Appears in Proceedings of the 27th USENIX Security Symposium,
Baltimore, MD, August 2018

One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA

Monjur M. Alam *Georgia Tech*
Haider A. Khan *Georgia Tech*
Moumita Dey *Georgia Tech*
Nishith Sinha *Georgia Tech*
Robert Callan *Georgia Tech*
Alenka Zajic *Georgia Tech*
Milos Prvulovic *Georgia Tech*

Abstract

This paper presents the first side channel attack approach that, without relying on the cache organization and/or timing, retrieves the secret exponent from a single decryption on arbitrary ciphertext in a modern (current version of OpenSSL) fixed-window constant-time implementation of RSA. Specifically, the attack recovers the exponent's bits during modular exponentiation from analog signals that are unintentionally produced by the processor as it executes the constant-time code that constructs the value of each "window" in the exponent, rather than the signals that correspond to squaring/multiplication operations and/or cache behavior during multiplicand table lookup operations. The approach is demonstrated using electromagnetic (EM) emanations on two mobile phones and an embedded system, and after only one decryption in a fixed-window RSA implementation it recovers enough bits of the secret exponents to enable very efficient (within seconds) reconstruction of the full private RSA key.

Since the value of the ciphertext is irrelevant to our attack, the attack succeeds even when the ciphertext is unknown and/or when message randomization (blinding) is used. Our evaluation uses signals obtained by demodulating the signal from a relatively narrow band (40 MHz) around the processor's clock frequency (around 1GHz), which is within the capabilities of compact sub-$1,000 software-defined radio (SDR) receivers.

Finally, we propose a mitigation where the bits of the exponent are only obtained from an exponent in integer-sized groups (tens of bits) rather than obtaining them one bit at a time. This mitigation is effective because it forces the attacker to attempt recovery of tens of bits from a single brief snippet of signal, rather than having a separate signal snippet for each individual bit. This mitigation has been submitted to OpenSSL and was merged into its master source code branch prior to the publication of this paper.

1 Introduction

Side channel attacks extract sensitive information, such as cryptographic keys, from signals created by electronic activity within computing devices as they carry out computation. These signals include electromagnetic emanations created by current flows within the device's computational and power-delivery circuitry [2, 3, 14, 21, 34, 47], variation in power consumption [9, 12, 15, 17, 24, 28, 35, 36, 37, 42], and also sound [6, 16, 26, 43], temperature [13, 31], and chasis potential variation [25] that can mostly be attributed to variation in power consumption and its interaction with the system's power delivery circuitry. Finally, not all side channel attacks use analog signals: some use faults [11, 27], caches [8, 44, 45], branch predictors [1], etc.

Most of the research on physical side-channel attacks has focused on relatively simple devices, such as smartcards and simple embedded systems, where the side-channel signals can be acquired with bandwidth much higher than the clock rates of the target processor and other relevant circuitry (e.g. hardware accelerators for encryption/decryption), and usually with highly intrusive access to the device, e.g. with small probes placed directly onto the chip's package [19, 36]. Recently, attacks on higher-clock-rate devices, such as smartphones and PCs , have been demonstrated [7, 20, 21, 23]. They have shown that physical side channel attacks are possible even when signals are acquired with bandwidth that is much lower than the (gigahertz-range) clock rates of the processor, with less-intrusive access to the device, and even though advanced performance-oriented features, such as super-scalar (multiple instructions per cycle) execution and instruction scheduling, and system software activity, such as interrupts and multiprocessing, cause significant variation in both shape and timing of the signal produced during cryptographic activity.

To overcome the problem of low bandwidth and variation, successful attacks on high-clock-rate systems tend

to focus on parts of the signal that correspond to activity that takes many processor cycles. A representative example of this is decryption in RSA, which consists of modular exponentiation of the ciphertext with an exponent that is derived from the private key. The attacker's goal is to recover enough bits of that secret exponent through side-channel analysis, and use that information to compute the remaining parts of the secret key. Most of the computational activity in large-integer modular exponentiation is devoted to multiplication and squaring operations, where each squaring (or multiplication) operation operates on large integers and thus takes many processor cycles.

Prior physical side-channel attacks on RSA rely on classifying the signals that correspond to large-integer square and multiply operations that together represent the vast majority of the computational work when performing large-integer exponentiation [10, 20, 25, 26]. Between these long-lasting square and multiply operations are the few processor instructions that are needed to obtain the next bit (or group of bits) of the secret exponent and use that to select whether the next large-integer operation will be squaring or multiplication, and/or which operands to supply to that operation. The focus on long-lasting operations is understandable, given that side channel attacks ultimately recover information by identifying the relevant sub-sequences of signal samples and assessing which of the possible categories is the best match for each sub-sequence. The sub-sequences that correspond to large-integer operations produce long sub-sequences of samples, so they 1) are easier to identify in the overall sequence of samples that corresponds to the entire exponentiation, and 2) provide enough signal samples for successful classification even when using relatively low sampling rates.

However, the operands in these large-integer operations are each very regular in terms of the sequence of instructions they perform, and the operands used in those instructions are ciphertext-dependent, so classification of signals according to exponent-related properties is difficult unless 1) the sequence of square and multiply operations is key-dependent or 2) the attacker can control the ciphertext that will be exponentiated, and chooses the ciphertext in a way that produces systematically different side channel signals for each of the possible exponent-dependent choices of operands.

1.1 Our Contributions

In this paper we present a side-channel attack that is based on analysis of signals that correspond to the brief computation activity that computes the value of each window during exponentiation, i.e. activity *between* large-integer multiplications, in contrast to most prior work that focuses on the large-integer multiplications themselves and/or the table lookups that obtain the multiplicand for the computed window value. The short duration of these window value computations may hinder signal-based classification to some extent. However, the values these computations operate on are related to the individual bits of the secret exponent and not the message (ciphertext). This absence of message-induced variation allows the small variation caused by different values of an individual exponent bit to "stand out" in the signal and be accurately matched to signals from training. More importantly, this message-independence makes the new attack completely immune to existing countermeasures that focus on thwarting chosen-ciphertext attacks and/or square/multiply sequence analysis.

The experimental evaluation of our attack approach was performed on two Android-based mobile phones and an embedded system board, all with ARM processors operating at high (800 MHz to 1.1 GHz) frequencies, and the signal is acquired in the 40 MHz band around the clock frequency, resulting in a sample rate that is $<5\%$ of the processor's clock frequency, and well within the signal capture capabilities of compact commercially available sub-$1,000 software-defined radio (SDR) receivers such as the Ettus B200-mini. The RSA implementation we target is the constant-time fixed-window implementation used in OpenSSL [39] version 1.1.0g, the latest version of OpenSSL at the time this paper was written. Our results show that our attack approach correctly recovers between 95.7% and 99.6% (depending on the target system) of the secret exponents' bits from the signal that corresponds to *a single instance of RSA decryption*, and we further verify that the information from each instance of RSA encryption/signing in our experiments was sufficient to quickly (on average <1 second of execution time) fully reconstruct the private RSA key that was used.

To further evaluate our attack approach, we apply it to a sliding-window implementation of modular exponentiation in OpenSSL – this was the default implementation in OpenSSL until Percival et al. [40] demonstrated that its key-dependent square/multiply sequence makes it vulnerable to side channel attacks. We show that in this implementation our approach also recovers nearly all of the secret-exponent bits from a single use (exponentiation) of that secret exponent.

To mitigate the side-channel vulnerability exposed by our attack approach, we change the window value computation to obtain a full integer's worth of bits from the exponent, then mask that value to obtain the window value, rather than constructing the window value one bit at a time with large-number Montgomery multiplication between these one-bit window-value updates. This mitigation causes the signal variation during the brief window computation to depend on tens of bits of the expo-

nent as a group, i.e. the signal variation introduced by one bit in the exponent during the window computation is now superimposed to the variation introduced by the other bits in the group, instead of having each bit's variation alone in its own signal snippet. Our experiments show that this mitigation actually improves exponentiation performance slightly and, more importantly, that with this mitigation the recovery rate for the exponents bits becomes equivalent to random guessing. This mitigation has been submitted to OpenSSL and was merged into its master source code branch on May 30th, 2018, prior to the publication of this paper.

1.2 Threat Model

1.2.1 Assumptions

Our attack model assumes that there is an adversary who wishes to obtain the secret key used for RSA-based public-key encryption or authentication. We further assume that the adversary can bring a relatively compact receiver into close proximity of the system performing these RSA secret-key operation, for example a smart-infrastructure or smart-city device which uses public key infrastructure (PKI) to authenticate itself and secure its communication over the Internet, and which is located in a public location, or that the adversary can hide a relatively compact receiver in a location where systems can be placed in close proximity to it, e.g. under a cellphone charging station at a public location, under the tabletop surface in a coffee shop, etc.).

We assume that the adversary can access another device of the same type as the one being attacked, which is a highly realistic assumption in most attack scenarios described above, and perform RSA decryption/authentication with known keys in preparation for the attack. Unlike many prior attacks on RSA, we do **not** assume that the adversary can choose (or even know) the message (ciphertext for RSA decryption) to which the private key will be applied, and we further assume that the RSA implementation under attack **does utilize blinding** to prevent such chosen-ciphertext attacks. Finally, we assume that it is highly desirable for the attacker to recover the secret key after only very few uses (ideally only one use) of that key on the target device. This is a very realistic assumption because PKI is typically used only to set up a secure connection, typically to establish the authenticity of the communication parties and establish a symmetric-encryption session key, so in scenarios where the attacker's receiver can only be in close proximity to the target device for a limited time, very few uses of the private RSA key may be observed.

1.2.2 Targeted Software

The software we target is OpenSSL version 1.1.0g [39], the latest version of OpenSSL at the time this paper was written. Its RSA decryption uses constant-time fixed-window large-number modular exponentiation to mitigate both timing-based attacks and attacks that exploit the exponent-dependent variation in the square-multiply sequence. The lookup tables used to update the result at the end of each window are stored in scattered form to mitigate attacks that examine the cache and memory behavior when reading these tables, and the RSA implementation supports blinding (which we turn on in our experiments) to mitigate chosen-ciphertext attacks.

1.2.3 Targeted Hardware

The hardware we target are two modern Android-based smartphones and a Linux-based embedded system board, all with ARM processor clocked at frequencies around 1GHz. In our experiments we place probes very close, but without physical contact with the (unopened) case of the phone, while for the embedded system board we position the probes 20 cm away from the board, so we consider the demonstrated attacks close-proximity but non-intrusive.

1.2.4 Current Status of Mitigation

The mitigation described in this paper has been submitted as a patch for integration into the main branch of OpenSSL. This patch was merged into the "master" branch of OpenSSL's source code on May 20th, 2018, before this paper was published.

2 Background

Long-lasting operations (such as large-integer square and multiply operations) facilitate matching by producing numerous signals samples even when the signal is collected at a limited sample rate.

A representative example is RSA's decryption, which at its core performs modular exponentiation of the ciphertext c with a secret exponent (d) modulo m or, in more a efficient implementation that rely on the Chinese Reminder Theorem (CRT), two such exponentiations, with secret exponents d_p and d_q with modulo p and q, respectively. The side-channel analysis thus seeks to recover either d or, in CRT-based implementations, d_p and d_q, using side-channel measurements obtained while exponentiation is performed.

The exponentiation is implemented as either left-to-right (starting with the most significant bits) or right-to-left (starting with the least significant bits) traversal

```
1  // result r starts out as 1
2  BN_one(r);
3  // For each bit of exponent d
4  for (b=bits-1;b>=0;b--){
5     // r = r*r mod m
6     BN_mod_mul(r,r,r,m);
7     if(BN_is_bit_set(d,b))
8        // r = r*c mod m
9        BN_mod_mul(r,r,c,m);
10 }
```

Figure 1: A simple implementation of large-number modular exponentiation

of the bits of the exponents, using large-integer modular multiplication to update the result until the full exponentiation is complete. Left-to-right implementations are more common, and without loss of generality we use c to denote the ciphertext, d for the secret exponent, and m for the modulus. A simple implementation of exponentiation considers one exponent bit at a time, as shown in Figure 1, which is adapted from OpenSSL's source code.

The `BN` prefix in Figure 1 stands for "Big Number" (i.e. large integer). Each large integer is represented by a vector of *limbs*, where a limb is an ordinary (machine-word-sized) integers. The `BN_is_bit_set(d,b)` function returns the value (0 or 1) of the `b`-th bit of large-integer exponent `d`, which only requires a few processor instructions: compute the index of the array element that contains the requested bit, load that element, then shift and bit-mask to keep only the requested bit. The instructions that implement the loop, the if statement, and function call/return are also relatively few in number.

However, the `BN_mod_mul` operation is much more time-consuming: it requires numerous multiplication instructions that operate on the limbs of the large-integer multiplicands. Large integers c, d, and m (or, in CRT-based implementations the d_q, d_p and the corresponding moduli), all have $O(n)$ bits and thus $O(n)$ limbs, where n is the size of the RSA cryptographic key. A grade-school implementation of `BN_mod_mul` thus requires $O(n^2)$ limb multiplications, but the Karatsuba multiplication algorithm [32] is typically used to reduces this to $O(n^{log_2 3}) \approx O(n^{1.585})$, In most modern implementations a significant further performance improvement is achieved by converting the ciphertext to a Montgomery representation, using Montgomery multiplication for `BN_mod_mul` during exponentiation, and at the end converting the result r back to the standard representation.

Even with Montgomery multiplication, however, the vast majority of execution time for large-number exponentiation is spent on large-number multiplications, so performance optimizations focus on reducing the number of these multiplications. Likewise, most of the side-channel measurements (e.g. signal samples) collected during large-number exponentiation correspond to large-number multiplication activity, so existing side channel cryptanalysis approaches tend to target multiplication activity.

One class of attacks focuses on distinguishing between squaring ($r*r$) and multiplication ($r*c$) operations, and recovering information about the secret exponent from the sequence in which they occur. Examples of such attacks include FLUSH+RELOAD [46] (which uses instruction cache behavior) and Percival's attack [40], which uses data cache behavior. In the naive implementation above, an occurrence of squaring tells the attacker that the next bit of the exponent is being used, and an occurrence of multiplication indicates that the value of that bit is 1, so an attack that correctly recovers the square-multiply sequence can trivially obtain all bits of the secret exponent.

To improve performance, most modern implementations use window-based exponentiation, where squaring is needed for each bit of the exponent, but a multiplication is needed only once per a multi-bit group (called a *window*) of exponent bits. A left-to-right (starting at the most significant bit) *sliding-window* implementation scans the exponent bits and forms windows of varying length. Since a window that contains only zero bits requires no multiplication (and thus cannot benefit from forming multi-bit windows), only windows that begin and end with 1-valued bits are allowed to form multi-bit windows, whereas zero bits in-between these windows are each treated as their own single-bit windows that can omit multiplication. A sliding-window implementation is shown in Figure 2, using code adapted from OpenSSL's source code for sliding-window modular exponentiation. The sliding-window approach chooses a maximum size *wmax* for the windows it will use, precomputes a table *ct* that contains the large-integer value $c^{wval} mod m$ for each possible value *wval* up to *wmax* length, and then scans the exponent, forming windows and updating the result for each window.

In this algorithm, a squaring (lines 7 and 26 in Figure 2) is performed for each bit while the multiplication operation (line 29) is performed only at the (1-valued) LSB of a non-zero window. Thus the square-multiply sequence reveals where some of the 1-valued bits in the exponent are, and additional bits of the exponent have been shown to be recoverable [10] by analyzing the number of squaring between each pair of multiplications. The fraction of bits that can be recovered from the square-multiply sequence depends on the maximum window size *wmax*, but commonly used values of *wmax* are relatively small and prior work [10] has experimentally demonstrated recovery of 49% of the exponent's bits on average when $wmax = 4$ based on the square-multiply sequence. Additional techniques [10, 30] have been shown

```
1  BN_one(r);
2  wstart=bits-1;
3  while(wstart>=0){
4    if(!BN_is_bit_set(d,wstart)){
5      // Window is 0, square and
6      // begin a new window
7      BN_mod_mul(r,r,r,m);
8      wstart--;
9      continue;
10   }
11   wval=1;
12   w=1;
13   // Scan up to max window length
14   for(i=1;i<wmax;i++){
15     // Don't go below exponent's LSB
16     if(wstart-i<0)
17       break;
18     // If 1 extend window to it
19     if(BN_is_bit_set(d,wstart-i)){
20       wval=(wval<<(i-w+1))+1;
21       w=i;
22     }
23   }
24   // Square result w times
25   for(i=0;i<w;i++)
26     BN_mod_mul(r,r,r,m);
27   // Multiply window's result
28   // into overall result
29   BN_mod_mul(r,r,ct[wval>>1],m);
30   // Begin a new window
31   wstart-=w;
32 }
```

Figure 2: Sliding-window implementation of large-number modular exponentiation

to recover the full RSA private key once enough of the exponent bits are known, and for $wmax = 4$ this has allowed full key recovery for 28% of the keys [10].

Concerns about the exponent-dependent square-multiply sequences have led to adoption of *fixed window* exponentiation in OpenSSL, which combines the performance advantages of window-based implementation with an exponent-independent square-multiply sequence. This implementation is represented in Figure 3, again adapted from OpenSSL's source code.

All windows now have the same number of bits `w`, with exactly one multiplication performed for each window. Note that the window value (which consists of the bits from the secret exponent) directly determines which elements of `ct` are accessed. These elements are each a large integers, each of which is typically stored as an array or ordinary integers (e.g. OpenSSL's "Big Number" BN structure). Since each such array is much larger than a cache block, different large integers occupy distinct cache blocks, and thus the address the cache set that is accessed when reading the elements of the `ct` array

```
1  b=bits-1;
2  while(b>=0){
3      wval=0;
4      // Scan the window,
5      // squaring the result as we go
6      for(i=0;i<w;i++){
7          BN_mod_mul(r,r,r,m);
8          wval<<=1;
9          wval+=BN_is_bit_set(d,b);
10         b--;
11     }
12     // Mulitply window's result
13     // into the overall result
14     BN_mod_mul(r,r,ct[wval],m);
15 }
```

Figure 3: Fixed-window implementation of large-number modular exponentiation

reveals key material. Percival's attack [40], for example, can note the sequence in which the cache sets are accessed by the victim during fixed-window exponentiation, which reveals which window values were used and in what sequence, which in turns yields the bits of the secret exponent. To mitigate such attacks, the implementation in OpenSSL has been changed to store *ct* such that each of the cache blocks it contains parts from a number of *ct* elements, and therefore the sequence of memory blocks that are accessed in each `ct[wval]` lookup leak none or very few bits of that lookup's *wval*.

Another broad class of side channel attacks relies on choosing the ciphertext such that the side-channel behavior of the modular multiplication reveals which of the possible multiplicands is being used. For example, Genkin et al. [25, 26] construct a ciphertext that produces many zero limbs in any value produced by multiplication with the ciphertext, but when squaring such a many-zero-limbed value the result has fewer zero limbs, resulting in an easily-distinguishable side channel signals whenever a squaring operation (`BN_mod_mul(r,r,r,m)` in our examples) immediately follows a 1-valued window (i.e. when r is equal to $r_{prev} * c \bmod m$). This approach has been extended [22] to construct a (chosen) ciphertext that reveals when a particular window value is used in multiplication in a windowed implementation, allowing full recovery of the exponent by collecting signals that correspond to 2^w chosen ciphertexts (one for each window value).

Chosen-ciphertext attacks can be prevented in the current implementation of OpenSSL by enabling *blinding*, which combines the ciphertext with an encrypted (using the public key) random "ciphertext", performs secret-exponent modular exponentiation on this *blinded* version of the ciphertext, and then "unblinding" the decrypted result.

Overall, because large-integer multiplication is where large-integer exponentiation spends most of its time, most of the side-channel measurements (e.g. signal samples for physical side channels) also correspond to this multiplication activity and thus both attacks and mitigation tend to focus on that part of the signal, leaving the (comparably brief) parts of the signal in-between the multiplications largely unexploited by attacks but also unprotected by countermeasures. The next section describes our new attack approach that targets the signal that corresponds to computing the value of the window, i.e .the signal *between* the multiplications.

3 Proposed Attack Method

In both fixed- and sliding-window implementations, our attack approach focuses on the relatively brief periods of computation that considers each bit of the exponent and forms the window value *wval*. The attack approach has three key components that we will discuss as follows. First, Section 3.1 describes how the signal is received and pre-processed. Second, Section 3.2 describes how we identify the point in the signal's timeline where each interval of interest begins. Finally, we describe how the bits of the secret exponent are recovered from these signal snippets for fixed-window (Section 3.3) and sliding-window (Section 3.4) implementations.

3.1 Receiving the Signal

The computation we target is brief and the different values of exponent bits produce relatively small variation in the side-channel signal, so the signals subjected to our analysis need to have sufficient bandwidth and signal-to-noise ratio for our analysis to succeed. To maximize the signal-to-noise ratio while minimizing intrusion, we position EM probes just outside the targeted device's enclosure. We then run RSA decryption in OpenSSL on the target device while recording the signal in a 40 MHz band around the clock frequency. The 40 MHz bandwidth was chosen as a compromise between recovery rate for the bits of the secret exponent and the availability and cost of receivers capable of capturing the desired bandwidth. Specifically, the 40 MHz bandwidth is well within the capabilities of Ettus USRP B200-mini receiver, which is very compact, costs less than $1,000, and can receive up to 56 MHz of bandwidth around a center frequency that can be set between 70 MHz and 6 GHz, and yet the 40 MHz bandwidth is sufficient to recover nearly all bits of the secret exponent from a single instance of exponentiation that uses that exponent.

We then apply AM demodulation to the received signal, and finally upsample it by a factor of 4. The upsampling consists of interpolating through the signal's existing sample points and placing additional points along the interpolated curve. This is needed because our receiver's sampling is not synchronized in any way to the computation of interest, so two signal snippets collected for the same computation may be misaligned by up to half of the sample period. Upsampling allows us to re-align these signals with higher precision, and we found that 4-fold upsampling yields sufficient precision for our purposes.

3.2 Identifying Relevant Parts of the Signal

Figure 4 shows a brief portion of the signal that begins during fixed-window exponentiation in OpenSSL. It includes part of one large-number multiplication (Line 7 in Figure 3), which in OpenSSL uses the Montgomery algorithm and a constant-time implementation designed to avoid multiplicand-dependent timing variation that was exploited by prior side-channel attacks. The point in time where Montgomery multiplication returns and the relevant part of the signal begins is indicated by a dashed vertical line in Figure 4. In this particular portion of the signal, the execution proceeds to lines 8 and 9 Figure 2, where a bit of the exponent is obtained and added to *wval*, then lines 10 and 6, and then 7 where, at the point indicated by the second dashed vertical line, it enters another Montgomery multiplication, whose signal continues well past the right edge of Figure 4. As indicated in the figure, the relevant part of the signal is very brief relative to the duration of the Montgomery multiplication.

A naive approach to identifying the relevant snippets in the overall signal would be to obtain reference signal snippets during training and then, during the attack, match against these reference snippets at each position in the signal and use the best-matching parts of the signal. Such signal matching works best when looking for a snippet that has prominent features, so they are unlikely to be obscured by the noise, and whose prominent features occur in a pattern which is unlikely to exist elsewhere in the signal. Unfortunately, the signal snippets relevant for our analysis have little signal variation (relative to other parts of the signal) and a signal shape (just a few up-and-downs) that many other parts of the signal resemble. In contrast, the signal that corresponds to the Montgomery multiplication has stronger features, and they occur in a very distinct pattern.

Therefore, instead of finding instances of relevant snippets by matching them against their reference signals from training, we use as a reference the signal that corresponds to the most prominent change in the signal during Mongtomery multiplication, where the signal abruptly changes from a period with a relatively low signal level to a period with a relatively high signal level. We identify this point in the signal using a very effi-

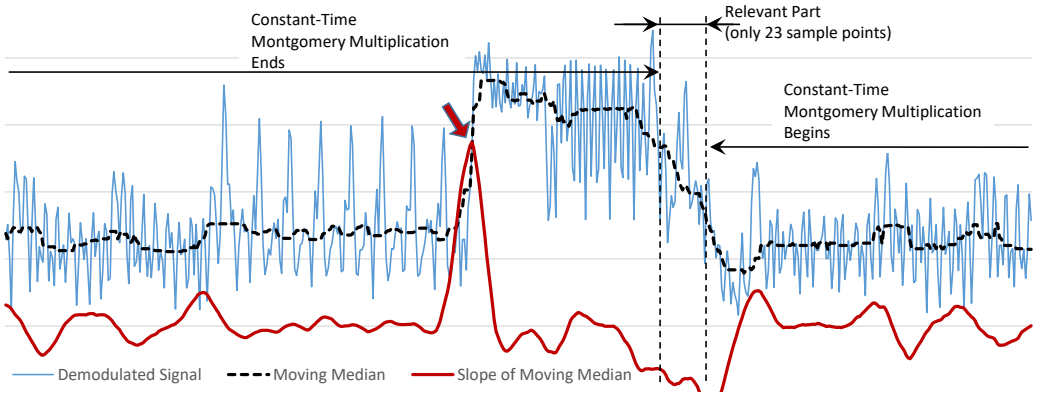

Figure 4: Signal that includes the end of one Montgomery multiplication, then the part relevant to our analysis, and then the beginning of another Montgomery multiplication. The horizontal axis is time (from left to right) and the vertical axis is the magnitude of the AM-demodulated signal.

cient algorithm. We first compute the signal's moving median (thick dashed black curve in Figure 4) to improve resilience to noise. We then examine the derivative (slope) of this moving median (thick red curve in Figure 4) to identify peaks that significantly exceed its statistically expected variation. In Figure4 the thick red arrow indicates such a peak, which corresponds to the most prominent change in the Montgomery multiplication that precedes the relevant part of the signal. Because the implementation of the Montgomery multiplication was designed to have almost no timing variation, the signal snippet we actually need for analysis is at a fixed time offset from the point of this match.

Because this method of identifying the relevant snippets of the signal is based on the signal that corresponds to the Montgomery multiplication that precedes each relevant snippet, the same method can be used for extracting relevant signal snippets for both fixed-window and sliding-window exponentiation – in both cases the relevant snippet is at the (same) fixed offset from the point at which a prominent-enough peak is detected in the derivative of the signal's moving median.

3.3 Recovering Exponent Bits in the Fixed-window Implementation

In the fixed-window implementation, large-number multiplication is used for squaring (Line 7 in Figure 3) and for updating the result after each window (Line 14). Thus there are four control-flow possibilities for activity between Montgomery multiplications.

The first two control flow possibilities begin when the Montgomery multiplication in line 7 completes. Both control flow possibilities involve updating the window value to include another bit from the exponent (lines 8, 9, and 10), and at line 6 incrementing i and checking it against w, the maximum size of the window. The first control flow possibility is the more common one - the window does not end and the execution proceeds to line 7 when another multiplication at line 7. We label this control flow possibility S-S (from a squaring to a squaring). The second control flow possibility occurs after the last bit of the window is examined and added to *wval*, and in that case the loop at line 6 is exited, the parameters for the result update at line 14 are prepared, and the Montgomery multiplication at line 14 begins. The parameter preparation in our code example would involve computing the address of $ct[wval]$ to create a pointer that would be passed to the Montgomery multiplication as its second multiplicand. In OpenSSL's implementation the ct is kept in a scattered format to minimize leakage of *wval* through the cache side channel while computing the Montgomery multiplication, so instead the value of *wval* is used to gather the scattered parts of $ct[wval]$ into a pre-allocated array that is passed to Montgomery multiplication. Since this pre-allocated array is used for all result-update multiplications, memory and cache behavior during the Montgomery multiplication no longer depend on *wval*. This means that in this second control-flow possibility involves significant activity to gather the parts of the multiplicand and place them into the pre-allocated array, and only then the Montgomery multiplication at line 14 begins. We label this control flow possibility S-U (from a squaring to an update).

The last two control flow possibilities occur after the result update in line 14 completes its Montgomery multiplication. The loop condition at line 2 is checked, and then one control flow possibility (third of the four) is that the entire exponentiation loop exits. We label this control flow possibility U-X (from an update to an exit). The last control-flow possibility, which occurs for all windows except the last one, is that after line 2 we execute

line 3, enter the window-scanning loop at line 6, and begin the next large-number Montgomery multiplication at line 7. We label this control flow possibility U-S (from an update to a squaring).

The sequence in which these four control flow possibilities are encountered in each window is always the same: $w-1$ occurrences of S-S, then one occurrence of S-U, then either U-S or U-X, where U-X is only possible for the last window of the exponent.

The first part of our analysis involves distinguishing among these four control flow possibilities. The reason for doing so is that noise bursts, interrupts, and activity on other cores can temporarily interfere with our signal and prevent detection of Montgomery multiplication. In such cases, sole reliance on the known sequence of control flow possibilities would cause a "slip" between the observed sequence and the expected one, causing us to use incorrect reference signals to recover bits of the exponent and to put the recovered bits at incorrect positions within the recovered exponent.

The classification into the four possibilities is much more reliable than recovery of exponent's bits. Compared to the other three possibilities, S-U spends significantly more time between Montgomery multiplications (because of the multiplicand-gathering activity), so it can be recognized with high accuracy and we use it to confirm that the exponentiation has just completed a window. The U-X possibility is also highly recognizable because, instead of executing Montgomery multiplication after it, it leads to executing code that converts from Montgomery to standard large-number format, and it serves to confirm that the entire exponentiation has ended. The S-S and U-S snippets both involve only a few instructions between Montgomery multiplications so they are harder to tell apart, but our signal matching still has a very high accuracy in distinguishing between them.

After individual snippets are matched to the four possibilities, that matching is used to find the most likely mapping of the sequence of snippets onto the known valid sequence. For example, if for $w = 5$ we observe S-U, U-S, S-S, S-S, S-S, S-U, all with high-confidence matches, we know that one S-S is missing for that window. We then additionally use timing between these snippets to determine the position of the missing S-S. Even if that determination is erroneous, we will correctly begin the matching for the next window after the S-U, so a missing snippet is unlikely to cause any slips, but even when it does cause a slip, such a slip is very likely to be "contained" within one exponentiation window. Note that a missing S-U or S-S snippet prevents our attack from using its signal matching to recover the value of the corresponding bit. A naive solution would be to assign a random value to that bit (with a 50% error rate among missing bits). However, for full RSA key recovery missing bits (erasures, i.e. the value of the bit is known to be unknown) are much less problematic than errors (the value of the bit is incorrect but not known a priori to be incorrect), we label these missing bits as erasures.

Finally, for S-S and S-U snippets we perform additional analysis to recover the bit of the exponent that snippet corresponds to. Recall that, in both S-S and S-U control flow possibilities, in line 9 a new bit is read from the exponent and is added to *wval*, and that bit is the one we will recover from the snippet. For ease of discussion, we will refer to the value of this bit as *bval*. To recover *bval*, in training we obtain examples of these snippets for each value of *bval*. To suppress the noise in our reference snippets and thus make later matching more accurate, these reference snippets are averages of many "identical" examples from training. Clearly, there should be separate references for $bval = 0$ (where only $bval = 0$ examples are averaged) and for $bval = 1$ (where only $bval = 1$ examples are averaged. However, *bval* is not the only value that affects the signal in a systematic way – the signal in this part of the computation is also affected by previous value of *wval*, loop counter *i*, etc. The problem is that these variations occur in the same part of the signal where variations due to *bval* occur, so averaging of these different variants may result in attenuating the impact of *bval*. We alleviate this problem by forming separate references for different bit-positions within the window, e.g. for window size $w = 5$ each value of *bval* would have 4 sets of S-S snippets and one set of S-U snippets, because the first for bits in the window correspond to S-S snippets and the last bit in the window to an S-U snippet. To account for other value-dependent in the signal, in each such set of snippets we cluster similar signals together and use the centroid of each cluster as the reference signal. We use the K-Means clustering algorithm and the distance metric used for clustering is Euclidean distance (sum of squared differences among same-position samples in the two snippets). We found that having at least 6-10 clusters for each set of snippets discussed above improves accuracy significantly. Beyond 6-10 clusters our recovery of secret exponent's bits improves only slightly but requires more training examples to compensate for having fewer examples per cluster (and thus less noise suppression in the cluster's centroid). Thus we use 10 clusters for each window-bit-position for each of the two possible values of *bval*. Overall, the number of S-S reference snippets for *bval* recovery is $2 * (w-1) * 10$ – two possible values of *bval*, $w-1$ bit-positions, 10 reference signals (cluster centroids) for each, while for S-U snippets we only have 20 reference snippets because S-U only happens for the last bit-position in the window. For commonly used window sizes this results in a relatively small overall number of reference snippets, e.g. for $w = 5$ there are only 100 reference snippets. To il-

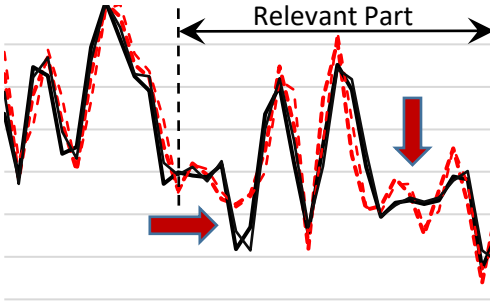

Figure 5: Example signal references (cluster centroid) for S-S snippets. Two references are shown for each value of the exponent's bit that corresponds to the snippet.

lustrate the difference in the signals created by the value of the exponent's bit, Figure 5 shows two reference S-S snippets (cluster centroids) for each value of the exponent's bit, with the most significant differences between 0-value and 1-value signals indicated by thick arrows.

Recall that, before attempting recovery of an unknown bit of the secret exponent, we have already identified which control-flow possibility (S-S or S-U) the snippet under consideration belongs to, and for S-S which bit-position it belongs to, so there are 20 reference snippets that each snippet-under-consideration is compared to (10 clusters for $bval = 0$ and 10 clusters for $bval = 1$). Thus the final step of our analysis involves finding the closest match (using Euclidean distance as a metric) among these 20 reference snippets and taking the *bval* associated with that reference snippet.

3.4 Recovering Exponent Bits in the Sliding-window Implementation

The sliding-window implementation of large-integer exponentiation (Figure 2) has three sites where Montgomery multiplication is called: the squaring within a window at line 26, which we label *S*, the update of the result at line 29, which we label *U*, and the squaring for a zero-valued window at line 7, which we label Z. The control flow possibilities between these include going from a squaring to another squaring (which we label as S-S). This transition is very brief (it only involves staying in the loop at line 25). The other transitions are S-U, which consumes more time because it performs the $ct[wval]$ computation; U-Z, which involves executing line 31, line 3, line 4 (where a bit of the exponent is examined), and finally entering Montgomery multiplication at line 7; U-S, which involves executing line 31, line 3, line 4, lines 11 and 12, and the entire window-scanning loop at lines 14-23, then line 25 and finally entering Montgomery multiplication at line 26; Z-Z where after line 7 the execution proceeds to line 8, line 9, line 3, line 4, and line 7 again; Z-S where after line 7 the execution proceeds to lines 8, 9, 3, 4, and then to lines 11 and 12, the loop at line 14-23, then line 25 and finally line 26; U-X where after the Montgomery multiplication at line 29 the execution proceeds to line 31 and then exits the loop at line 3; and finally S-X, where after Montgomery multiplication at line 7 the execution proceeds to lines 8 and 9 and then exits the loop at line 3.

Just like in fixed-window implementations, our recovery of the secret exponent begins with determining which snippet belongs to which of these control-flow possibilities. While in Section 3.3 this was needed only to correct for missing snippets, in the sliding-window implementation the window size varies depending on which bit-values are encountered in the exponent, so distinguishing among the control-flow possibilities is crucial for correctly assigning recovered bits to bit-positions in the exponent even if no snippets are missing. Furthermore, many of the exponent's bits can be recovered purely based on the sequence of these control-flow possibilities.

Our overall approach for distinguishing among control flow possibilities is similar to that in Section 3.3, except that here there are more control-flow possibilities, and the U-S and Z-S coarse-grained possibilities each have multiple control flow possibilities within the snippet: for each bit considered for the window, line 19 determines whether or not to execute lines 20 and 21. However, at the point in the sequence where U-S can occur, the only alternative is U-Z, which is much shorter and thus they are easy to tell apart. Similarly, the only alternative to Z-S is the much shorter Z-Z, so they are also easy to tell apart.

By classifying snippets according to which control-flow possibility they belong (where U-S and U-Z are each treated as one possibility), and by knowing the rules the sequence of these must follow, we can recover from missing snippets and, more importantly, use rules similar to those in [10] to recover many of the bits in the secret exponent. However, in contrast to work in [10] that could only distinguish between a squaring (line 7 or line 26, i.e. S or Z in our sequence notation) and an update (line 29, U in our sequence notation) using memory access patterns within each Montgomery multiplication (which implements both squaring and updates), our method uses the signal snippets between these Montgomery multiplications to recover more detailed information, e.g., for each squaring our recovered sequence indicates whether it is an S or a Z, and this simplifies the rules for recovery of exponent's bits and allows us to extract more of them. Specifically, after a U-S or Z-S, which compute the window value *wval*, the number of bits in the window can be obtained by counting the S-S occurrences that follow before an S-U is encountered. For example, consider the

sequence U-S, S-S, S-S, S-U, U-Z, Z-Z, Z-Z, Z-S. The first U-S indicates that a new window has been identified and a squaring for one of its bits is executed. Then the two occurrences of S-S indicate two additional squaring for this window, and S-U indicates that only these three squaring are executed, so the window has only 3 bits. Because the window begins and ends with 1-valued bits, it is trivial to deduce the values of two of these 3 bits. If we also know that $wmax = 5$, the fact that the window only has 3 bits indicates that the two bits after this window are both 0-valued (because a 1-valued bit would have expanded the window to include it). Then, after S-U, we observe U-Z, which indicates that the bit after the window is 0-valued (which we have already deduced), then two occurrences of Z-Z indicate two more 0-valued bits (one of which we have already deduced), and finally Z-S indicates that a new non-zero window begins, i.e. the next bit is 1. Overall, out of the seven bits examined during this sequence, six were recovered solely based on the sequence. Note that two of the bits (the two zeroes after the window) were redundantly recovered, and this redundancy helps us correct mistakes such as missing snippets or miss-categorized snippets.

In general, this sequence-based analysis recovers all zeroes between windows and two bits from each window. In our experiments, when using $wmax = 5$ this analysis alone on average recovers 68% of the secret exponent's bits, and with using $wmax = 6$, another commonly used value for *wmax*, this analysis alone on average recovers 55% of the exponent's bits. These recovery rates are somewhat higher than what square-update sequences alone enable [10], but recall that in our approach sequence recovery is only the preparation for our analysis of exponent-bit-dependent variation within individual signal snippets.

Since the only bits not already recovered are the "inner" (not the first and not the last) bits of each window, and since U-S and Z-S snippets are the only ones that examine these inner bits, our further analysis only focuses on these. To simplify discussion, we will use U-S to describe our analysis because the analysis for Z-S snippets is virtually identical.

Unlike fixed-window implementations, where the bits of the exponent are individually examined in separate snippets, in sliding-window implementations a single U-S or Z-S snippet contains the activity (line 4) for examining the first bit of the window and the execution of the entire loop (lines 14-23) that constructs the *wval* by examining the next $wmax - 1$. Since these bits are examined in rapid succession without intervening highly-recognizable Montgomery multiplication activity, it would be difficult to further divide the snippet's signal into pieces that each correspond to consideration of only one bit. Instead, we note that *wmax* is relatively small (typically 5 or 6), and that there are only 2^{wmax} possibilities for the control flow and most of the operands in the entire window-scanning loop. Therefore, in training we form separate reference snippets for each of these possibilities, and then during the attack we compare the signal snippet under consideration to each of the references, identify the best-matching reference snippet (smallest Euclidean distance), and use the bits that correspond to that reference as the recovered bit values.

3.5 Full Recovery of RSA Private Key Using Recovered Exponent Bits

Our RSA key recovery algorithm is a variant of the algorithm described by Henecka et al. [29], which is based on Heninger and Shacham's branch-and-prune algorithm [30]. Like Bernstein et al. [10], we recover from the side channel signal only the bits of the private exponents d_p and d_q, and the recovery of the full private key relies on exploiting the numerical relationships (Equations (1) in Bernstein et al. [10]) between these private exponents (d_p and d_q), the public modulus N and exponent e, and p and q, the private factors of N:

$$ed_p = 1 + k_p(p-1) \bmod 2^i$$
$$ed_q = 1 + k_q(q-1) \bmod 2^i$$
$$pq = N \bmod 2^i$$

where k_p and k_q are positive integers smaller than the public exponent e and satisfy $(k_p - 1)(k_q - 1) \equiv k_p k_q N \bmod e$. The public exponent practically never exceeds 32 bits [30] and in most cases $e = 65537$, so a key recovery algorithm needs to try at most e pairs of k_p, k_q.

We could not simply apply Bernstein's algorithm [10] to the exponents recovered by our signal analysis because, like the original branch-and-prune algorithm, such recovery requires certain knowledge of the bit values at some fraction of bit-positions in d_p and d_q, while the remaining bits are unknown but *known to be unknown*, i.e. they are *erasures* rather than errors. Such branch-and-prune search has been shown to be efficient when up to 50% of the bit-positions (chosen uniformly at random) in d_p and d_q are erasures, while its running time grows exponentially when the erasures significantly exceed 50% of the bit positions.

Henecka's algorithm [29] can be applied with the above pruning equations to recover the private key when some of the bits are in error. However, its pruning is based on a key assumption that errors are uniformly distributed, and it does not explicitly consider erasures. Recall, however, that for some of the bit positions our analysis cannot identify the relevant signal snippet for matching against training signals (see Section 3.2), which results in an erasure. A naive approach for handling erasures would be to randomly assign a bit value for each

erasure (resulting in a 50% error rate among erasures) and then apply Henecka's algorithm. Unfortunately, the erasures during our recovery are a product of disturbances in the signal that are very large in magnitude, and such a disturbance also tends to last long enough to affect multiple bits. With random values assigned to erasures, this produces 50%-error-rate bursts that are highly unlikely to be produced by uniformly distributed errors, causing Henecka's algorithm to either prune the correct partial candidate key or become inefficient (depending on the choice of the ε parameter).

Instead, we modify Henecka's algorithm to handle erasures by branching at a bit position when it encounters an erasure, but ignoring that bit position for the purposes of making a pruning decision. We further extend Henecka's algorithm to not do a "hard" pruning of a candidate key when its error count is too high. Instead, we save such a candidate key so that, if no candidate keys remain but the search for the correct private key is not completed, we can "un-prune" the lowest-error-count candidate keys that were previously pruned due to having too high of an error count. This is similar to adjusting the value of ε in Henecka's algorithm and retrying, except that the work of previous tries is not repeated, and this low cost of relaxing the error tolerance allows us to start with a low error tolerance (large ε in Henecka et al.) and adjust it gradually until the solution is found.

We further modify Henecka's algorithm to, rather than expand a partial key by multiple bits (parameter t in Henecka et al.) at a time, expand by one bit at a time and, among the newly created partial keys, only further expand the lowest-recent error-count ones until the desired expansion count (t) is reached. In Henecka's algorithm, full exansion by t bits at a time creates 2^t new candidate keys, while our approach discovers the same set of t-times-expanded non-pruned candidates without performing all t expansions on those candidates that encounter too many errors even after fewer than t single-bit expansions. For a constant t, this reduces the number of partial keys that are examined by a constant factor, but when the actual error rate is low this constant factor is close to 2^t.

Overall, our actual implementation of this modified algorithm is very efficient - it considers (expands by one bit) about 300,000 partial keys per second using a single core on recent mobile hardware (4th generation Surface Pro with a Core i7 processor), and for low actual error rates typically finds a solution after only a few thousand partial keys are considered. We evaluate its ability to reconstruct private RSA keys using d_p and d_q bits that contain errors and/or erasures by taking 1,000 RSA keys, introducing random errors, random erasures, and a half-and-half mix of errors and erasures, at different error/erasure rates, and counting how many partial keys had to be considered (expanded by a bit) before the correct private key was reconstructed. The median number of steps for each error/erasure rate is shown in Figure 6. We only show results for error/erasure rates up to 10% because those are the most relevant to our actual signal-based recovery of the exponent's bits.

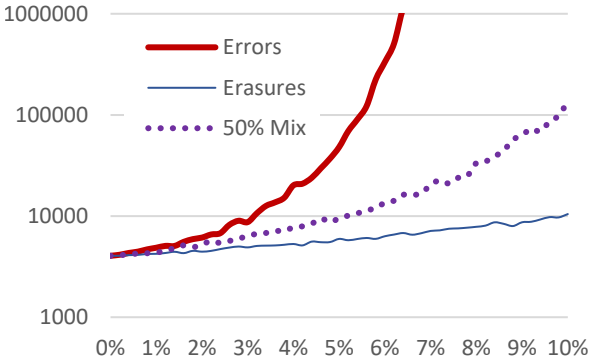

Figure 6: Single-bit expansion steps needed to reconstruct the private RSA key (vertical axis, note the logarithmic scale) as a function of the rate at which errors and/or erasures are injected (horizontal axis).

We observe that our implementation of reconstruction quickly becomes inefficient when only errors are present and the error rate approaches 7%, which agrees with the theoretical results of Henecka et al. – since d_p and d_q are used, the m factor in Henecka et al. is 2, and the upper bound for efficient reconstruction is at 8.4% error rate. In contrast, when only erasures are present, our implementation of reconstruction remains very efficient even as the erasure rate exceeds 10%, which agrees with Bernstein et al.'s finding that reconstruction should be efficient with up to 50% erasure rates. Finally, when equal numbers of errors and erasures are injected, the efficiency for each injection rate is close to (only slightly worse than) the efficiency for error-only injection at half that rate, i.e. with a mix of errors and erasures, the efficiency of reconstruction is largely governed by the errors.

Figure 7 shows the percentage of experiments in which the correct RSA key was recovered in fewer than 5,000,000 steps (about 17 seconds on the Surface 4 tablet). When only errors are present, $< 90\%$ of the reconstructions take fewer than 5,000,000 steps until the error rate exceeds 5.4%, at which point the percentage of under-five-million-steps reconstructions rapidly declines and drops below 10% at the 7.9% error rate. In contrast, all erasure-only reconstructions are under 5,000,000 steps even at the 10% erasure rate. Finally, when erasures and errors are both present in equal measure, the percentage of under-5,000,000-step reconstructions remains above 90% until the injection rate reaches 9.8% (4.9% of the bits are in error and another 4.9% are erased).

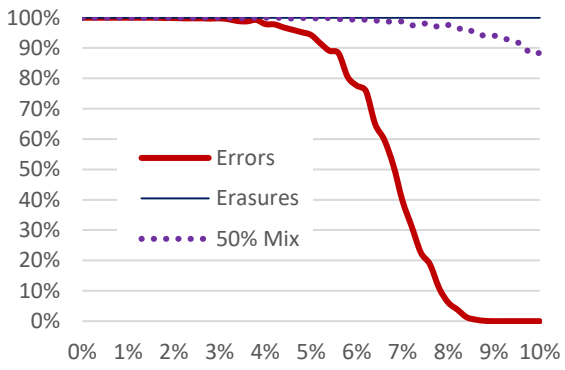

Figure 7: Percentage of keys recovered in fewer than 5,000,000 single-bit expansion steps (vertical axis) as a function of the rate at which errors and/or erasures are injected (horizontal axis).

4 Evaluation

In this section we describe our measurement setup and obtained results for recovering keys from blinded RSA encryption runs on three different devices.

4.1 Experimental Setup

We run the OpenSSL RSA application on Android smart phones Samsung Galaxy Centura SCH-S738C [41] and Alcatel Ideal [4], and on an embedded device (A13-OLinuXino board [38])) . The Alcatel Ideal cellphone has quad-core 1.1 GHz Qualcomm Snapdragon processor with Android OS(version 6) and the Samsung phone has a single-core 800 MHz Qualcomm MSM7625A Chipset with Android OS(version 5). The A13- OLinuXino board is a single-board computer that has an in order, 2-issue Cortex A8 ARM processor [5] and runs Debian Linux operating system.

In our experimental setup, we receive signals using small magnetic probe. We place the probe close to the monitored system as shown in Figure 8. The signals collected by the probe are recorded with Keysight N9020A MXA spectrum analyzer [33]. Our decision to use spectrum analyzer was mainly driven by its existing features such as built-in support for automating measurements, saving and analyzing measured results, visualizing the signals when debugging code, etc. We have observed very similar signals when using less expensive equipment such as Ettus USRP B200-mini receiver [18]. The analysis was implemented in MATLAB and on a personal computer runs in under one minute per decryption instance (i.e. per recovered 1024-bit exponent).

4.2 Experimental Results

4.3 Results for OpenSSL's Constant-Time Fixed-Window Implementation

Our first set of experiments evaluates the attack's ability to recover bits of the 1024-bit secret exponent d_p used during RSA-2048 decryption. OpenSSL uses a fixed window size $w = 5$ for exponentiation of this size. Note that RSA decryption involves another exponentiation, with d_q, and uses the Chinese Remainder Theorem to combine their results. However, the two exponentiations use exactly the same code and d_p and d_q are of the same size, so results for recovering d_q are statistically the same to those shown here for recovering d_p.

For each device, our training uses signals that correspond to 15 decryption instances, one for each of 15 randomly generated but known keys, and with ciphertext that is randomly generated for decryption. Note that these 15 decryptions provide around 12 thousand examples of S-S signal snippets, 3 thousand S-U, 3 thousand U-S, and 15 U-X snippets. This is more than enough examples of each control flow possibility to distinguish between these control flow possibilities accurately. More importantly, this provides on average 1,500 snippet examples for each of the 100 $(2*5*w)$ clusters whose centroids are used as reference snippets when recovering the bits of the unknown secret exponents. Note that using larger RSA keys proportionally increases the number of snippets produced by each decryption, while w changes little or not at all. Thus for larger RSA keys we expect that even fewer decryptions would be needed for training.

After training we perform the actual attack. We randomly generate 135 RSA-2048 keys, and for each of these keys we record, demodulate, and upsample (see Section 3.1) the signal that corresponds to *only one decryption* with that key, using a ciphertext that is randomly generated for each decryption. Next, the signal that corresponds to each decryption is processed to extract the relevant snippets from it (see Section 3.2). Then, as described in Section 3.3, each of these snippets is matched against reference snippets (from training) to identify which of the control-flow possibilities each snippet belongs to and, for S-S and S-U snippets, which bit-position in the exponent (and the window) the snippet corresponds to. Finally, S-S and S-U snippets are matched against the 20 clusters that correspond to its position in the window to recover the value of the bit at that position in the secret exponent.

The metric we use for the success of this attack is the success rate for recovery of exponent's bits, i.e. the fraction of the exponent's bits for which the recovery produces the value that the secret exponent at that position

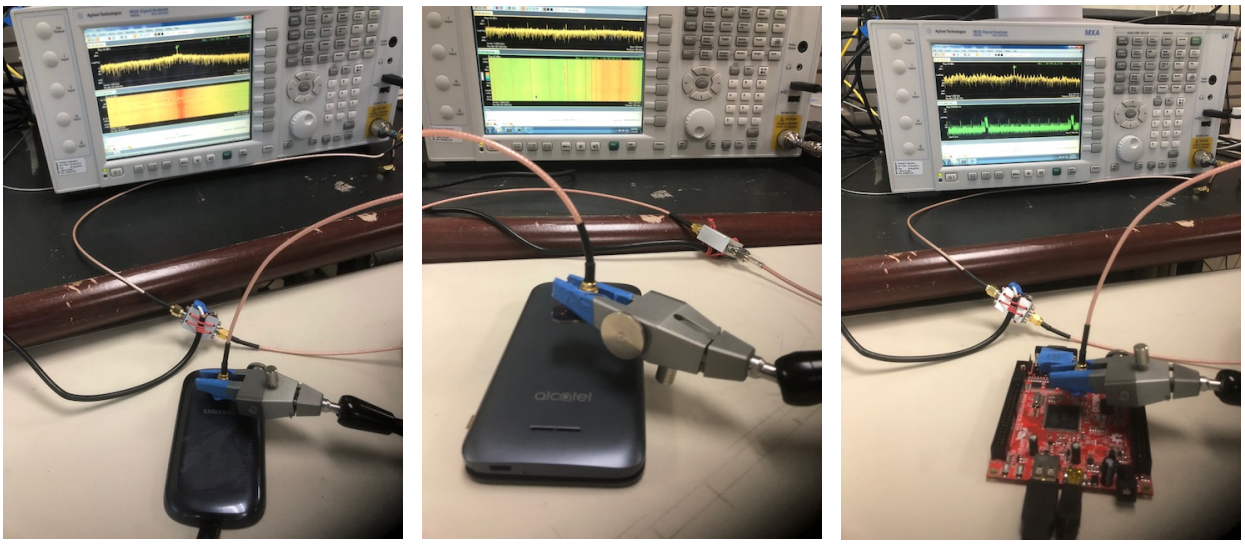

Figure 8: The measurement setup for each of the three devices (shown in the right-to-left order): Samsung Galaxy Centura SCH-S738C smart phone, Alcatel Ideal smart phone, and the A13-OLinuXino board.

actually had. To compute this success rate, we compare the recovered exponents to the actual exponents d_p and d_q that were used, counting the bit positions at which the two agree and, at the end, dividing that count with the total number of bits in the two exponents.

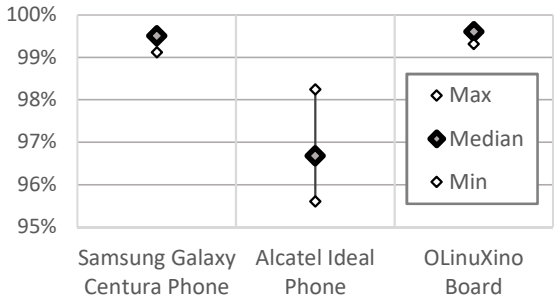

Figure 9: Success rate for recovery of secret exponent d_p's bits during only one instance of RSA-2048 decryption that uses that exponent. For each device, the maximum, median, and minimum success rate among decryption instances (each with a different randomly generated key) is shown.

The maximum, median, and minimum success rate for each of the three targeted devices is shown in Figure 9. We observe that the success rate of the attack is extremely high - among all decryptions on all three devices the lowest recovery rate is 95.7% of the bits. For the OLinuXino board, most decryption instances (>85% of them) had all bits of the exponent recovered correctly, except for the most significant 4 bits. These 4 bits are processed before entering the code in Figure 3 to leave a whole number of 5-bit windows for that code, so we do not attempt to recover them and treat them as erasures. Among the OLinuXino decryption instances that had any other reconstruction errors, nearly all had only one additional incorrectly recovered bit (error, not erasure), and a few had two.

The results for the Samsung phone were slightly worse – in addition to the 4 most significant bits, several decryption instances had one additional bit that was left unknown (erasure) because of an interrupt that occurs between the derivative-of-moving-median peak and the end of the snippet that follows it, which either obliterates the peak or prevents the snippet from correctly being categorized according to its control flow. In addition to these unknown (but known-to-be-unknown) bits, for the Samsung phone the reconstruction also produced between 0 and 4 incorrectly recovered (error) bits.

Finally, for the Alcatel Ideal phone most instances of the encryption had between 13 and 16 unknown bits in each of the two exponents, mostly because activity on the other three cores interferes with the activity on the core doing the RSA decryption), and a similar number of incorrectly recovered bits (errors).

To examine how the results would be affected when training using signals collected on one device and then recovering exponent bits using signals obtained from another device of the same kind, we use eight OLinuXino boards[1], which we label #1 through #8. Our training uses signals obtained only from board #1, and then the

[1]The OLinuXino boards are much less expensive than the phones, so we could easily obtain a number of OLinuXino boards

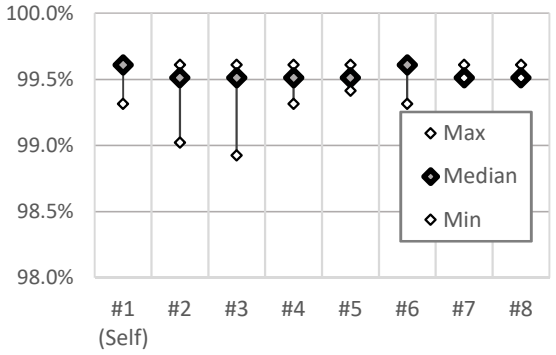

Figure 10: Success rate for recovery of secret exponent d_p's bits during only one instance of RSA-2048 decryption that uses that exponent, when training on OLinuXino board #1 and then using that training data for unknown exponent recovery on the same board and on seven other boards. For each device, the maximum, median, and minimum success rate among decryption instances (each with a different randomly generated key) is shown.

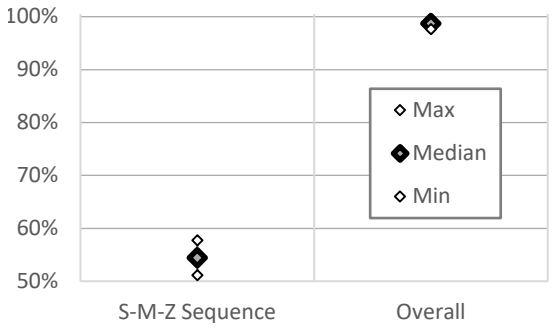

Figure 11: Success rate for recovery of secret exponent d_p's bits during only one instance of RSA-2048 decryption that uses that exponent for sliding-window exponentiation. The maximum, median, and minimum success rate among decryption instances (each with a different randomly generated key) is shown for recovery that only uses the snippet-type sequence (S-M-Z Sequence), and for recovery that also recovers window bits from U-S and Z-S snippets (Overall).

unknown keys are used on each of the eight boards and subjected to analysis using the same training data (from board #1). The results of this experiment are shown in Figure 10, where the leftmost data points correspond to training and recovery on the same device, while the remaining seven sets of data points correspond to training on one board and recovery on another.

These results indicate that training on a different device of the same kind does not substantially affect the accuracy of recovery.

Finally, for each RSA decryption instance, the recovered exponent bits, using both the recovered d_p and the recovered d_q, were supplied to our implementation of the full-key reconstruction algorithm. For each instance, the correct full RSA private key was reconstructed within one second on the Core i7-based Surface Pro 4 tablet, including the time needed to find the k_p and k_q coefficients that were not known a priori. This is an expected result, given that even the worst bit recovery rates (for the Alcatel phone) correspond to a an error rate of about 1.5%, combined with an erasure rate of typically 1.5% but sometimes as high as 3% (depending on how much system activity occurs while RSA encryption is execution on the phone), which is well withing the range for which our full-key reconstruction is extremely efficient.

4.4 Results for the Sliding-Window Implementation

To improve our understanding of the implications for this new attack approach, we also apply it to RSA-2048 whose implementation uses OpenSSL's sliding-window exponentiation – recall that this was the default implementation used in OpenSSL until it switched to a fixed-window implementation in response to attacks that exploit sliding-window's exponent-dependent square-multiply sequence.

In these experiments we use 160 MHz of bandwidth and target the OLinuXino board. Recall that in a sliding-window implementation our method can categorize the snippets according to their beginning/ending point to recover the sequence of zero-squaring (Z), window-squaring (S), and result update (M) occurrences. The fraction of the exponent's bits recovered by this sequence reconstruction (shown as "S-M-Z Sequence" in Figure 11) is in our experiments between 51.2% and 57.7% with a median of 54.5%. This sequence-based recovery has produces no errors in most cases (keys), and among the few encryptions that had any errors, none had more than one.

In our attack approach, after this sequence-based reconstruction, the U-S and Z-S snippets are subjected to further analysis to recover the remaining bits of the window computed in each U-S and Z-S snippet. At the end of this analysis, the fraction of the exponent's bits that are correctly recovered ("Overall" in Figure 11) is between 97.7% and 99.6%, with a median of 98.7%.

This rate of recovery for exponent bits provides for very rapid reconstruction of the full RSA key. However, we note that it is somewhat inferior to our results on fixed-window exponentiation on the same device (OLinuXino board), in spite of using more bandwidth for at-

tacks on sliding-window (160MHz bandwidth) than on fixed-window (40MHz bandwidth) implementation. The primary reason for this is that in the fixed-window implementation each analyzed snippet corresponds to examining only one bit of the exponent, whereas in the sliding-window implementation $wmax = 6$ bits of the exponent are examined in a single U-S or Z-S snippet, while the exponent-dependent variation in the snippet is not much larger. Since sliding-window recovery tries to extract several times more information from about the same amount of signal change, its recovery is more affected by noise and thus slightly less accurate.

5 Mitigation

We focus our mitigation efforts on the fixed-window implementation, which is the implementation of choice in the current version of OpenSSL, and which already mitigates the problem of exponent-dependent square-multiply sequences and timing variation. We identify three key enablers for this attack approach, which roughly correspond to discussion in Sections 3.1, 3.2, and 3.3. Successful mitigation requires removing at least one of these enablers, so we now discuss each of the attack enablers along with potential mitigation approaches focused on that enabler.

The first enabler of the specific attack demonstrated in this paper is the existence of computational-activity-modulated EM signals around the processor's clock frequency, and the attacker's ability to obtain these signals with sufficient bandwidth and signal-to-noise ratio. Potential mitigation thus include circuit-level approaches that reduce the effect the differences in computation have the signal, additional shielding that attenuates these signals to reduce their signal-to-noise ratio outside the device, deliberate creation of RF noise and/or interference that also reduces the signal-to-noise ratio, etc. We do not focus on these mitigation because all of them increase the device's overall cost, weight, and/or power consumption, all of them are difficult to apply to devices that are already in use, and all of them may not provide protection against attacks that use this attack approach but through a different physical side channel (e.g. power).

The second enabler of our attack approach is the attacker's ability to precisely locate, in the overall signal during an exponentiation operation, those brief snippets of signal that correspond to examining the bits of the exponent and constructing the value of the window. A simple mitigation approach would thus insert random additional amounts of computation before, during, and/or after window computation. However, additional computation that has significant variation in duration would also have a significant mean of that duration, i.e. it would slow down the window computation. Furthermore, it is possible (and indeed likely) that our attack can be adapted to identify and ignore the signal that corresponds to this additional activity.

The final (third) enabler of our attack approach is the attacker's ability to distinguish between the signals whose computation has the same control flow but uses different values for a bit in the exponent. In this regard, the attack benefits significantly from 1) the limited space of possibilities for value returned by `BN_is_bit_set` – there are only two possibilities: 0 or 1, and from 2) the fact that the computation that considers each such bit is surrounded by computation that operates on highly predictable values – this causes any signal variation caused by the return value of `BN_is_bit_set` to stand out in a signal that otherwise exhibits very little variation.

Based on these observations, our mitigation relies on obtaining all the bits that belong to one window at once, rather than extracting the bits one at a time. We accomplish this by using the `bn_get_bits` function (defined in `bn_exp.c` in OpenSSL's source code), which uses shifts and masking to extract and return a BN_ULONG-sized group of bits aligned to the requested bit-position – in our case, the LSB of the window. The BN_ULONG is typically 32 or 64 bits in size, so there are billions of possibilities for the value it returns, while the total execution time of `bn_get_bits` is only slightly more than the time that was needed to append a single bit to the window (call to `BN_is_bit_set` shifting the *wval*, and or-ing to update wval with the new bit). For the attacker, this means that there are now billions of possibilities for the value to be extracted from the signal, while the number of signal samples available for this recovery is similar to what was originally used for making a binary (single-bit) decision. Intuitively, the signal still contains the same amount of information as the signal from which one bit used to be recovered, but the attacker must now attempt to extract tens of bits from that signal.

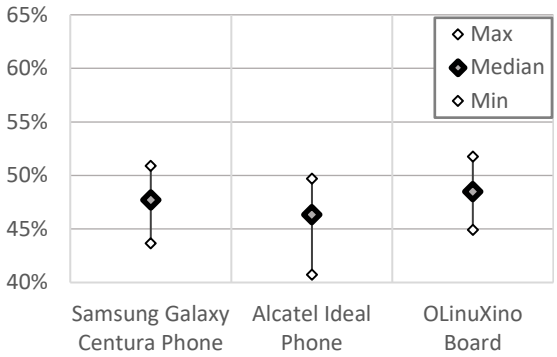

Figure 12: Success rate for recovery of secret exponent d_p's bits after the initial implementation of our window value randomization mitigation is applied.

This mitigation results in a slight *improvement* in execution time of the exponentiation and, as shown in Figure 12, with the mitigation the recovery rate for the exponent's bits is no better than randomly guessing each bit (50% recovery rate). In fact, the recovery rate with the mitigation is lower than 50% because, as in our pre-mitigation results, the bits whose signal snippets could not be located are counted as incorrectly recovered. However, these bits can be treated as erasures, i.e. for each such bit the attacker knows that the value of the bit is unknown, as opposed to a bits whose value is incorrect but the attacker has no a-priori knowledge of that, so our recovery rate can be trivially improved by randomly guessing (with 50% accuracy) the value of each erasure, rather than having 0% accuracy on them. With this, the post-mitigation recovery rate indeed becomes centered around 50%, i.e. equivalent to random guessing for all of the bits.

This mitigation has been submitted to OpenSSL and was merged into its master source code branch on May 20th, prior to the publication of this paper.

6 Conclusions

This paper presents the first side channel attack approach that, without relying on the cache organization and/or timing, retrieves the secret exponent from a single decryption on arbitrary ciphertext in a modern (current version of OpenSSL) fixed-window constant-time implementation of RSA. Specifically, the attack recovers the exponent's bits during modular exponentiation from analog signals that are unintentionally produced by the processor as it executes the constant-time code that constructs the value of each "window" in the exponent, rather than the signals that correspond to squaring/multiplication operations and/or cache behavior during multiplicand table lookup operations. The approach is demonstrated using electromagnetic (EM) emanations on two mobile phones and an embedded system, and after only one decryption in a fixed-window RSA implementation it recovers enough bits of the secret exponents to enable very efficient (within seconds) reconstruction of the full private RSA key.

Since the value of the ciphertext is irrelevant to our attack, the attack succeeds even when the ciphertext is unknown and/or when message randomization (blinding) is used. Our evaluation uses signals obtained by demodulating the signal from a relatively narrow band (40 MHz) around the processor's clock frequency (around 1GHz), which is within the capabilities of compact sub-$1,000 software-defined radio (SDR) receivers.

Finally, we propose a mitigation where the bits of the exponent are only obtained from an exponent in integer-sized groups (tens of bits) rather than obtaining them one bit at a time. This mitigation is effective because it forces the attacker to attempt recovery of tens of bits from a single brief snippet of signal, rather than having a separate signal snippet for each individual bit. This mitigation has been submitted to OpenSSL and was merged into its master source code branch prior to the publication of this paper.

7 Acknowledgments

We thank the anonymous reviewers for their very helpful comments and recommendations on revising this paper, and the developers of OpenSSL for helping us merge our mitigation into OpenSSL's source code repository on GitHub. This work has been supported, in part, by NSF grant 1563991and DARPA LADS contract FA8650-16-C-7620. The views and findings in this paper are those of the authors and do not necessarily reflect the views of NSF and DARPA.

References

[1] ACIIÇMEZ, O., KOÇ, C. K., AND SEIFERT, J.-P. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications security (ASIACCS)* (Mar. 2007), ACM Press, pp. 312–320.

[2] AGRAWAL, D., ARCHAMBEULT, B., RAO, J. R., AND ROHATGI, P. The EM side-channel(s). In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2002* (2002), pp. 29–45.

[3] AGRAWAL, D., ARCHAMBEULT, B., RAO, J. R., AND ROHATGI, P. The EM side-channel(s): attacks and assessment methodologies. In *http://www.research.ibm.com/intsec/emf-paper.ps* (2002).

[4] ALCATEL. Alcatel ideal / streak specifications. `http://www.phonescoop.com/phones/phone.php?p=5097`, Feb 24, 2016.

[5] ARM. Arm cortex a8 processor manual. `https://www.arm.com/products/processors/cortex-a/cortex-a8.php`, accessed April 3, 2016.

[6] BACKES, M., DURMUTH, M., GERLING, S., PINKAL, M., AND SPORLEDER, C. Acoustic side-channel attacks on printers. In *Proceedings of the USENIX Security Symposium* (2010).

[7] BALASCH, J., GIERLICHS, B., REPARAZ, O., AND VERBAUWHEDE, I. Dpa, bitslicing and masking at 1 ghz. In *Cryptographic Hardware and*

Embedded Systems – CHES 2015 (Berlin, Heidelberg, 2015), T. Güneysu and H. Handschuh, Eds., Springer Berlin Heidelberg, pp. 599–619.

[8] BANGERTER, E., GULLASCH, D., AND KRENN, S. Cache games - bringing access-based cache attacks on AES to practice. In *Proceedings of IEEE Symposium on Security and Privacy* (2011).

[9] BAYRAK, A. G., REGAZZONI, F., BRISK, P., STANDAERT, F.-X., AND IENNE, P. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the 48th Design Automation Conference (DAC)* (2011).

[10] BERNSTEIN, D. J., BREITNER, J., GENKIN, D., BRUINDERINK, L. G., HENINGER, N., LANGE, T., VAN VREDENDAAL, C., AND YAROM, Y. Sliding right into disaster: Left-to-right sliding windows leak. Conference on Cryptographic Hardware and Embedded Systems (CHES) 2017, 2017.

[11] BIHAM, E., AND SHAMIR, A. Differntial Cryptanalysis of the Data Encryption Standard. In *Proceedings of the 17th Annual International Cryptology Conference* (1997).

[12] BONEH, D., AND BRUMLEY, D. Remote Timing Attacks are Practical. In *Proceedings of the USENIX Security Symposium* (2003).

[13] BROUCHIER, J., KEAN, T., MARSH, C., AND NACCACHE, D. Temperature attacks. *Security Privacy, IEEE 7*, 2 (March 2009), 79–82.

[14] CALLAN, R., ZAJIC, A., AND PRVULOVIC, M. A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events. In *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)* (2014).

[15] CHARI, S., JUTLA, C. S., RAO, J. R., AND ROHATGI, P. Towards sound countermeasures to counteract power-analysis attacks. In *Proceedings of CRYPTO'99, Springer, Lecture Notes in computer science* (1999), pp. 398–412.

[16] CHARI, S., RAO, J. R., AND ROHATGI, P. Template attacks. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2002* (2002), pp. 13–28.

[17] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., AND SUTTER, B. D. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), pp. 45–60.

[18] ETTUS. Usrp-b200mini. `https://www.ettus.com/product/details/USRP-B200mini-i`, accessed February 4, 2018.

[19] GANDOLFI, K., MOURTEL, C., AND OLIVIER, F. Electromagnetic analysis: Concrete results. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems* (London, UK, UK, 2001), CHES '01, Springer-Verlag, pp. 251–261.

[20] GENKIN, D., PACHMANOV, L., PIPMAN, I., SHAMIR, A., AND TROMER, E. Physical key extraction attacks on pcs. *Commun. ACM 59*, 6 (May 2016), 70–79.

[21] GENKIN, D., PACHMANOV, L., PIPMAN, I., AND TROMER, E. Stealing keys from pcs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *Cryptographic Hardware and Embedded Systems – CHES 2015*, T. Gneysu and H. Handschuh, Eds., vol. 9293 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2015, pp. 207–228.

[22] GENKIN, D., PACHMANOV, L., PIPMAN, I., AND TROMER, E. Stealing keys from pcs using a radio: cheap electromagnetic attacks on windowed exponentiation. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)* (2015).

[23] GENKIN, D., PACHMANOV, L., PIPMAN, I., TROMER, E., AND YAROM, Y. Ecdsa key extraction from mobile devices via nonintrusive physical side channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1626–1638.

[24] GENKIN, D., PIPMAN, I., AND TROMER, E. Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs. In *Cryptographic Hardware and Embedded Systems CHES 2014*, L. Batina and M. Robshaw, Eds., vol. 8731 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 242–260.

[25] GENKIN, D., PIPMAN, I., AND TROMER, E. Get your hands off my laptop: physical side-channel key-extraction attacks on pcs. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)* (2014).

[26] GENKIN, D., SHAMIR, A., AND TROMER, E. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference (CRYPTO)* (2014).

[27] GIRAUD, C. DFA on AES. In *Advanced Encryption Standard - AES, 4th International Conference, AES 2004* (2003), Springer, pp. 27–41.

[28] GOUBIN, L., AND PATARIN, J. DES and Differential power analysis (the "duplication" method). In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999* (1999), pp. 158–172.

[29] HENECKA, W., MAY, A., AND MEURER, A. Correcting Errors in RSA Private Keys. In *Proceedings of CRYPTO* (2010).

[30] HENINGER, N., AND SHACHAM, H. Reconstructing rsa private keys from random key bits. In *International Cryptology Conference (CRYPTO)* (2009).

[31] HUTTER, M., AND SCHMIDT, J.-M. The temperature side channel and heating fault attacks. In *Smart Card Research and Advanced Applications*, A. Francillon and P. Rohatgi, Eds., vol. 8419 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 219–235.

[32] KARATSUBA, A., AND OFMAN, Y. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences 145*, 293-294 (1962).

[33] KEYSIGHT. N9020a mxa spectrum analyzer. `https://www.keysight.com/en/pdx-x202266-pn-N9020A/mxa-signal-analyzer-10-hz-to-265-ghz?cc=US&lc=eng`, accessed February 4, 2018.

[34] KHUN, M. G. Compromising emanations: eavesdropping risks of computer displays. *The complete unofficial TEMPEST web page: http://www.eskimo.com/˜joelm/tempest.html* (2003).

[35] KOCHER, P. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of CRYPTO'96, Springer, Lecture notes in computer science* (1996), pp. 104–113.

[36] KOCHER, P., JAFFE, J., AND JUN, B. Differential power analysis: leaking secrets. In *Proceedings of CRYPTO'99, Springer, Lecture notes in computer science* (1999), pp. 388–397.

[37] MESSERGES, T. S., DABBISH, E. A., AND SLOAN, R. H. Power analysis attacks of modular exponentiation in smart cards. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999* (1999), pp. 144–157.

[38] OLIMEX. A13-olinuxino-micro user manual. `https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware`, accessed April 3, 2016.

[39] OPENSSL SOFTWARE FOUNDATION. OpenSSL Cryptography and SSL/TLS Toolkit. `https://www.openssl.org`.

[40] PERCIVAL, C. Cache missing for fun and profit. In *Proc. of BSDCan* (2005).

[41] SAMSUNG. Samsung galaxy centura sch-s738c user manual with specs. `http://www.boeboer.com/samsung-galaxy-centura-sch-s738c-user-manual-guide-straight-talk/`, June 7, 2013.

[42] SCHINDLER, W. A timing attack against RSA with Chinese remainder theorem. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2000* (2000), pp. 109–124.

[43] SHAMIR, A., AND TROMER, E. Acoustic cryptanalysis (On nosy people and noisy machines). http://tau.ac.il/˜tromer/acoustic/.

[44] TSUNOO, Y., TSUJIHARA, E., MINEMATSU, K., AND MIYAUCHI, H. Cryptanalysis of block ciphers implemented on computers with cache. In *Proceedings of the International Symposium on Information Theory and its Applications* (2002), pp. 803–806.

[45] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture* (2007), ACM, pp. 494–505.

[46] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), USENIX Association, pp. 719–732.

[47] ZAJIC, A., AND PRVULOVIC, M. Experimental demonstration of electromagnetic information leakage from modern processor-memory systems. *Electromagnetic Compatibility, IEEE Transactions on 56*, 4 (Aug 2014), 885–893.