Vulnerabilities Introduced by Features for Software-based Energy Measurement

Technical Report TUD-CS-2017-0304

November 2017

Heiko Mantel, Johannes Schickel, Alexandra Weber, Friedrich Weber

Technische Universität Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Modeling and Analysis of Information Systems

This work has been funded by the DFG as part of the project Secure Refinement of Cryptographic Algorithms (E3) within the CRC 1119 CROSSING.

CROSSING

Vulnerabilities Introduced by Features for Software-based Energy Measurement

Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber

Department of Computer Science, TU Darmstadt, Germany
{mantel, schickel, weber, fweber}@mais.informatik.tu-darmstadt.de

Abstract. The growing need to save energy has led to the introduction of new features into CPUs. Such features allow software-based energy measurement in order to track and limit consumption as necessary, e.g., for green IT. Despite the importance of these new CPU features to datacenters, their consequences on software security are not understood yet. In this article, we investigate a security issue caused by these features, namely software-based energy side channels. We identify a weakness in Bouncy Castle that allows an attacker to distinguish two secret RSA keys based on their induced energy consumption characteristics. Through software-based energy measurement, the attacker can sample the energy consumption even without physical access to the system. We show how very few such samples already give the attacker a high probability of success. From an initial investigation of countermeasures against software-based energy side channels, we conclude that the mitigation of such weaknesses requires further attention.

1 Introduction

Side channels are unintended communication channels arising from physical execution characteristics of a program. Characteristics known to introduce side channels include: execution time [4, 10, 11, 20], power consumption [21, 22, 35], and cache behavior [8, 29, 49]. All of these characteristics can be exploited by a remote attacker [8, 11, 35]. In this article, we focus on a different class, namely *software-based energy side channels*, and present a method to exploit them.

Software-based energy measurement features, e.g., in Intel CPUs, support the achievement of energy-consumption goals in datacenters (see, e.g., [19, Ch. 14. 9]). In view of green IT and increasing energy prices, features for monitoring energy consumption become more and more important.

Our research project was triggered by the introduction of such software-based energy measurement features. Our goal was to clarify whether features for software-based energy measurement introduce any danger for a software-based energy side channel that an attacker could exploit to obtain secret information.

For a detailed investigation, we focus on the example of Intel RAPL, a software-based energy measurement feature introduced by Intel in desktop and server CPUs [19]. We investigate how Intel RAPL introduces weaknesses such that an attacker could obtain secret information through an software-based energy side channel. To this end, we perform distinguishing experiments at the example of the RSA implementation in the cryptographic library Bouncy Castle. We measure the energy consumption of RSA decryption operations in software using Intel RAPL. That is, we do not require physical access to the system.

Based on our experiments, we detect a weakness that Intel RAPL introduces into Bouncy Castle RSA. We find that two RSA keys are distinguishable by the energy consumption of RSA decryption, measured via Intel RAPL. This weakness is exposed across a broad spectrum of attacker models, ranging from an invasive attacker with root privileges to a passive attacker observing a concurrent process.

We quantify the severity of the detected weakness based on statistical methods. We define a decision procedure with which an attacker could guess the key based on his measurements. For this procedure and our measurements, we compute the probability that the attacker guesses the key correctly after making a certain number of side-channel observations. In our experiments, we find that with just 25 side-channel observations, the probability for a correct guess is above 99%. Hence, the weakness in Bouncy Castle RSA is a serious concern.

Two obvious countermeasures against the described energy side channels are: forbidding access to Intel RAPL completely or very restrictive access control. In addition, more flexible software-level countermeasures would be desirable. We investigate the effectiveness of such countermeasures at the example of cross-copying [2]. Unfortunately, our experiments show that cross-copying is not very effective as a countermeasure. Moreover, based on the experiences gained, it is unclear to us how an effective software-level countermeasure could look like.

In summary, the main contribution of this article is a cross-cutting study of the security implications of software-based energy measurement features at the example of the feature Intel RAPL. The study covers

- a qualitative analysis how attackers with different capabilities could make use of energy side channels,
- a quantitative analysis how easily attackers could make use of such side channels, and
- an initial investigation of potential countermeasures against energy side channels.

This article is structured as follows. In Section 2, we recapitulate key concepts needed. We define the attacker models we consider in Section 3 and present our approach to assess such side channels in Section 4. We present our qualitative results on Bouncy Castle RSA in Section 5 and our quantitative results in Section 6. In Section 7 we summarize our evaluation of cross-copying. After discussing related work in Section 8, we conclude in Section 9.

2 Preliminaries

2.1 Side Channels

A prime example of a side channel goes back to Kocher's seminal work on timing attacks [20]. Kocher shows that a naive square-and-multiply implementation of modular exponentiation is vulnerable to timing attacks. Modular exponentiation is, for example, used in RSA decryption. Decryption of an RSA ciphertext requires computation of $p = c^d \pmod{n}$ [41], where c is the ciphertext and d is the secret exponent. A naive implementation is given in Figure 1. Line 5 is only executed when the condition in Line 4 evaluates to *true*. Execution of Line 5 takes additional time. Since the condition depends on bits from the exponent, the execution time of the program encodes the Hamming weight of the exponent. An attacker can exploit this variation in execution times to extract the secret exponent d (as shown in [20]).

In the style of Millen [36], a side channel can be modeled as an information-theoretic channel [15], where the input alphabet and output alphabets are modeled by a random variables X and Y, respectively. The input alphabet are the possible secret inputs a program can process, and the output alphabet are the possible observations an observer can make through the side channel.

The leakage through a side channel can be measured by the mutual information [15] of X and Y, i.e., by the amount of information that Y contains about X. The information contained in the variables is measured using a notion of entropy. Multiple definitions of entropy exist. In this article, we rely on the classical definition of Shannon entropy [45].

The mutual information of a channel depends on the prior probability distribution of secrets in X (e.g., if one secret is chosen with probability 1, no additional information can be gained through the side channel). The channel capacity [15] $C(X;Y)$ is defined as the worst-case mutual information across all prior distributions. In this article, we use channel capacity to quantify the effectiveness of a countermeasure against software-based energy side channels.

2.2 Software-Based Energy Measurement

Energy (measured in J for *joule*) and power (measured in W for *watt*) are well-defined physical concepts. They are related as follows: Energy E is the aggregation of instantaneous power values over time, i.e., $E = \int_{t_0}^{t_1} p(t)dt$, where $p(t)$ is the instantaneous power consumption [17].

```
 1: Input: $(d, n), c$
 2: $r \leftarrow 1$
 3: for $i = 1$ to $i = bitLength(d)$ do
 4:     if $d \,\%\, 2 == 1$ then
 5:         $r \leftarrow (r * y) \,\%\, n$
 6:     end if
 7:     $y \leftarrow (y * y) \,\%\, n$
 8:     $d \leftarrow d >> 1$
 9: end for
10: return $r \,\%\, n$
```

Fig. 1: Square-and-multiply modular exponentiation

We define the energy consumption of a software S as the energy consumed by the CPU and the main memory during execution of S. Energy consumption is caused, for example, by accessing data in the main memory or by the CPU doing arithmetic calculations. Our definition is similar to [38]'s definition.

Software-based energy measurement allows a software to monitor the energy consumption of the system through functionality provided by the hardware itself. For example, the *Power Capping framework* (powercap) on Linux [1] exposes an energy counter to userspace software.

Running Average Power Limit (Intel RAPL) is a set of energy sensors introduced with Intel's Sandy Bridge processor architecture [18]. While Intel RAPL's primary purpose is to enforce power consumption limits [19, Ch. 14], it also exposes the energy consumption of the CPU through the *model-specific register* (MSR) MSR_PKG_ENERGY_STATUS. This register is updated every millisecond. The energy measurements provided by Intel RAPL are accurate [18].

Linux exposes Intel RAPL to userspace through the msr kernel module [28] and powercap. The module msr provides access to MSRs through pseudo-files. For example, the MSRs of the first CPU core can be accessed through /dev/cpu/0/msr. Both, loading the module and accessing its pseudo-files, requires *root* privileges. Unlike the pseudo-files for MSRs, powercap's pseudo-file /sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj can be accessed by any user, even without root privileges. From this file, the user can obtain an energy consumption measurement in the unit of $\mu J = 10^{-6} J$.

2.3 Distinguishing Experiments

In a *distinguishing experiment*, two distinct secret inputs are passed to a program and a side channel output is repeatedly measured for each input. For instance, Mantel and Starostin [33] use distinguishing experiments to show that a program exhibits a timing-side-channel vulnerability.

Based on the empirical data obtained in a distinguishing experiment, different statistical tools can be used to quantify the side-channel leakage of the program under test. For a given attacker strategy, the success probability can be computed based on hypothesis testing. Independent of a concrete attacker strategy, the side-channel capacity $C(X;Y)$ of the program can be estimated with a statistical procedure (e.g, the one by Chatzikokolakis, Chothia, and Guha [12]).

A *test of hypothesis* (or short *test*) is a tool to investigate conformance of a hypothesis with observations from an experiment [46, p. 64]. The hypothesis to test is called the *null hypothesis* (H_0). We denote the alternative hypothesis by H_1. There are two error cases in a test: (a) the test accepts H_0 although H_1 holds, or (b) the test refutes H_0 although H_0 holds. Case (a) is called a *false positive*, Case (b) is called a *false negative*. The probabilities for a false positive and a false negative are denoted by $P(H_0|H_1)$ and $P(H_1|H_0)$, respectively.

The *binomial distribution* (or Bernoulli distribution) is the probability distribution for the number of successes in n independent experiments [46, p. 112]. The probability that in n experiments, each featuring success probability p, r

successes are observed is given as follows:

$$P_{n,p}(r) = \binom{r}{n} p^r p^{n-r}$$

Where $\binom{r}{n} = \frac{n!}{r!(n-r)!}$ is the *binomial coefficient*. We write $P_{n,p}(r \leq X)$ for the probability that at most X out of n experiments exhibit a success. This probability is given by:

$$P_{n,p}(r \leq X) = \sum_{i=0}^{X} P_{n,p}(i)$$

Conversely, the probability that more than X successes are observed in n experiments is given by:

$$P_{n,p}(r > X) = 1 - P_{n,p}(r \leq X)$$

Chothia and Smirnov show in [13] how tests of hypothesis can be used to distinguish the e-passport of a victim from e-passports of other people. Based on a simple selection criterion, their distinguishing attack tests the hypothesis that the passport under attack belongs to the victim (H_0). Using $P(H_0|H_1)$ and $P(H_1|H_0)$, Chothia and Smirnov calculate the number of observations an attacker needs to distinguish passports with error rates below 1%.

2.4 Program Transformations for Side Channels

Multiple source-to-source program transformations were proposed for mitigating timing side channels, including cross-copying [2], conditional assignment [37], transactional branching [6], and unification [24]. The *cross-copying* technique transforms a program such that alternative branches execute in constant time whenever the choice of the branch might depend on secrets. Cross-copying pads branches by adding copies of the statements in one branch to the end of the respective other branch. Instead of the original statements, dummy statements are used in the copies, i.e., statements that do not affect the program's state, but require the same execution time as the respective original statements.

The cross-copying technique was analytically verified to soundly enforce a timing-sensitive noninterference-like property [2]. In addition, cross-copying's effectiveness to mitigate timing side channels was evaluated experimentally. In [33], an estimation of side-channel capacity based on experimental results confirmed the effectiveness of cross-copying when applied to modular exponentiation.

2.5 RSA in Bouncy Castle

Bouncy Castle is a collection of cryptography implementations for Java, maintained by Legion of the Bouncy Castle Inc. [27]. It contains implementations for various popular symmetric and asymmetric cryptography primitives. A provider class allows the use of Bouncy Castle through the Java Cryptography Extension (JCE). In the form of Spongy Castle [42], Bouncy Castle is widely used on Android, e.g., in the WhatsApp messenger [29]. Bouncy Castle's popularity makes side-channel weaknesses a serious security threat. Recently, Lipp, Gruss, Spreitzer, Maurice, and Mangard have shown Bouncy Castle 1.5's AES implementation is vulnerable to cache side-channel attacks [29].

Bouncy Castle contains implementations of various variants of the RSA asymmetric encryption scheme. The RSA encryption and decryption functionality is implemented in the Java class RSAEngine. RSAEngine can be used either directly or as backend in cipher modes, such as *OAEP* [7] and *PKCS1* [43]. An RSA key can be generated using the class RSAKeyPairGenerator.

3 Attacker Models

In this section, we capture the capabilities of an attacker who can exploit software-based energy side channels by an *attacker model*. More concretely, we introduce three attacker models relevant for software-based energy side channels.

In each of the models, the attacker can execute an attack procedure on the machine running the victim program, but the attacker cannot obtain the secret directly. The attack procedure has standard capabilities, in particular, it can query energy consumption information using a system API.

We define the following three attacker models, presented in decreasing order of capabilities:

invasive	An attacker under *invasive* can trigger victim program executions at will. In addition, he can modify the victim program to add code paths to measure the energy consumption of parts of the program. He also has privileged access to the system, e.g., he can alter the system configuration itself during run-time.
sequential	An attacker under *sequential* can trigger victim program executions at will.
concurrent	An attacker under *concurrent* is passive. In particular, he cannot trigger victim program executions at will.

Attackers under *invasive* and *sequential* are active, i.e., they can trigger victim program executions at will such that they can obtain any number of observations. In contrast, *concurrent* can only passively observe executions of the victim program. Thus, he is dependent on a third party triggering executions of the victim program to obtain measurements. The attacker model *invasive* is used as a reference point in our study. An attacker under this model can precisely measure the energy consumption of a single function and thus identify whether a function is vulnerable to an energy side channel.

We substantiate our attacker models in the context of the Linux operating system. For all three attacker models, the attack procedure on Linux can read pseudo-files in the /sys and /proc file systems. These file systems are standard Linux file systems containing information about the system itself and currently running processes. Through the /sys file system, an attacker can read powercap's pseudo-files containing information about the energy consumption of the system. An attacker under *invasive* is privileged by the possession of root permissions. This allows the attacker to load kernel modules and to modify system configuration by writing into the /sys file system.

We will implement attack procedures for each of the attacker models in Section 4 and use them to investigate software-based energy side channels in Sections 5 and 6.

4 Our Approach

The core of many side-channel attacks is to distinguish between secrets from a restricted set (e.g., a set of secrets varying only in one bit or byte), based on a collection of samples obtained through a side channel. Already Kocher's [20] timing attack in 1996 was based on bit-wise distinguishing between secret inputs. Bernstein [8] distinguished between AES keys on the byte-level. More recently, AlFardan and Paterson [3] mounted a distinguishing attack on implementations of the TLS record protocol. Their attack distinguishes between two plaintexts by the time taken to decryption them. Furthermore, they describe an attack that recovers entire plaintexts using byte-wise distinguishing based on running times.

Using distinguishing experiments [33], one can detect weaknesses in implementations that allow to distinguish between two secret inputs, e.g., as a basic step in a side-channel attack. We follow the approach of distinguishing experiments to assess the vulnerability of cryptographic implementations. We define a general procedure for distinguishing experiments, because no explicit definition existed so far. One of the steps in our procedure for distinguishing experiments, namely the sample-collection step, depends on the attacker model. We implement this phase for all three attacker models defined in Section 3.

4.1 Procedure for Distinguishing Experiments

An implementation *imp* is assessed with respect to a particular security concern, namely the leakage of a secret input s to an attacker under an attacker model a. For instance, *imp* could be an RSA implementation, s could be the secret RSA key, and a could be the model *sequential* of an energy-side-channel attacker. The assessment consists of four steps, visualized in Figure 2: input generation, sample collection, result computation, and result evaluation.

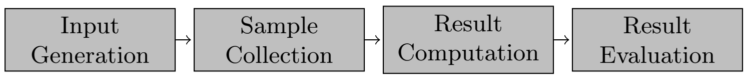

Fig. 2: Procedure for a distinguishing experiment

In the first step, *input generation*, two input vectors to the implementation *imp* are generated. The input vectors differ only in the secret input s. All values in the input vectors must be within the spectrum of valid input data. For instance, to assess the leakage of a secret RSA key, two valid secret RSA keys are generated randomly.

In the *sample collection* step, the implementation *imp* is run on the two input vectors that were generated in the previous step. For both runs, the observation made under the attacker model a is recorded. This step is repeated many times to obtain a collection of observations for each input.

Next, in the *result computation* step, the arithmetic means of the two collections of observations are computed and the collections are plotted as histograms. That is, for each collection, the frequency with which each observation occurs in the collection is computed and visualized.

The last step is the *result evaluation*. From the difference between the mean observations for the two inputs, one can already assess whether an attacker would be able to distinguish between the two inputs. Furthermore, the degree of overlap between the histograms for the two outputs allows one to assess how difficult the distinction would be.

Based on the means and histograms obtained with a distinguishing experiment, one can detect weaknesses in implementations (if the means and histograms are clearly distinguishable). This allows a proactive assessment with which weaknesses can be identified early, before they are exploited by concrete attacks. In addition to such qualitative results, quantitative results can be obtained through a statistical test, as we describe in Section 6.

4.2 Sample Collection

To carry out distinguishing experiments against software-based energy side channels, we implement the sample collection step for each of our the attacker models *sequential*, *concurrent*, and *invasive*.

Sample Collection under *sequential* Our measurement procedure for energy samples under *sequential* is shown in Figure 3 as pseudocode. For experimental evaluation, we implemented the procedure in Python.

Firstly (Line 2), the attacker reads the energy-consumption counter through powercap. To this end, the function READCOUNTER is called. Secondly, the attacker waits busily for the first change to the energy-consumption counter (Lines 3 – 5). Once the counter has been refreshed, the attacker invokes an execution of the victim program (Line 6) using the invocation command supplied as input to the attack procedure. After executing the victim program, the attacker queries the energy-consumption counter again (Line 7). The difference between the values of the counter before and after the victim's execution is the attacker's sample. If the sample is negative, that is, if there was a wraparound of the counter, the sample is discarded (Line 9). Otherwise, the sample is returned.

Sample Collection under *concurrent* Unlike attackers under *sequential*, an attacker under *concurrent* cannot actively trigger executions of the victim program. Hence, an attacker under *concurrent* needs to identify when a decryption

```
1: function readCounter
2:     contents ← read /sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj
3:     return toInteger(contents)
4: end function
```

```
 1: Parameters: cmdLine
 2: $E_{instant}$ ← readCounter()
 3: repeat                          ▷ Align beginning of measurement with register update
 4:     $E_{begin}$ ← readCounter()
 5: until $E_{begin} \neq E_{instant}$
 6: invoke(cmdLine)                                             ▷ Execute victim program
 7: $E_{end}$ ← readCounter()
 8: if $E_{end} < E_{begin}$ then
 9:     discard measurement                                  ▷ A wraparound has occurred
10: else
11:     return $E_{end} - E_{begin}$
12: end if
```

Fig. 3: Measurement procedure under *sequential*

operation takes place. For our analysis, we model the step as follows: The attacker waits for execution of the victim program. When the program starts its execution, the attacker obtains the current energy consumption counter, waits for the program to terminate, and obtains the energy consumption counter. The difference between both values is the energy consumption of the victim program.

We use Python to implement the measurement procedure. Pseudocode for the procedure is shown in Figure 4. The attacker waits until the victim program is executed (Lines 2 – 17). He detects the invocation of a program by monitoring the /proc filesystem. He recognizes the victim program by the command that was used to invoke it (Line 11). Once the victim program is executed, the attacker measures the energy consumption (Lines 19 – 27).

Sample Collection under *invasive* Under *invasive*, an attacker can modify the source code of the program under attack. We consider an attack procedure where an attacker under *invasive* encapsulates a routine in the victim program to log its energy consumption. To measure the energy consumption, the attacker accesses the machine-specific registers exposed through the msr kernel module.

We implement a measurement functionality for energy consumption of the CPU through Intel RAPL in C, using the msr kernel module. We use this measurement functionality in Java through a Java Native Interface module. Pseudocode for the measurement under *invasive* is shown in Figure 5.

The victim program is modified such that it calls `NativeAlignedMSR.start` before the victim routine and `NativeAlignedMSR.stop` after the victim routine. The attacker collects samples by calling `NativeAlignedMSR.getComsumption`. To obtain precise measurements, we follow the alignment approach from [18]. On program start up, we monitor 1000 Intel RAPL register updates to estimate the

```
Parameters: $victimComm$                    ▷ the command name of the victim program
function WAITFORVICTIM
    while $true$ do
        $lastpid \leftarrow$ fifth field of /proc/loadavg
        repeat
            $newlastpid \leftarrow$ fifth field of /proc/loadavg
        until $lastpid \neq newlastpid$
        $pid \leftarrow lastpid$
        while $pid \leq newlastpid$ do
            $comm_{pid} \leftarrow$ contents of /proc/$pid$/comm
            if $comm_{pid} = victimComm$ then
                return $pid$
            end if
            $pid \leftarrow pid + 1$
        end while
    end while
end function
$pid \leftarrow$ WAITFORVICTIM()
$E_{begin} \leftarrow$ READCOUNTER()
while /proc/$pid$/ exists do
    do nothing
end while
$E_{end} \leftarrow$ READCOUNTER()
if $E_{end} < E_{begin}$ then
    discard measurement                              ▷ A wraparound has occurred
else
    return $E_{end} - E_{begin}$
end if
```

Fig. 4: Measurement procedure under *concurrent*

the power consumption of the system. We use the estimated power consumption to adjust for the energy consumption while waiting for register updates.

5 Qualitative Results on Bouncy Castle RSA

We investigate the consequences of software-based energy measurement on software security at the example of Intel RAPL and Bouncy Castle RSA. Using a distinguishing experiment, we identify that running Bouncy Castle RSA on a system with Intel RAPL gives rise to a weakness. The energy consumption of the decryption operation allows to distinguish between secret RSA keys. In the following, we describe the setup and results of our experiment in detail.

5.1 Experimental Setup

Assessed Implementation To assess the vulnerability of Bouncy Castle RSA, we implement a Java program RSA that decrypts an RSA ciphertext using

```
1: function NativeAlignedMSR.start
2:     $eBefore \leftarrow$ value of MSR counter upon next update
3: end function
4: function NativeAlignedMSR.stop
5:     $eAfter \leftarrow$ value of MSR counter upon next update
6: end function
7: function NativeAlignedMSR.getComsumption
8:     return $eAfter$ - $eBefore$
9: end function
```

Fig. 5: Measurement procedure under *invasive*

```
1: Input: $(d, n), ct$
2: $rsa \leftarrow$ New RSAEngine()
3: $rsa$.init($false, (d, n)$)
4: $result \leftarrow rsa$.processBlock($ct, 0, ct.length$)
5: return $result$
```

Fig. 6: RSA decryption

Bouncy Castle 1.53. It takes a secret key and a ciphertext as input. It decrypts the ciphertext, using the secret key, and then returns the resulting plaintext.

Figure 6 lists the pseudo-code of the program. Line 4 decrypts the ciphertext *ct* using the secret key *(d, n)*. processBlock is a method from Bouncy Castle's RSAEngine class, which implements the RSA decryption.

We use our implementation for distinguishing experiments with respect to *sequential* and *concurrent*. As described in Section 4.2, attackers under *invasive* encapsulate the victim routine. To account for such encapsulation, we implement an additional wrapper for Bouncy Castle RSA.

The wrapper for Bouncy Castle RSA under *invasive* is shown as pseudocode in Figure 7. The code inserted by an attacker under *invasive* is represented by Line 3, Line 5, and Line 7. Line 5 starts the attacker's measurement procedure. Line 7 stops the attacker's measurement procedure.

Machine Configuration We conduct our experiments on a Lenovo ThinkCentre M93p featuring one RAPL-capable Intel i5-4590 CPU @ 3.30GHz with 4GB of RAM. The machine runs Ubuntu 14.10 with a Linux kernel version 3.16.0-44-generic from Ubuntu's repository. The programs are executed using an OpenJDK 7 64-bit server Java Virtual Machine version 7u79-2.5.5-0ubuntu0.14.10.2 from Ubuntu's repository. To simulate a server machine that is shared between attacker and victim, we disable the X-server.

In the distinguishing experiments under *invasive*, we account for the additional root privileges of an attacker under this model. To this end, we disable all but the first CPU core, as a user with root permissions would do to reduce the noise in his energy-consumption measurements.

```
1: Input: $(d, n), ct$
2: $rsa \leftarrow$ NEW RSAEngine()
3: $measurement \leftarrow$ NEW $NativeAlignedMSR()$
4: $rsa$.INIT($false, (d, n)$)
5: $measurement$.START()
6: $result \leftarrow rsa$.PROCESSBLOCK($ct, 0, ct.length$)
7: $measurement$.STOP()
8: print $measurement$.GETCONSUMPTION()
9: return $result$
```

Fig. 7: Modified RSA decryption

Parameters and Sampling We generate two RSA keys $k1$ and $k2$ to supply as input to our RSA decryption program during our distinguishing experiment. First, we randomly select two 1536 bit primes p and q to calculate the 3072 bit modulus $n = p * q$ shared by our keys. Then, we randomly select a ciphertext $c < n$. Finally, to select private exponents for the two keys $k1$ and $k2$, we exploit that $d * e \equiv 1 \pmod{(p-1)*(q-1)}$ must hold for valid RSA keys [41]. For $k1$, we randomly generate a public exponent e_{k1} and calculate the corresponding private exponent d_{k1}. For $k2$, we fix the public exponent to $e_{k2} = 65\,537$ and calculate the corresponding private exponent d_{k2}. The secret exponents that we obtain for $k1$ and $k2$ have Hamming weight 1460 and 1514, respectively.

In our distinguishing experiments, we utilize our measurement procedures to collect 100 000 samples per secret key under the attacker models *sequential* and *concurrent*. For the attacker model *concurrent*, under which an attacker cannot trigger executions of the victim program himself, we invoke the victim program after random delays between 100ms and 1000ms. Under *invasive*, we collect 60 000 samples per secret key (note that, for *invasive* a sample takes approximately 2.5s instead of at most 1.3s for *concurrent*).

We reject outliers that lie further than six median absolute deviations from the median. For $k1$, we reject 1.24% of the samples under *sequential*, 10.78% of the samples under *concurrent*, and 2.22% of the samples under *invasive*. For $k2$, we reject 1.11% of the samples under *sequential*, 11.01% of the samples under *concurrent*, and 2.28% of the samples under *invasive*. We plot the collected samples for each key and attacker model as histograms.

5.2 Results for *sequential*

The samples collected in our distinguishing experiment under *sequential* are depicted in Figure 8. One histogram of energy-consumption samples is given per input. The histograms are colored based on the input: The blue (left) histogram corresponds to the samples for $k1$ with Hamming weight 1460, and the red (right) histogram corresponds to the samples for $k2$ with Hamming weight 1514.

The estimated mean energy consumption for $k1$ is $5.07J$, and for $k2$ the estimated mean energy consumption is $5.14J$. The peaks of the histograms and the mean energy consumptions for the inputs are clearly distinct.

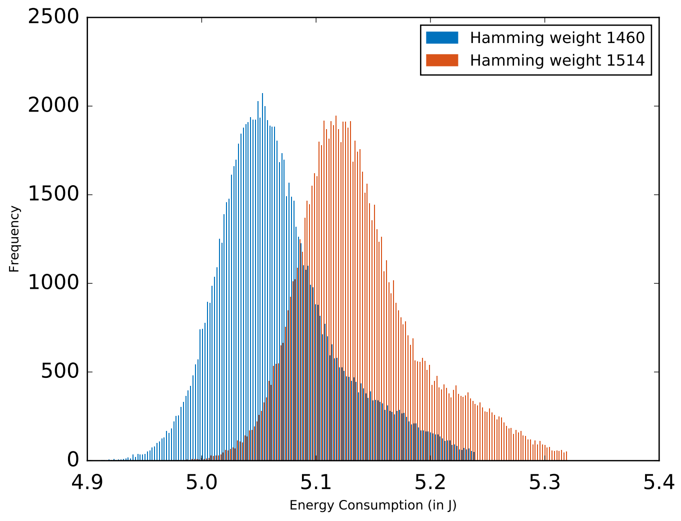

Fig. 8: Results for *sequential*

Based on the histograms, an attacker under the model *sequential* can distinguish between the two secret RSA keys. Hence, there is a weakness in Bouncy Castle RSA in the presence of the Intel RAPL feature.

5.3 Results for *concurrent*

Figure 9 shows the histograms of the samples per key under *concurrent*. Again, the blue (left) histogram corresponds to $k1$ (Hamming weight 1460) and the red (right) histogram corresponds to $k2$ (Hamming weight 1514).

The mean energy consumptions are $7.20J$ and $7.32J$ for the keys with Hamming weights 1460 and 1514, respectively. The peaks of the two histograms are clearly distinct. Interestingly, the overlap of the histograms is even a bit smaller compared to the overlap of the histograms under *sequential*. We will get back to this peculiarity in Section 6.

The mean energy consumptions and the histograms for the two RSA keys are clearly distinct. This means that the weakness we detected in Bouncy Castle RSA is even exposed to the weaker attacker model *concurrent*, under which an attacker only passively observes an RSA decryption.

Remark 1. Note that, the energy consumption measured under *concurrent* increased significantly by $2.13J$ and $2.18J$, respectively, compared the observations under *sequential*. This increase is due to the attacker actively monitoring the /proc filesystem to identify termination of the RSA process.

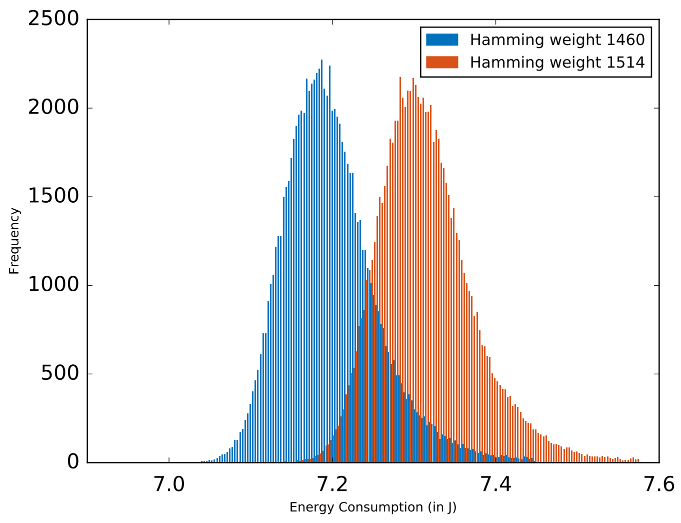

Fig. 9: Results for *concurrent*

5.4 Results for *invasive*

Figure 10 shows the results of our experiment under *invasive*. Again, the blue (left) histogram shows the energy-consumption samples for $k1$, and the red (right) histogram shows the energy-consumption samples for $k2$.

The mean energy consumption for the keys $k1$ and $k2$ is $1.84J$ and $1.91J$, respectively. The peaks of the two histograms are clearly distinct.

The clearly distinct peaks and distinct mean values imply that an attacker can distinguish the keys based on the energy-consumption characteristics of the decryption that they induce. The degree to which the histograms and means differ is similar to the model under *sequential*. That is, the additional attacker capabilities of *invasive* do not significantly amplify the weakness.

Remark 2. Note that, the energy consumption measured under *invasive* is noticeably lower than the consumption measured under *sequential* and *concurrent*. This is due to the different experimental setup we used with *sequential* and *concurrent*. In the setup used for *sequential* and *concurrent*, the additional CPU cores consume additional energy. Furthermore, *sequential* and *concurrent* start and stop their measurement outside the victim program, i.e., they also measure the energy consumed by the wrapper around RSA.

Overall, we identify a weakness in Bouncy Castle RSA that is exposed across the attacker models *sequential*, *concurrent*, and *invasive*. For all three models, the mean energy consumption of the decryption routine differs significantly between the two RSA keys that we consider.

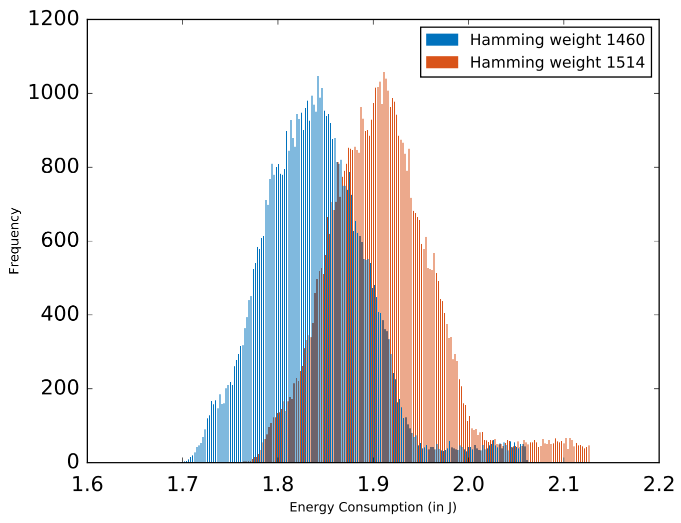

Fig. 10: Results for *invasive*

Based on the histograms from our distinguishing experiments, an attacker is able to clearly distinguish between the two secret keys if he collects enough samples. In the following section, we quantify exactly how many samples an attacker needs in order to be successful.

6 Quantification of the Weakness

The results of our distinguishing experiments show that it is intuitively possible that an attacker can distinguish RSA keys by exploiting a weakness in Bouncy Castle RSA via Intel RAPL. We further investigate the likelihood of an attacker to distinguish keys. To this end, we devise a test procedure that allows an attacker to guess which of the two RSA key is used during decryption. Based on the false positive and false negative rates of the test procedure, we compute how many measurements an attacker requires to correctly guess the key in 99% of all cases.

6.1 A Distinguishing Test

Side-channel attacks, e.g., [8, 13], can be mounted in two phases. In the first phase, the attacker collects a set of offline observations through the side channel as reference point, possibly on a different machine with the same software and hardware setup as the machine he shares with the victim. During the second phase, the attacker collects a set of online observations on the machine he shares with the victim. By relating his online side-channel observations with the offline observations, the attacker deduces information about the secret being processed.

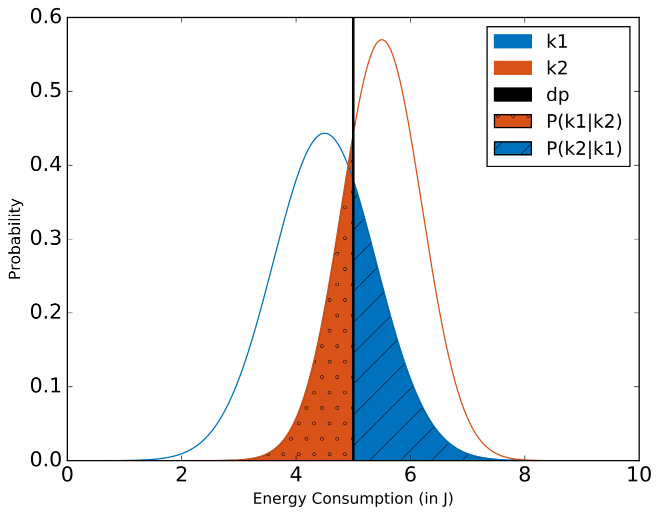

Fig. 11: Example of a distinguishing test

For our distinguishing experiment setting, the offline observations are the collected energy-consumption characteristics of the RSA decryption operation for both, $k1$ and $k2$. The online observations would be side-channel observations collected to identify which key is used during a system run. To guess which key the system is using, the attacker compares how likely the learned energy-consumption characteristics allow to explain the observations. We model the guess by a statistical test to distinguish between the keys.

The attacker's distinguishing test works as follows: Given two keys, $k1$ and $k2$, with mean energy consumptions of m_{k1} and m_{k2}, where $m_{k1} < m_{k2}$, the attacker determines a *distinguishing point* dp using the following formula:

$$dp = \frac{m_{k1} + m_{k2}}{2}$$

If the attacker observes an energy consumption less than dp, the attacker guesses $k1$ was used. In case the observed energy consumption is higher than or equal to dp, the attacker guesses $k2$ was used. A false positive is: $k2$ was used for decryption, but the attacker guesses $k1$ was used. A false negative is: $k1$ was used for decryption, but the attacker guesses $k2$ was used.

A visualization of an example for the test is given in Figure 11. The distribution of energy consumption values for $k1$ and $k2$ each follow a normal distribution. More concretely, for $k1$ the distribution follows $\mathcal{N}(4.5J, 0.81)$, while for $k2$ the distribution follows $\mathcal{N}(5.5J, 0.49)$. Thus, the decision point is at $5J$. The area under the curve $k2$ to the left of dp corresponds to the false positive probability of the test. In this example, the probability is $P(k1|k2) = 23.75\%$. Conversely, the area under the curve $k1$ to the right of dp corresponds to the false negative probability of the test. The probability for our example is $P(k2|k1) = 28.93\%$.

The attacker can use majority voting to increase his chances of guessing the correct key. For this, he observes multiple decryption operations and uses his test on each observation. Based on the individual guesses, he chooses the key on which the majority of guesses agreed. Let n be the number of observations the attacker makes. Then the false positive probability is given as:

$$p^n_{P(k1|k2)} = P_{n,P(k1|k2)}(r > \lfloor \frac{n}{2} \rfloor)$$

and the false negative probability is given by:

$$p^n_{P(k2|k1)} = P_{n,P(k2|k1)}(r > \lfloor \frac{n}{2} \rfloor)$$

Based on $P(k1|k2)$ and $P(k2|k1)$, one can determine the number n of observations needed for the attacker in order to distinguish between the keys with 99% success rate, i.e., with $p^n_{P(k1|k2)} < 1\%$ and $p^n_{P(k2|k1)} < 1\%$.

In our example, which is visualized in Figure 11, majority voting allows to determine the number of observations as follows. Based on the false positive probability $P(k1|k2) = 23.75\%$, we calculate that an attacker requires 17 observations to achieve a false positive rate below 1% in the majority voting case. More concretely, the majority voting features a false positive probability of $p^{17}_{P(k1|k2)} = 0.87\%$. Conversely, the number of observations required to obtain a false negative rate below 1% for $P(k2|k1) = 28.93\%$ is 29. For 29 observations, the false negative probability for majority voting is $p^{29}_{P(k2|k1)} = 0.81\%$. We conclude that the attacker requires 29 observations to distinguish between the two keys successfully in 99% of all cases.

6.2 Quantitative Results

	n	1 observ.	7 observ.	13 observ.	17 observ.	19 observ.	25 observ.
invasive	$p^n_{P(k1\|k2)}$	27.54%	9.64%	4.05%	2.35%	1.80%	**0.82%**
	$p^n_{P(k2\|k1)}$	23.48%	5.74%	1.73%	**0.81%**	0.55%	0.18%
sequential	$p^n_{P(k1\|k2)}$	24.75%	6.83%	2.30%	1.16%	**0.83%**	0.31%
	$p^n_{P(k2\|k1)}$	19.77%	3.20%	**0.66%**	0.24%	0.14%	0.03%
concurrent	$p^n_{P(k1\|k2)}$	13.69%	**0.87%**	0.07%	0.01%	0.007%	0.0006%
	$p^n_{P(k2\|k1)}$	13.39%	**0.80%**	0.06%	0.01%	0.005%	0.0005%

Table 1: False-positive and false-negative rates for attackers

For a quantitative evaluation of the weakness in Bouncy Castle RSA, we need to know the false positive and false negative probabilities of the distinguishing test. We estimate the probabilities based on the energy consumption characteristics collected offline by the attacker on his reference system. To estimate $P(k1|k2)$, we count the number of offline observations below dp of decryption samples with $k2$ and divide them by the total number of offline observations for $k2$. Conversely, to estimate the false negative probability we count the number of offline observations above dp of decryption samples of $k1$ and divide them by the total number of offline observations for $k1$. Formally, the probabilities can be estimated as follows. Let O_{k1} be the set of all offline observations for decryption operations with $k1$ and let O_{k2} be the set of all offline observations for $k2$.

$$P(k1|k2) = \frac{\{x|x \in O_{k2} \land x < dp\}}{|O_{k2}|}$$
$$P(k2|k1) = \frac{\{x|x \in O_{k1} \land x \geq dp\}}{|O_{k1}|}$$

We evaluate the weakness for all three attacker models, i.e., *invasive*, *sequential*, and *concurrent*, using our distinguishing test. For *invasive*, the distinguishing point is at $dp = 1.88J$, due to the means for $k1$ and $k2$ being $1.84J$ and $1.91J$, respectively (see Section 5.4). For *sequential*, the distinguishing point is at $dp = 5.10J$, due to the means for $k1$ and $k2$ being $5.07J$ and $5.14J$, respectively (see Section 5.2). For *concurrent*, the distinguishing point is at $dp = 7.26J$, due to the means for $k1$ and $k2$ being $7.20J$ and $7.32J$, respectively (see Section 5.3).

Table 1 lists the false positive and false negative probabilities $p^n_{P(k1|k2)}$ and $p^n_{P(k1|k2)}$ that result from given amounts n of online observations under the three attacker models. Note that, $P(k1|k2) = p^1_{P(k1|k2)}$ and $P(k2|k1) = p^1_{P(k2|k1)}$. In addition to $p^1_{P(k1|k2)}$ and $p^1_{P(k2|k1)}$, we only list the cases in which one of the probabilities falls below 1% for the first time. We highlight the first value below 1% for each of the probabilities by printing it in bold face.

The false positives for 1 observation range from 13.69% for *concurrent* to 27.54% for *invasive*. The false negatives for 1 observation range from 13.39% for *concurrent* to 23.48% for *invasive*. Already for 7 online observations, the false positive and false negative probabilities fall below 1% for *concurrent*. At 13 observations, the false negative probability for *sequential* falls below 1%. The false negative probability of *invasive* falls below 1% at 17 observations. For *sequential* the false positive probability falls below 1% at 19 observations. At 25 observations the false positive probability for *invasive* falls below 1%.

The distinguishing tests show that, in the worst case, only 25 observations are required to distinguish key $k1$ from key $k2$ in 99% of all cases. In this case of 25 observations, *concurrent*'s test exhibits false negative and false positive probabilities below 0.001% each. This means that, given only 25 decryption observations, *concurrent* can distinguish both keys in 99.999% of all cases. Moreover, to distinguish both keys in 99% of all cases, *concurrent* requires only 7 observations. The finding that *concurrent*, our weakest attacker model, can distinguish both keys with high likelihood at 7 observations and, even worse, with near certainty at 25 observations, gives us reason to classify the weakness we discovered as severe.

Comparing the distinguishing tests for the attackers modeled by *invasive*, *sequential*, and *concurrent* gives the surprising result that *concurrent* requires the least amount of observations to distinguish both keys in 99% of the cases. An attacker under *concurrent* only requires 7 observations to distinguish both keys, i.e., less than a third of the observations an attacker under *invasive* needs ($n = 25$). Intuitively, an attacker under *invasive* should be able to distinguish the keys more easily compared to *sequential* and *concurrent*, due to its ability to carry out more precise measurements.

After investigating the histograms from Section 5 again, our explanation why an attacker under *concurrent* can distinguish keys more easily than an attacker under *invasive* is as follows. For all attacker our models, *invasive*, *sequential*, and *concurrent*, the overlap between both histograms seems to be roughly $0.25J$ wide. The estimated means differ by $0.07J$, $0.07J$, and $0.12J$, respectively. Together with visual inspection, we identify that, while the overlapping area remains similar with decreasing attacker capabilities, the peaks move further apart. Thus, the likelihood to observe an energy consumption value from the overlapping area decreases. As a result, for *concurrent*, the likelihood that a value between both peaks is observed is the lowest across the attacker models. This means that an attacker under *concurrent* has the highest likelihood to distinguish between the two RSA keys, which is also shown by our quantitative results.

7 A Security Evaluation of Cross-Copying

As we have shown in the previous sections, software-based energy side channels are a serious threat. Such side channels could be avoided by restricting access to software-based energy management. Unfortunately, such a restriction would limit the use of features like Intel RAPL, e.g., in green IT. On the other hand, software-level countermeasures would allow to use software-based energy measurement while mitigating the leakage through energy side channels.

One candidate software-level countermeasure is cross-copying [2] – originally a countermeasure against timing side channels. Cross-copying ensures that all secret-dependent branches execute equivalent statements. Intuitively, execution of equivalent statements should consume equivalent amounts of energy. Thus, we anticipate that cross-copying mitigates software-based energy side channels.

In the following, we evaluate the effectiveness of cross-copying to mitigate software-based energy side channels, based on information theory. We do not aim to remove the weakness in Bouncy Castle RSA. Rather, we are interested whether cross-copying can, in principle, protect against weaknesses due to software-based energy side channels.

7.1 Case Study

To investigate whether cross-copying can help to mitigate leakage through software-based energy side channels, we quantify its effectiveness on a small example program. Motivated by the weakness that we detected in the Bouncy Castle RSA

```
1: Input: $(d, n), c$
2: $r \leftarrow 1$
3: for $i = 1$ to $i = bitLength(d)$ do
4:     if $d \,\%\, 2 == 1$ then
5:         $r \leftarrow (r * y) \,\%\, n$
6:     else
7:         $r_{dummy} \leftarrow (r_{dummy} * y) \,\%\, n$
8:     end if
9:     $y \leftarrow (y * y) \,\%\, n$
10:    $d \leftarrow d >> 1$
11: end for
12: return $r \,\%\, n$
```

Fig. 12: Cross-copied modular exponentiation

implementation, we base the example on RSA. More concretely, we use a naive implementation of square-and-multiply modular exponentiation (Figure 1).

We first check that there is a concern for software-based energy side channel leakage already in this simple implementation. To this end, we approximate the channel capacity for this example. In the next step, we check whether cross-copying mitigates this threat. To this end, we approximate the channel capacity of a cross-copied version of the example program. We evaluate the effectiveness of the countermeasure by the reduction in channel capacity that it causes.

The cross-copied example program is given in Figure 12. As visible in the figure, cross-copying inserts a dummy assignment (Line 7) into the else-branch of the secret-dependent branching. The branches now contain equivalent statements whose execution is anticipated to consume an equivalent amount of energy.

7.2 Experimental Setup

For brevity, we call the naive square-and-multiply implementation *baseline* and the cross-copied implementation *cross-copied*. For experimental evaluation, we use [33]'s Java implementation of baseline and cross-copied. We adapt the implementations to log the energy consumption measured through powercap. To avoid zero energy consumption results due to execution times below 1ms, we repeat the computation 1.31×10^5 times. This results in approximately 100 updates of the energy-consumption counter for a single execution of the baseline version. We estimate the channel capacity using an iterative Blahut-Arimoto algorithm [5, 9] based on the samples collected during the distinguishing experiment.

We conduct distinguishing experiments in a system setting similar to an attacker under *invasive* in Section 5.1. In addition, we disable the network to reduce noise in the measurements and we disable the just-in-time (JIT) compiler of the Java VM to prevent optimizations from interfering with our results. We use two input vectors that share $n = 4096$ and $c = 1\,234\,567\,890$. One secret exponent with Hamming weight 5 ($d = 2\,080\,374\,784$) and one secret exponent

	Baseline		**Cross-Copied**	
$mean(E)(nJ)$	**Input 1**	**Input 2**	**Input 1**	**Input 2**
	15370.07 ± 3.18	18925.46 ± 4.00	20372.21 ± 4.48	21040.05 ± 3.97
$C(X;Y)$	0.9922 ± 0.0		0.9171 ± 0.0097	

Table 2: Statistical results for modular exponentiation

with Hamming weight 25 ($d = 33\,554\,431$) are used as the first and second value of the secret input, respectively.

We follow [33] and collect 10 000 samples per input. We reject outliers that lie further than six median absolute deviations from the median. In the baseline version, we reject 1.32% and 1.07% of all samples for input one and input two, respectively. For cross-copied, 1.77% and 2.73% of all samples are rejected for inputs one and two, respectively.

7.3 Experimental Results

Table 2 presents the results of our experiments. The mean energy-consumptions for baseline and cross-copied for each input and the channel capacities for baseline and cross-copied are given with 95% confidence intervals.

The mean energy consumption for the first input to baseline is roughly $15\,373.73nJ$. The mean energy consumption for the second input to baseline is roughly $18\,934.13nJ$. These means are clearly distinguishable. Hence, there is a clear security concern already in this simple example.

The channel capacity quantifies the threat to the implementation. Since we consider a scenario in which the attacker tries to distinguish between two inputs to the implementation, the secret is 1 bit, namely the choice of the input. Since $C(X;Y)$ is approximately 0.9922 bits/symbol, one attacker observation reveals almost the entire secret under the worst-case prior distribution of inputs.

We investigate the results for cross-copied. Here, the mean energy consumptions for the two inputs are roughly $20\,372.21nJ$ and $21\,040.05nJ$, respectively. Intuitively, these means are still clearly distinguishable.

The quantification of the security concern for the cross-copied version confirms that the concern is still substantial. The channel capacity is approximately 0.9171 bits/symbol. Cross-copied can still leak 91% of the secret under the worst-case prior input distribution. This shows that [33]'s cross-copying implementation does not mitigate the software-based energy side channel significantly.

We can only speculate why cross-copying is not effective against the energy side channel in our experiments. The difference of data dependencies introduced by the branches might be responsible. In the *else* branch (Figure 12 Line 7), the result is written to r_{dummy} instead of r. This might cause a subtle difference in energy consumption, for example, due to different patterns of pipeline stalling. In the future, we want to conduct further experiments to test this hypothesis.

8 Related Work

8.1 Power-Consumption Side Channels

Power-consumption side channels are exploited, e.g., by the techniques *Simple Power Analysis* (SPA) and *Differential Power Analysis* (DPA). These techniques were first introduced by Kocher, Jaffe, and Jun in 1999 by attacks on smartcards implementing the DES cryptosystem [21]. In both techniques, traces of the power consumption of a circuit are measured and analyzed. For SPA, the traces are directly interpreted, and can lead to revealing the secret key of a device during cryptographic computations [21, 22]. DPA is a statistical method to identify correlations between data processed and power consumption [21,22]. Variations of power analysis have been used in attacks on implementations of cryptographic primitives, e.g., of DES [21, 26, 44], of RSA [21, 22, 34, 39], and of AES [22, 31, 40]. All these attacks obtain traces of a device's power consumption from measurements with dedicated hardware.

Recently, power-consumption side channels were exploited without dedicated hardware [35, 48]. We briefly give an overview on Michalevsky, Schulman, Arumugam, Boneh, and Nakibly's work on tracking Android devices through power analysis [35]. They measure the power consumption of a device using its battery monitoring unit. By the measured power consumption, they can identify known routes, track users in real-time, and identify new routes.

Our work on software-based energy side channels differs from the previously described work on power analysis in the two following aspects.

(a) We investigate a fundamentally weaker attacker model. Our attacker is only able to measure the energy consumption, which is the aggregate of instantaneous power consumption. As a result, the observations required for an attack through software-based energy side channels are more coarse-grained.

(b) On the technical side, we use software-based measurement techniques available on machines without battery, e.g., on desktop and server machines. Software-based techniques allow an attacker to conduct his attack without dedicated hardware and without physical access to the device under attack. Thus, the observations required to exploit software-based energy side channels are easier to obtain than power traces and might be obtainable remotely in the cloud.

Overall, we think that software-based energy side channels are an interesting target for future security research because they are based on more coarse-grained observations that are easier to obtain.

8.2 Quantitative Side-Channel Analysis

Side channels have been the focus of many research projects since their first appearance in Kocher's work in 1996 [20]. A multitude of work focuses on exploiting side channels, e.g., [3, 4, 8, 10, 11, 20, 29, 49]. In addition, analysis of side channels using information-theoretic methods has become an area of focus. Köpf and Basin propose a model to analyze adaptive side-channel attacks using information theory [23]. More concretely, they analyze the attacker's uncertainty of a secret in respect to the number of side-channel measurements the attacker obtained. CacheAudit [16] by Doychev, Feld, Köpf, Mauborgne, and Reineke is a tool employing information theory to give upper bounds of information leakage through cache side channels in x86 binaries. Other work on analysis of side channels using information theory includes [25, 30, 32, 47].

The mentioned works are foremost of analytic nature. On the *empirical* analysis of side channels, we are aware of only a few works, e.g., [14, 33]. Mantel and Starostin evaluate the practical reduction of program transformations to mitigate timing side channels [33]. For their evaluation, they consider the capacity of the timing side channel in a program. They introduce the idea of distinguishing experiments to obtain experimental results on the capacity of timing side channels in benchmark programs.

We apply [33]'s concept of distinguishing experiments to demonstrate that software-based energy side channels exist. Following [33]'s approach, we use the channel capacity of a software-based energy side channel for evaluation of the effects of cross-copying on the software-based energy side channel. In summary, we build on [33]'s techniques, but apply them to a novel kind of side channel.

Our distinguishing test to quantitatively evaluate the weakness in Bouncy Castle RSA is a variant of [13]'s test to distinguish e-passports. Distinguishing e-passports is done through sending a random message and a replayed message to a passport to obtain the difference in response times. Using a normal distribution as model of response times and a manually selected distinguishing point, Chothia and Smirnov calculate the number of observations needed to distinguish passports in 98% of all cases. We transfer the test to our setting. Unlike Chothia and Smirnov, we estimate error probabilities based on offline samples alone, because our observations do not follow a normal distribution.

9 Conclusion

Software-based energy measurement features facilitate the optimization of energy consumption, which is crucial in datacenters. We showed, at the example of Intel RAPL and Bouncy Castle RSA, that these important features also introduce a security issue. Based on only 25 energy samples measured with Intel RAPL, an attacker can distinguish between two RSA keys with 99% success probability.

To counter software-based energy side channels, one could deny untrusted applications access to measurement features. This countermeasure would exclude a large fraction of programs from the optimization of energy consumption. Hence, more fine-grained countermeasures against this vulnerability are desirable. We have investigated the effectiveness of cross-copying, which is a technique for mitigating timing side channels, as a countermeasure against software-based energy side channels and showed that it does not qualify as a general solution. Therefore, it makes sense to look for alternative, more effective countermeasures against software-based energy side channels.

Overall, the challenge is to obtain flexible solutions for energy savings and reliable security guarantees in combination. We have shown that achieving this combination is not straightforward, because software-based energy side channels are a serious security issue. Naturally, our study is only a first step and further investigation of software-based energy side channels is needed.

Acknowledgements We thank Artem Starostin and Ximeng Li for helpful suggestions, respectively, in the initial and final phase of our research project. This work has been funded by the DFG as part of the project Secure Refinement of Cryptographic Algorithms (E3) within the CRC 1119 CROSSING.

References

[1] Power Capping Framework. `https://www.kernel.org/doc/Documentation/power/powercap/powercap.txt`, accessed 2017-04-11

[2] Agat, J.: Transforming out timing leaks. In: POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA. pp. 40–53. ACM (2000)

[3] AlFardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the TLS and DTLS record protocols. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA. pp. 526–540 (2013)

[4] Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., Shacham, H.: On subnormal floating point and abnormal timing. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA. pp. 623–639 (2015)

[5] Arimoto, S.: An algorithm for computing the capacity of arbitrary discrete memoryless channels. IEEE Trans. Information Theory 18(1), 14–20 (1972)

[6] Barthe, G., Rezk, T., Warnier, M.: Preventing timing leaks through transactional branching instructions. Electr. Notes Theor. Comput. Sci. 153(2), 33–55 (2006)

[7] Bellare, M., Rogaway, P.: Optimal asymmetric encryption. In: Santis, A.D. (ed.) Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, Proceedings. Lecture Notes in Computer Science, vol. 950, pp. 92–111. Springer (1994)

[8] Bernstein, D.J.: Cache-timing attacks on aes (2005)

[9] Blahut, R.E.: Computation of channel capacity and rate-distortion functions. IEEE Trans. Information Theory 18(4), 460–473 (1972)

[10] Bortz, A., Boneh, D.: Exposing private information by timing web applications. In: Williamson, C.L., Zurko, M.E., Patel-Schneider, P.F., Shenoy, P.J. (eds.) Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada. pp. 621–628. ACM (2007)

[11] Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: Atluri, V., Díaz, C. (eds.) Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium. Proceedings. Lecture Notes in Computer Science, vol. 6879, pp. 355–371. Springer (2011)

[12] Chatzikokolakis, K., Chothia, T., Guha, A.: Statistical measurement of information leakage. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus. Proceedings. Lecture Notes in Computer Science, vol. 6015, pp. 390–404. Springer (2010)

[13] Chothia, T., Smirnov, V.: A traceability attack against e-passports. In: Sion, R. (ed.) Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6052, pp. 20–34. Springer (2010)

[14] Cock, D., Ge, Q., Murray, T.C., Heiser, G.: The last mile: An empirical study of timing channels on sel4. In: Ahn, G., Yung, M., Li, N. (eds.) Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA. pp. 570–581. ACM (2014)

[15] Cover, T.M., Thomas, J.A.: Elements of information theory (2. ed.). Wiley (2006)

[16] Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. ACM Trans. Inf. Syst. Secur. 18(1), 4:1–4:32 (2015)

[17] Farkas, K.I., Flinn, J., Back, G., Grunwald, D., Anderson, J.M.: Quantifying the energy consumption of a pocket computer and a java virtual machine. In: Brandwajn, A., Kurose, J., Nain, P. (eds.) Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Santa Clara, CA, USA. pp. 252–263. ACM (2000)

[18] Hähnel, M., Döbel, B., Völp, M., Härtig, H.: Measuring energy consumption for short code paths using RAPL. SIGMETRICS Performance Evaluation Review 40(3), 13–17 (2012)

[19] Intel: Intel-64 and IA-32 Architectures Software Developer's Manual. Volume 3 (3A, 3B, & 3C): System Programming Guide (2017)

[20] Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Koblitz, N. (ed.) Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, Proceedings. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer (1996)

[21] Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M.J. (ed.) Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, Proceedings. Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer (1999)

[22] Kocher, P.C., Jaffe, J., Jun, B., Rohatgi, P.: Introduction to differential power analysis. J. Cryptographic Engineering 1(1), 5–27 (2011)

[23] Köpf, B., Basin, D.A.: An information-theoretic model for adaptive side-channel attacks. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA. pp. 286–296. ACM (2007)

[24] Köpf, B., Mantel, H.: Transformational typing and unification for automatically correcting insecure programs. Int. J. Inf. Sec. 6(2-3), 107–131 (2007)

[25] Köpf, B., Smith, G.: Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In: Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom. pp. 44–56. IEEE Computer Society (2010)

[26] Ledig, H., Muller, F., Valette, F.: Enhancing collision attacks. In: Joye, M., Quisquater, J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA. Proceedings. Lecture Notes in Computer Science, vol. 3156, pp. 176–190. Springer (2004)

[27] Legion of the Bouncy Castle Inc.: The Legion of the Bouncy Castle. `https://www.bouncycastle.org/`, accessed 2017-08-14

[28] Linux Programmer's Manual: msr - x86 CPU MSR access device. `http://man7.org/linux/man-pages/man4/msr.4.html` (2009), accessed 2017-04-11

[29] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: Armageddon: Cache attacks on mobile devices. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA. pp. 549–564 (2016)

[30] Macé, F., Standaert, F., Quisquater, J.: Information theoretic evaluation of side-channel resistant logic styles. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, Proceedings. Lecture Notes in Computer Science, vol. 4727, pp. 427–442. Springer (2007)

[31] Mangard, S.: A simple power-analysis (SPA) attack on implementations of the AES key expansion. In: Lee, P.J., Lim, C.H. (eds.) Information Security and Cryptology - ICISC 2002, 5th International Conference Seoul, Korea, Revised Papers. Lecture Notes in Computer Science, vol. 2587, pp. 343–358. Springer (2002)

[32] Mantel, H., Weber, A., Köpf, B.: A Systematic Study of Cache Side Channels across AES Implementations. In: Proceedings of the 9th International Symposium on Engineering Secure Software and Systems. pp. 213–230 (2017)

[33] Mantel, H., Starostin, A.: Transforming out timing leaks, more or less. In: Pernul, G., Ryan, P.Y.A., Weippl, E.R. (eds.) Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9326, pp. 447–467. Springer (2015)

[34] Messerges, T.S., Dabbish, E.A., Sloan, R.H.: Power analysis attacks of modular exponentiation in smartcards. In: Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, Proceedings. Lecture Notes in Computer Science, vol. 1717, pp. 144–157. Springer (1999)

[35] Michalevsky, Y., Schulman, A., Veerapandian, G.A., Boneh, D., Nakibly, G.: Powerspy: Location tracking using mobile device power analysis. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA. pp. 785–800 (2015)

[36] Millen, J.K.: Covert channel capacity. In: Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA. pp. 60–66. IEEE Computer Society (1987)

[37] Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. In: Won, D., Kim, S. (eds.) Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3935, pp. 156–168. Springer (2005)

[38] Noureddine, A., Rouvoy, R., Seinturier, L.: Monitoring energy hotspots in software - energy profiling of software code. Autom. Softw. Eng. 22(3), 291–332 (2015)

[39] Novak, R.: Spa-based adaptive chosen-ciphertext attack on RSA implementation. In: Naccache, D., Paillier, P. (eds.) Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002, Paris, France, Proceedings. Lecture Notes in Computer Science, vol. 2274, pp. 252–262. Springer (2002)

[40] Renauld, M., Standaert, F., Veyrat-Charvillon, N.: Algebraic side-channel attacks on the AES: why time also matters in DPA. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, Proceedings. Lecture Notes in Computer Science, vol. 5747, pp. 97–111. Springer (2009)

[41] Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21(2), 120–126 (1978)

[42] Roberto Tyley: Spongy Castle by rtyley. `https://rtyley.github.io/spongycastle/`, accessed 2017-08-14

[43] RSA Laboratories: PKCS #1 v2.2: RSA Cryptography Standard. `https://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf` (2012), accessed 2017-08-15

[44] Schramm, K., Wollinger, T.J., Paar, C.: A new class of collision attacks and its application to DES. In: Johansson, T. (ed.) Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, Revised Papers. Lecture Notes in Computer Science, vol. 2887, pp. 206–222. Springer (2003)

[45] Shannon, C.E.: A mathematical theory of communication. Mobile Computing and Communications Review 5(1), 3–55 (2001)

[46] Snedecor, G.W., Cochran, W.G.: Statistical Methods (8. ed.). Iowa State University Press (1989)

[47] Standaert, F., Malkin, T., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks. In: Joux, A. (ed.) Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany. Proceedings. Lecture Notes in Computer Science, vol. 5479, pp. 443–461. Springer (2009)

[48] Yan, L., Guo, Y., Chen, X., Mei, H.: A study on power side channels on mobile devices. In: Mei, H., Lü, J., Ma, X., Wang, Q., Yin, G., Liao, X. (eds.) Proceedings of the 7th Asia-Pacific Symposium on Internetware, Internetware 2015, Wuhan, China. pp. 30–38. ACM (2015)

[49] Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: Fu, K., Jung, J. (eds.) Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA. pp. 719–732 (2014)

A RSA Parameters

We list the ciphertext c, the modulus n, and, for each of $k1$ and $k2$, the private exponent d. Table 3 lists the bit length and Hamming weight of the individual key parameters.

Variable	Bit Length	Hamming Weight
n	3071	1550
d_{k1}	2880	1460
d_{k2}	3070	1514

Table 3: RSA parameter information

$$\begin{aligned} c = \;& 21\,444\,858\,737\,899\,529\,054\,620\,511\,370\,454\,507 \\ & 092\,966\,801\,560\,642\,267\,256\,271\,104\,479\,565\,623 \\ & 317\,752 \end{aligned}$$

$n =$ 2 701 439 070 847 831 436 302 643 023 883 472 860
688 598 232 186 843 078 227 336 630 239 028 012
256 550 437 650 268 769 791 198 665 992 795 439
484 217 556 231 560 025 070 371 698 339 396 459
200 881 954 828 050 340 830 157 513 508 421 214
770 279 402 829 167 697 307 613 566 394 176 659
624 110 756 710 628 073 014 761 357 607 996 466
364 229 898 558 058 073 647 928 107 882 490 406
530 947 890 797 815 573 279 825 845 151 878 854
668 533 049 684 979 849 046 263 217 739 454 991
182 947 451 853 315 650 216 590 304 861 483 322
060 060 830 631 094 083 537 687 041 942 037 690
007 693 207 305 415 195 214 688 380 836 084 216
172 144 792 635 213 107 935 419 683 137 307 723
939 160 685 162 963 798 575 432 937 877 504 919
069 927 206 463 822 812 215 130 775 583 846 864
507 114 293 297 396 044 572 999 463 005 723 946
293 357 342 314 317 073 651 823 518 140 604 749
430 721 177 242 193 915 300 702 995 100 318 209
072 680 035 930 026 760 088 409 999 868 552 738
596 292 995 373 879 363 788 033 672 926 557 820
859 907 396 638 610 163 158 192 481 639 061 519
053 725 943 865 537 221 937 014 172 943 369 946
317 527 944 500 414 286 628 781 268 545 323 413
089 483 205 130 985 579 709 706 141 004 772 358
028 235 835 383 909 088 091 781

$d_{k1} =$ 834 165 241 298 999 430 572 239 556 741 255 001
409 654 369 991 231 022 229 220 766 012 080 697
463 656 309 174 093 432 158 675 603 340 216 003
665 704 131 245 121 040 967 995 188 366 594 646
886 723 499 562 164 775 785 136 008 896 297 468
405 676 356 520 936 826 945 820 428 827 348 255
217 929 032 541 402 713 897 358 199 944 878 768
362 082 394 995 264 828 906 821 922 160 081 896
178 733 905 626 880 183 545 477 730 549 240 816
967 899 639 830 638 962 585 672 589 316 902 773

646 421 798 550 172 445 107 122 780 716 202 671
225 380 537 248 843 847 787 001 886 230 297 573
272 017 826 827 441 391 799 971 383 481 609 479
693 434 609 255 364 781 237 298 674 935 211 620
000 100 041 121 931 493 922 732 461 726 369 423
008 396 966 929 501 865 211 495 345 778 306 377
790 415 705 746 828 081 157 687 854 396 051 014
887 511 709 430 472 332 036 102 915 852 198 291
900 816 398 410 487 823 293 583 922 839 328 518
348 451 707 669 403 333 993 535 972 295 702 111
655 470 282 959 323 284 437 483 178 409 938 904
891 941 353 380 152 662 307 486 605 772 459 905
400 151 595 208 101 373 686 515 401 901 692 964
058 539 933 630 431 256 790 357 003 951 566 054
871

d_{k2}= 849 669 096 348 419 204 365 570 298 477 349 071
171 614 131 865 471 357 729 223 033 692 678 706
938 741 080 172 802 999 095 258 832 447 464 674
826 253 513 078 126 047 832 149 347 969 391 019
019 909 054 959 345 128 332 576 053 617 789 744
725 266 175 298 192 375 980 008 826 221 571 989
636 873 751 134 110 143 415 982 969 381 778 707
618 076 367 532 496 926 501 132 827 071 452 381
857 918 868 318 894 249 233 517 709 784 025 494
473 083 475 794 688 338 318 669 205 292 634 477
215 223 397 852 394 761 705 823 824 009 487 094
582 053 403 448 414 519 187 059 874 506 785 829
441 820 347 012 931 983 749 032 937 029 535 204
674 669 118 349 387 871 614 945 298 028 125 580
430 251 234 668 630 080 219 358 718 245 352 291
415 465 763 013 100 923 209 592 436 665 013 250
115 828 673 733 662 998 810 262 212 481 440 283
643 807 643 936 814 117 781 430 012 258 146 460
658 672 860 115 805 136 484 154 272 106 257 859
724 501 287 380 315 081 559 737 344 179 353 409
746 394 603 117 859 928 408 887 186 955 223 875
953 551 569 984 766 380 086 437 972 232 285 448

676 372 452 773 194 118 503 147 494 678 742 399
709 855 779 414 952 984 145 813 209 160 450 714
556 753 389 051 248 506 613 925 218 229 813 615
602 923 271 485 462 745 822 621

B Java Sources for Target Programs

```java
public RSA(/*[...]*/) {
    rsa = new RSAEngine();
    RSAKeyParameters key;
    //[...]
    key = new RSAKeyParameters(true, new BigInteger(n), new
    BigInteger(d));
    //[...]
    rsa.init(false, key);
}

public void decrypt() {
    rsa.processBlock(input, 0, input.length);
}

private RSAEngine rsa;
private RSAKeyParameters key;
private byte[] input;
```

Fig. 13: RSA decryption sources

```java
private final static int n = 4096;
private int r;
// [...]
public int modExp(int y, int k) {
    r = 1;
    for (int i = 0; i < 32; i++) {
        if (k % 2 == 1)
            r = (r * y) % n;
        y = (y * y) % n;
        k >>=1;
    }
    return r % n;
}
```

Fig. 14: [33]'s baseline modExp sources

```java
private final static int n = 4096;
private int r;
private int rSkip;
// [...]
public int modExpCC(int y, int k) {
    r = 1;
    rSkip = 1;
    for (int i = 0; i < 32; i++) {
        if (k % 2 == 1)
            r = (r * y) % n;
        else
            rSkip = (rSkip * y) % n;
        y = (y * y) % n;
        k >>=1;
    }
    return r % n;
}
```

Fig. 15: [33]'s cross-copied modExp sources