \ldots, b_0$ (prepending zeros if necessary, and ignoring any $b_j$ for $j \geq t$ if they exist). The algorithm then compares each bit string and its corresponding wNAF against $\mathcal{M}^*$ via the Correlate function (see Section 2). If the candidate passes the Correlate test, then the candidate solution is added to the set $\mathcal{C}and\mathcal{S}et$, otherwise it is discarded. Once all bit strings of length $t+w-1$ have been checked, we move on to the second stage of the algorithm (cf. lines 6–12). We first initialise a set $\mathcal{C}and\mathcal{S}et'$ to be empty. For each string $x$ in $\mathcal{C}and\mathcal{S}et$, we prepend all bit strings of length $t$ to $x$ (giving us the ability to compute the next $t$ signed digits of the wNAF). We then calculate the wNAFs of (the integer conversions of) all the strings. Again, we prepend zeros to the wNAF if necessary, and we ignore any $b_j$ for $j \geq 2t$. Then the algorithm compares each bit string and its corresponding wNAF against $\mathcal{M}^*$ via the Correlate function. If the candidate solution passes the test it is added to $\mathcal{C}and\mathcal{S}et'$. When all appropriate strings have been checked, we overwrite $\mathcal{C}and\mathcal{S}et \leftarrow \mathcal{C}and\mathcal{S}et'$. If we let $\ell'$ denote the length of the partial candidates, then we repeat the previous stage of the algorithm until $\ell' > \ell - t$ (because, at this point, prepending $t$ bits to the candidate solutions would result in them having a greater length than the private key $a$). At this juncture the algorithm will prepend all bit-strings of length $\ell - \ell'$ to all the strings in $\mathcal{C}and\mathcal{S}et$ (cf. lines 13–16). Each of these new strings $x$ is then compared against the public key $Q = aP$, via the calculation $xP$. If there is a match with $Q = aP$, then the algorithm outputs $x$, otherwise the algorithm outputs $\perp$.

We will now discuss the modifications that we make for the OpenSSL implementation of the wNAF encoding. From Algorithm 3 it is clear that the OpenSSL wNAF only modifies the textbook wNAF in (at most) the most significant $w+1$ digits (excluding leading zeros). Algorithm 5 relies on the fact that textbook wNAFs can be built up in a bit-by-bit fashion from the least significant bit (cf. Fact 1), but this is no longer possible with the modified wNAF. Therefore, when dealing with the OpenSSL NAF, we include an extra parameter $k > 0$ in Algorithm 5, where $k \in \mathbb{N}^{>0}$. The only difference is that instead of entering the final stage of the algorithm when $\ell' > \ell - t$, we now enter the final stage when $\ell' > \ell - t - k$. That is, we stop $k$ bits earlier than with $k = 0$, and then the final stage appends $\ell - \ell'$ bits to each string in $\mathcal{C}and\mathcal{S}et$ and checks whether any of these new strings matches the private key, $a$. The reasoning behind this is that if the bit representation of an integer has a leading 1 in position $i$, then the standard wNAF will only be affected in positions $i+1$ to $i-w+1$. In Algorithm 5, at most we compute $\ell - k - w + 1$ signed digits for each candidate solution. For a uniformly random private key $a$, the higher we set $k$, the more likely it is that the textbook wNAF and modified wNAF of $a$ agree in the positions our algorithm computes (since a uniformly random key is more likely to have a leading 1 in bit positions $\ell - 1$ to $\ell - k - 1$, meaning the first $\ell - k - w + 1$ signed digits remain unaffected). This will be discussed in more detail in Section 4.4. However, there is a trade-off between running-time and success. A higher $k$ results in a higher success, but the last stage of Algorithm 5 appends bit-strings of at least length $k$ to all surviving candidates. Hence, the greater $k$ is, the longer the running-time of this final phase. A typical value for $k$ would be below 10.

### 4.3 Comb encodings

In this section we consider key-recovery for comb-based methods. The textbook comb encoding together with the original key represents merely a repetition code, and there are standard techniques to recover the key for such a code. Hence, we shall proceed straight to the discussion of PolarSSL combs. To prevent side-channel attacks (cf. Section 3.4), the PolarSSL comb uses a lookahead algorithm, so we will need a more sophisticated algorithm than the standard techniques used for repetition codes. The pseudocode for our algorithm can be found in Algorithm 6. The inputs are: the noisy memory image, $\mathcal{M}^*$; the public key, $Q = aP$; the length of the comb (i.e., the number of prongs), $w$; the number of comb positions, $d$; and a variable parameter $t$. To calculate component $K^0$ of the comb requires knowledge of bits $a_{(w-1)d}, a_d, \ldots, a_0$ (and only these bits). If we want to calculate $K^1$ and $\sigma^0$, we additionally need bits $a_{1+(w-1)d}, a_{1+d}, \ldots, a_1$, and so on (cf. Fact 2). Our algorithm considers $t$-many comb components at each stage. During the first stage (cf. lines 1–9) we wish to compute $K^{t-1}, (\sigma^{t-2}, K^{t-2}), \ldots, (\sigma^0, K^0)$ for each candidate solution. To calculate these components only requires knowledge of $tw$ bits (in the appropriate positions of the key). Since PolarSSL only handles odd scalars, there are $2^{tw-1}$ candidate solutions across these $tw$ bits. For each of these candidate strings, we compare the bits of the string $x$ and its comb with the noisy versions via the Correlate function. If the candidate passes the Correlate test, the string is added to $\mathcal{C}and\mathcal{S}et$ (which we initialize to empty), otherwise it is discarded. We then (cf. lines 10–20) repeat the

procedure by combining each surviving candidate with all possible bit combinations in the $tw$ positions that will allow us to compute the next $t$ comb components, which are $K^{2t-1}, (\sigma^{2t-2}, K^{2t-2}), \ldots, (\sigma^t, K^t)$. If $\ell'$ denotes the length of the current candidates, the algorithm exits this particular For loop when $dw - \ell' \leq tw$ (i.e., when adding $t$ more $\bar{K}^j$ would result in there being more $\bar{K}^j$ than exist for the private key). At this point, the algorithm fills in all the missing bits with all possible combinations (cf. lines 21–26). Then the algorithm checks whether any of the strings is a match for the private key (by using the public information $Q = aP$). If there is a match, the algorithm outputs the string, otherwise it outputs $\perp$.

**Algorithm 6** Generic key-recovery algorithm for PolarSSL comb method.

**Input:** noisy memory image $\mathcal{M}^*$, reference public key $Q = aP$, parameters $d, w, t$
**Output:** secret key $a$ or $\perp$

1: $\mathcal{C}and\mathcal{S}et \leftarrow \emptyset$
2: **for all** $x \in \{0,1\}^{tw-1} \times \{1\}$ **do**
3: **for** $j \leftarrow 0$ **to** $t-1$ **do**
4: $\bar{K}^j \leftarrow (x_{(j+1)w-1}, \ldots, x_{jw})$
5: compute $K^{t-1}, (\sigma^{t-2}, K^{t-2}), \ldots, (\sigma^0, K^0)$
6: using lines 2–10 of Algorithm 4
7: calculate partial representation $\mathcal{M}_x$
8: **if** Correlate$(\mathcal{M}_x, \mathcal{M}^*) =$ pass **then**
9: add $x$ to $\mathcal{C}and\mathcal{S}et$
10: **for** $i \leftarrow 2$ **to** $\lceil d/t \rceil - 1$ **do**
11: $\mathcal{C}and\mathcal{S}et' \leftarrow \emptyset$
12: **for all** $x \in \{0,1\}^{tw} \times \mathcal{C}and\mathcal{S}et$ **do**
13: **for** $j \leftarrow 0$ **to** $it-1$ **do**
14: $\bar{K}^j \leftarrow (x_{(j+1)w-1}, \ldots, x_{jw})$
15: compute $K^{it-1}, (\sigma^{it-2}, K^{it-2}), \ldots, (\sigma^0, K^0)$
16: using lines 2–10 of Algorithm 4
17: calculate partial representation $\mathcal{M}_x$
18: **if** Correlate$(\mathcal{M}_x, \mathcal{M}^*) =$ pass **then**
19: add $x$ to $\mathcal{C}and\mathcal{S}et'$
20: $\mathcal{C}and\mathcal{S}et \leftarrow \mathcal{C}and\mathcal{S}et'$
21: **for all** $x \in \{0,1\}^{wd-(\lceil d/t \rceil - 1)tw} \times \mathcal{C}and\mathcal{S}et$ **do**
22: **for** $j \leftarrow 0$ **to** $d-1$ **do**
23: $\bar{K}^j \leftarrow (x_{(j+1)w-1}, \ldots, x_{jw})$
24: $a \leftarrow \sum_{j=0}^{d-1} \sum_{i=0}^{w-1} 2^{j+id} \bar{K}_i^j$
25: **if** $Q = aP$ **then**
26: **return** $a$
27: **return** $\perp$

*Remark 1.* We note that in some cases there is a simple way to slightly increase the efficiency of Algorithm 6. If $\ell$ is the length of the private key, but $\ell \neq wd$, then the private key will have to be prepended with $wd - \ell$ zero bits. Algorithm 6 can be improved by utilising this information and only considering candidate solutions with zeros in these particular positions. However, as in practice $w = 4$ or $w = 5$ is used and we consider $\ell = 160$ in our simulations, there will be no need for prepended zeros and our algorithm will run exactly as presented in Algorithm 6.

*Remark 2 (Optimality of Algorithms 5 and 6).* We do not claim that Algorithms 5 or 6 are the optimal procedures for recovering keys in their respective scenarios. However, these algorithms are appealing because we are able to provide a theoretical analysis of the success rate (cf. Section 2). Furthermore, the experimental results we obtain from these algorithms are good in practice, as we shall see in the coming sections.

### 4.4 Success analysis of OpenSSL implementation

We now analyse the success probability of Algorithm 5 when combined with the Correlate test from Section 2. The success probability is relatively straightforward to calculate if the input is an image of a textbook wNAF: the correct candidate will pass the Correlate test (equation (4)) with probability $\gamma^2$, where $\gamma = \int_0^C \chi_1^2(x)\mathrm{d}x$. Hence, the probability of recovering the correct key would be $\gamma^{2\cdot\lfloor(\ell+1-w)/t\rfloor}$ because there are $\lfloor(\ell+1-w)/t\rfloor$-many Correlate tests to pass and the probability of passing each test is independent (because Correlate considers only the newly computed bits at each stage). However, since a modified NAF is used in OpenSSL, the corresponding analysis of success will differ slightly. Fortunately, the difference between textbook NAF and modified NAF is only in the most significant $w+1$ bits (and sometimes there is no difference at all): If the leading 1 bit of the discrete logarithm key is in position $i$ then, at most, only signed digits $i-w+1$ to $i+1$ of the standard wNAF will be affected by the transformation to the modified wNAF. Therefore the standard and modified wNAFs will agree up to position $i-w$. Algorithm 5 only computes the least significant $j = \ell-k-w+1-(\ell-k-w+1 \bmod t)$ digits of the wNAF, i.e., $b_{j-1},\ldots,b_0$. Therefore, we must now bound the probability that a randomly chosen private key's standard wNAF is equal to its modified NAF up to digit $b_{j-1}$. If the private key has a 1 bit anywhere between positions $j+w-1$ and $\ell-1$ then the computed NAF digits will be identical to the modified wNAF digits up to position $j-1$, and then the multinomial test will behave exactly as expected (having probability $\gamma$ of passing each test). The probability of a 1 bit appearing in any of these positions is precisely

$$1-2^{-k-(\ell-k-w+1 \bmod t)}\ .$$

If we let M-NAF denote the modified wNAF, and $\mathrm{wNAF}_{j-1}$ (resp. $\mathrm{M\text{-}NAF}_{j-1}$) denote digits 0 to $j-1$ of wNAF (resp. M-NAF), then it follows that

$$\begin{aligned}\mathbb{P}(\text{success}) &= \mathbb{P}(\text{success}|w\text{-NAF}_{j-1} = \text{M-NAF}_{j-1})\cdot\mathbb{P}(w\text{-NAF}_{j-1} = \text{M-NAF}_{j-1})\\ &\quad+\ \mathbb{P}(\text{success}|w\text{-NAF}_{j-1} \neq \text{M-NAF}_{j-1})\cdot\mathbb{P}(w\text{-NAF}_{j-1} \neq \text{M-NAF}_{j-1})\\ &\geq \left(1-2^{-k-(\ell-k-w+1 \bmod t)}\right)\cdot\gamma^{2\cdot\lfloor(\ell-k+1-w)/t\rfloor}\ .\end{aligned}$$

Thus, by setting the thresholds $k$ and $C$ (and, hence, $\gamma$) appropriately, we can achieve any desired success rate (potentially at the expense of a long running time).

If either $\alpha = 0$ or $\beta = 0$ our algorithm has a slightly different analysis. Since neither $\alpha$ nor $\beta$ will be zero in practice, we have relegated this analysis to Appendix A.

### 4.5 Success analysis of PolarSSL implementation

Given the previous discussion regarding the success of recovering keys of the NAF algorithms, it is now very easy to analyse the success of Algorithm 6. It is clear from the algorithm that there are $\lceil d/t\rceil - 1$ Correlate tests to pass. The correlate function is in equation (4), and the correct candidate has probability $\gamma^2$ of passing the test, where $\gamma = \int_0^C \chi_1^2(x)\mathrm{d}x$. Since each Correlate test only considers the newly calculated bits, the probability of passing each Correlate test is independent, so we have $\mathbb{P}(\text{success}) = \gamma^{2\cdot(\lceil d/t\rceil-1)}$.

## 5 Implemented simulations of key recovery

We present the results of some simulations of Algorithms 5 and 6 using the Correlate test from equation (4). Unless otherwise stated, we ran 100 tests for each set of parameters. The results for OpenSSL can be seen in Table 1a and those for PolarSSL in Table 1b. The values displayed in these tables are merely to support the validity of our theoretical analysis, and they do not represent the practical limits of our algorithms. However, it is clear that any algorithm attempting key recovery in the PolarSSL and OpenSSL settings will not be able to match the performance of the RSA algorithms. We discuss the reasons why in Appendix D. For each set of parameters, the table shows the predicted theoretical success of the algorithms and the success rate we achieved with our 100 simulations. Note that as the noise rate increases the success rate will slowly decline. However, for OpenSSL, the success rate for $\beta = 0.15$ was higher than for $\beta = 0.10$, despite all other parameters being the same. This is merely an outlier, which

| key length | $w$ | $\alpha$ | $\beta$ | $t$ | $C$ | $k$ | $\chi^2$ est. | prac. suc. |
|---|---|---|---|---|---|---|---|---|
| 160 | 2 | 0.001 | 0.01 | 7 | 6 | 3 | 0.51 | 0.92 |
| 160 | 2 | 0.001 | 0.05 | 10 | 3.5 | 3 | 0.15 | 0.45 |
| 160 | 2 | 0.001 | 0.10 | 10 | 3.5 | 3 | 0.15 | 0.17 |
| 160 | 2 | 0.001 | 0.15 | 10 | 3.5 | 3 | 0.15 | 0.20 |
| 160 | 2 | 0.001 | 0.20 | 14 | 2 | 3 | 0.02 | 0.07 |
| 160 | 2 | 0.001 | 0.25 | 12 | 2 | 3 | 0.01 | 0.06 |
| 160 | 2 | 0.001 | 0.30 | 12 | 2 | 3 | 0.01 | 0.04 |
| 160 | 2 | 0.001 | 0.35 | 14 | 0.75 | 3 | 0 | 0.02 |

(a) OpenSSL

| key length | $w$ | $d$ | $\alpha$ | $\beta$ | $t$ | $C$ | $\chi^2$ est. | prac. suc. |
|---|---|---|---|---|---|---|---|---|
| 160 | 4 | 40 | 0.001 | 0.01 | 2 | 7 | 0.73 | 0.81 |
| 160 | 4 | 40 | 0.001 | 0.02 | 2 | 5 | 0.38 | 0.65 |
| 160 | 4 | 40 | 0.001 | 0.03 | 2 | 4 | 0.17 | 0.60 |
| 160 | 4 | 40 | 0.001 | 0.05 | 2 | 3.5 | 0.09 | 0.58 |
| 160 | 4 | 40 | 0.001 | 0.06 | 2 | 3 | 0.04 | 0.55 |
| 160 | 4 | 40 | 0.001 | 0.07 | 2 | 3 | 0.04 | 0.52 |
| 160 | 4 | 40 | 0.001 | 0.08 | 2 | 2.5 | 0.01 | 0.37 |
| 160 | 4 | 40 | 0.001 | 0.10 | 2 | 2.5 | 0.01 | 0.08 |

(b) PolarSSL

Table 1: Results of simulations of cold boot attacks against the point multipliers of OpenSSL and PolarSSL. The theoretically estimated success probabilities, based on the convergence to the chi-squared distribution, are in the columns labelled '$\chi^2$ est'. Note that the effective success rates of our implemented attacks, in columns 'prac. suc.', are generally much larger.

is a result of the limited number of simulations we ran. If we were to perform a much larger number of simulations, we expect this outlier to disappear. All values we have used for $\alpha$ and $\beta$ are practical, but higher values of $\beta$ are much rarer in practice. For small values of $\beta$ (which are most common) our algorithms have a good success rate. For example, for OpenSSL we have a success rate of 45% when $\beta = 0.05$. Furthermore, for small values such as this we could significantly improve the success by increasing threshold $C$. For such small values of $\beta$ this would not greatly affect the running time. Note that the majority of the experiments were conducted in a $1 \to 0$ region (where $\alpha \ll \beta$). This choice will not affect the theoretical success of the algorithm, but is likely to have an impact on the running-time. In the PolarSSL setting, the difference between a $1 \to 0$ and $0 \to 1$ region will be very small, due to the approximately equal distribution of 1 and 0 bits in the private key. However, for OpenSSL there will be a noticeable discrepancy, which is explored further in Appendix C.

## 6 Conclusions

We propose key-recovery algorithms for various discrete logarithm cryptosystems, with particular emphasis on the widely deployed PolarSSL and OpenSSL implementawtions. These algorithms represent a large improvement over previous key-recovery algorithms for discrete-log cold boot attacks. We provide a theoretical analysis that lower-bounds the success of our algorithms. Furthermore, the statistical test we use in our framework provides an avenue to obtain arbitrary success rates in the RSA setting when the errors are asymmetric. Such results were only previously available in the symmetric setting. We provide results of several key-recovery simulations, both for PolarSSL and OpenSSL, that fully support our theoretical analyses and show that our attacks are practical.

## Acknowledgments

The authors were supported by EPSRC Leadership Fellowship EP/H005455/1. The first author received additional support from a Sofja Kovalevskaja Award of the Alexander von Humboldt Foundation, and the German Federal Ministry for Education and Research.

## References

1. Martin Albrecht and Carlos Cid. Cold boot key recovery by solving polynomial systems with noise. In Javier Lopez and Gene Tsudik, editors, *ACNS 11*, volume 6715 of *LNCS*, pages 57–72, Nerja, Spain, June 7–10, 2011. Springer, Berlin, Germany.
2. Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Selecting elliptic curves for cryptography: An efficiency and security analysis. Cryptology ePrint Archive, Report 2014/130, 2014. http://eprint.iacr.org/2014/130.

3. Peter Gutmann. Data remanence in semiconductor devices. In *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*. USENIX, 2001.
4. J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.
5. Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
6. Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau. A comb method to render ECC resistant against side channel attacks. Cryptology ePrint Archive, Report 2004/342, 2004. http://eprint.iacr.org/2004/342.
7. Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau. Countermeasures for preventing comb method against SCA attacks. In Robert H. Deng, Feng Bao, HweeHwa Pang, and Jianying Zhou, editors, *Information Security Practice and Experience, First International Conference, ISPEC 2005, Singapore, April 11-14, 2005, Proceedings*, volume 3439 of *LNCS*, pages 85–96. Springer, 2005.
8. Wilko Henecka, Alexander May, and Alexander Meurer. Correcting errors in RSA private keys. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 351–369, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Berlin, Germany.
9. Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 1–17, Santa Barbara, CA, USA, August 16–20, 2009. Springer, Berlin, Germany.
10. Jakob Jonsson and Burton S. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1 (RFC 3447), 2003. https://www.ietf.org/rfc/rfc3447.txt.
11. Abdel Alim Kamal and Amr M. Youssef. Applications of SAT solvers to AES key recovery from decayed key schedule images. Cryptology ePrint Archive, Report 2010/324, 2010. http://eprint.iacr.org/2010/324.
12. Noboru Kunihiro and Junya Honda. RSA meets DPA: Recovering RSA secret keys from noisy analog data. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 261–278, Busan, South Korea, September 23–26, 2014. Springer, Berlin, Germany.
13. Noboru Kunihiro and Junya Honda. RSA meets DPA: Recovering RSA secret keys from noisy analog data. Cryptology ePrint Archive, Report 2014/513, 2014. http://eprint.iacr.org/2014/513.
14. Noboru Kunihiro, Naoyuki Shinohara, and Tetsuya Izu. Recovering RSA secret keys from noisy key bits with erasures and errors. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 180–197, Nara, Japan, February 26 – March 1, 2013. Springer, Berlin, Germany.
15. Hyung Tae Lee, HongTae Kim, Yoo-Jin Baek, and Jung Hee Cheon. Correcting errors in private keys obtained from cold boot attacks. In Howon Kim, editor, *ICISC 11*, volume 7259 of *LNCS*, pages 74–87, Seoul, Korea, November 30 – December 2, 2011. Springer, Berlin, Germany.
16. Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 95–107, Santa Barbara, CA, USA, August 21–25, 1994. Springer, Berlin, Germany.
17. Bodo Möller. Improved techniques for fast exponentiation. In Pil Joong Lee and Chae Hoon Lim, editors, *ICISC 02*, volume 2587 of *LNCS*, pages 298–312, Seoul, Korea, November 28–29, 2002. Springer, Berlin, Germany.
18. Kenneth G. Paterson, Antigoni Polychroniadou, and Dale L. Sibborn. A coding-theoretic approach to recovering noisy RSA keys. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 386–403, Beijing, China, December 2–6, 2012. Springer, Berlin, Germany.
19. Santanu Sarkar, Sourav Sen Gupta, and Subhamoy Maitra. Reconstruction and error correction of RSA secret parameters from the MSB side. In *WCC 2011 - Workshop on coding and cryptography*, pages 7–16, Paris, France, April 2011.
20. Santanu Sarkar and Subhamoy Maitra. Side channel attack to actual cryptanalysis: Breaking CRT-RSA with low weight decryption exponents. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 476–493, Leuven, Belgium, September 9–12, 2012. Springer, Berlin, Germany.
21. L.Z. Scheick, S.M. Guertin, and G.M. Swift. Analysis of radiation effects on individual DRAM cells. *Nuclear Science, IEEE Transactions on*, 47(6):2534–2538, Dec 2000.
22. Paul J. Smith, Donald S. Rae, Ronald W. Manderscheid, and Sam Silbergeld. Approximating the moments and distribution of the likelihood ratio statistic for multinomial goodness of fit. *Journal of the American Statistical Association*, 76(375):pp. 737–740, 1981.
23. Alex Tsow. An improved recovery algorithm for decayed AES key schedule images. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *SAC 2009*, volume 5867 of *LNCS*, pages 215–230, Calgary, Alberta, Canada, August 13–14, 2009. Springer, Berlin, Germany.
24. D. A. Williams. Improved likelihood ratio tests for complete contingency tables. *Biometrika*, 63(1):pp. 33–37, 1976.

## A Analysis of success when $\alpha$ or $\beta$ is zero

Throughout this paper we have assumed that both $\alpha := \mathbb{P}(0 \to 1)$ and $\beta := \mathbb{P}(1 \to 0)$ are non-zero (as they are likely to be in practice), and the analyses of Sections 4.4 and 4.5 were dependent on this fact. Here we briefly discuss how to handle the case when either $\alpha$ or $\beta$ is zero (if both are zero, then no bits will flip, hence the noisy key will be identical to the original key). We will first discuss the OpenSSL case (cf. Section 4.2). We assume that $\alpha = 0$ (the case $\beta = 0$ is corresponding), so there will be no 0 to 1 bit flips from the original key to the noisy key. We cannot perform a multinomial test on the 0s of the candidate solutions (because this requires non-zero $\alpha$), so instead we merely discard any candidate solution in which a 0 must have flipped to a 1. Notice that it is impossible to reject the correct solution via this test (because $\mathbb{P}(n_{01} = 0) = 1$ for the correct candidate). The Correlate test is then

$$\text{Correlate}_C(s_i, r) = \text{pass} \quad \Leftrightarrow \quad \text{Correlate}^1(s_i, r) < C \wedge n_{01}^i = 0 \ . \tag{6}$$

The theoretical success rate of Algorithm 5 with $k = 0$ is $\gamma^{\lfloor(\ell+1-w)/t\rfloor}$, and its success rate is at least $\left(1 - 2^{-k-(\ell-k-w+1 \bmod t)}\right) \cdot \gamma^{\lfloor(\ell-k+1-w)/t\rfloor}$ if $k > 0$, where $\gamma$ is the probability of passing a single multinomial test. If instead we have $\beta = 0$, then

$$\text{Correlate}_C(s_i, r) = \text{pass} \quad \Leftrightarrow \quad \text{Correlate}^0(s_i, r) < C \wedge n_{10}^i = 0 \ . \tag{7}$$

The success of the PolarSSL algorithm (cf. Section 4.3) now follows easily. The Correlate functions are those in equations (6) and (7) (for $\alpha = 0$ and $\beta = 0$ respectively) and the success rate is estimated with $\gamma^{\lceil d/t\rceil - 1}$.

## B Running-time analysis

So far we have provided a theoretical analysis for the success of our key-recovery procedures from Section 4. To complete the picture, we would also like an analysis of the running-time of the algorithms. Unfortunately, such an analysis appears very difficult to obtain in our setting. If we were able to bound the probability of a 'Type II error' (not rejecting an incorrect solution) in the multinomial test, this would allow us to bound the running-time. In the RSA setting, a typical assumption is that incorrect candidate solutions are uniformly random and independent of all previous key bits[7]. In the two scenarios we consider, such assumptions clearly do not hold. The bits of a particular candidate wNAF or comb solution are entirely dependent on *all* the previous key bits. Given that we cannot employ any independence assumptions, we are unable to provide a theoretical analysis of the running-time of our algorithms.

Notice, however, that the Correlate function can be modified to produce a test that has a bounded running-time. We have previously suggested setting a threshold and discarding any solutions that do not pass it. This meant that the algorithm would output lists of a variable size. Instead, we can modify the Correlate function to output shorter (or even fixed-sized) lists, thereby allowing the running-time to be easily bounded. The Correlate test could be modified so that it computes $\text{Correlate}^0$ and $\text{Correlate}^1$ (equations (2) and (3) respectively), and then sums these two values. Then the algorithm would output the $L$-many candidates with the lowest values of $\text{Correlate}^0 + \text{Correlate}^1$, where $L$ is a parameter chosen by the attacker. This algorithm clearly has a bounded running-time, but a theoretical analysis of success rate is lacking for this particular approach.

## C Comparison of ground states

In this section we study the impact of the ground states on our cold boot recovery algorithms. Recall that there are two ground states: 0 and 1. In a 0 region, 1 bits have a reasonably high chance of flipping to a 0, but 0 bits will have a very low probability (typically 0.001) of flipping to a 1. In a 1 region the opposite is true. In the RSA cold boot analyses of [8,9,18] it was assumed that secret keys consist of approximately an equal number of 0 and 1 bits. Evidently then, the recovery algorithms would run equally well in both types of region. In contrast, in an OpenSSL wNAF there are significantly more

[7] The analyses of [8,9] used such an assumption, whereas [18] made use of a slightly weaker assumption.

| $\alpha$ | $\beta$ | $C$ | suc. | min. | LQ | med. | UQ | max. |
|---|---|---|---|---|---|---|---|---|
| 0.001 | 0.01 | 4 | 0.338 | 1 | 2 | 2 | 4 | 120 |
| 0.01 | 0.001 | 4 | 0.524 | 1 | 48 | 144 | 483 | 82944 |
| 0.01 | 0.001 | 2.5 | 0.195 | 2 | 16 | 48 | 144 | 2880 |

Table 2: Quartile data for the number of candidate solutions that passed the final Correlate test of Algorithm 5 (i.e., the size of $\mathcal{C}and\mathcal{S}et$ at line 13). For each set of parameters we ran 1000 tests with 160-bit keys, and we set $t = 7$.

0 bits than 1 bits. As a result our algorithm may have different performance results depending on the decay region. The theoretical success is obviously unaffected, but the running-time varies significantly, as Table 2 shows. Note that results were implemented with the textbook NAF (i.e., $k = 0$ in Algorithm 5), rather than the modified version.

For all tests we set the key size to be 160 bits and we conducted 1000 tests for each set of parameters. We chose to run 1000 tests in order to eradicate any statistical anomalies that might arise from using small sample sizes. For the first two sets of parameters we set $w$ to be 2, the threshold $C$ was 4, and $t$ was 7. For the first set of parameters, we set $\alpha = \mathbb{P}(0 \to 1) = 0.001$ and $\beta = \mathbb{P}(1 \to 0) = 0.01$, to represent a 1-to-0 region. For the second set, we reversed these values, so $\alpha = \mathbb{P}(0 \to 1) = 0.01$ and $\beta = \mathbb{P}(1 \to 0) = 0.001$, which represents a 0-to-1 region. For each test in which we successfully recovered the private key we kept a record of the number of solutions that passed the final Correlate function test. Table 2 displays the quartiles of this data (the minimum, lower quartile, median, upper quartile and maximum). It is clear to see from the table that the algorithm had to consider many more solutions in the 0-to-1 region, which results in a much greater running time. However, this could be partially explained by the much higher success rate in the 0-to-1 region. In the 1-to-0 region, the success rate was 0.338, compared to 0.524 in the 0-to-1 region. This elevated success rate will obviously result in more candidates being checked, but this is not the only reason. The convergence of the multinomial test to the chi-squared distribution is dependent on the expected values of $n_{00}$ and $n_{01}$ (the higher they are, the better the convergence), where we recall that $n_{ij}$ is the number of $i$ bits in the candidate solution that map to $j$ bits in the noisy information. By changing the values of $\alpha$ and $\beta$, we change the expected values of $n_{ij}$, which results in varying success probabilities, despite having the same threshold. To counteract this problem, we ran another set of experiments. For $\alpha = 0.01$ and $\beta = 0.001$ we re-ran the simulations, but with the threshold now set to $C = 2.5$. The success probability for this new set of tests was 0.195, which is much lower than the success for the 1-to-0 region. Despite this, however, the quartiles were still much higher for the 0-to-1 region. The explanation for this phenomenon appears to be simple. In a wNAF, the 1 bits are sparse. In a 1-to-0 region, if we observe a 1 bit in the noisy information then, with high probability, the private key has a 1 bit in that particular position. Since 1 bits are infrequent, there will be very few candidate solutions that have a 1 bits in the necessary positions. Conversely, in a 0-to-1 region, if we observe a 0 bit in the noisy information then, with high probability, the private key has a 0 bit in the corresponding position. Unfortunately, since 0 bits are abundant in a wNAF, there are typically many candidates that have 0 bits in these positions. Hence, whilst the success is unaffected by the ground state, the running-time will be much greater in a 0-to-1 region because there will be many more incorrect candidates that pass the Correlate test.

For the PolarSSL comb the distribution of bits is not uniform (since some bits are always set to be 1), which will result in slightly different performances in the two regions. We do not expect the running-time to vary considerably, but for particularly small values of $\alpha$ and $\beta$ we might see a slight difference in the success rates of simulations.

## D Comparison with the RSA setting

At first glance our experimental results for both OpenSSL's wNAF and PolarSSL's comb multiplier (Tables 1a and 1b) appear to be inferior to corresponding results in the RSA setting. Whilst this is true, it should not come as a surprise. Analyses in the RSA setting (such as those of [8,9,18]) enjoyed several benefits. The major advantage they had was the relationship between key bits via equations such as $N = pq$, where $N$ is the public exponent, and $p$ and $q$ are the private primes. There are four such

equations relating the five components of the RSA private key. Heninger *et al.* [9] showed that if there is a partial solution for the private RSA key and an adversary wishes to discover the next bit of each of the five private key components, then the RSA equations give only two possible solutions for the string of five bits, rather than the 32 solutions that would need to be checked in a brute-force search. Hence, when the RSA algorithms calculate possible solutions across a new set of $5t$ bits, if there are $M$ surviving candidates from the previous pruning phase, there will be $M \cdot 2^t$ possible solutions to check at the next stage. These solutions may then be tested to discard unlikely candidates. In the NAF and comb settings, such strong structure in the private key does not exist. Hence, when we consider solutions in a string of $5t$ bits, if $M$ candidates pass the previous pruning phase, then we have to consider $M \cdot 2^{5t}$ solutions, and then discard unlikely candidates. Furthermore, when the RSA algorithms calculate the two possible solutions for a particular set of five bits, the two solutions have a Hamming distance of four. Consequently, if the correct key has high likelihood of coming from the noisy information (as expected), then the second possible solution will have a low likelihood. This allows the RSA algorithms to easily discard incorrect candidates with high probability. Unfortunately, in our settings we will have to consider many solutions that have Hamming distance less than four from the correct solution. The solutions with low Hamming distance from the correct key will have a high probability of passing the threshold test. With these facts in mind, it is quite clear that any algorithm in the NAF or comb settings will not be able to compete with the RSA algorithms in terms of the cross-over probabilities that can be handled for the asymmetric channel.