An Evaluation of Bucketing in Systems with Non-deterministic Timing Behavior

Yuri Dantas, Richard Gay, Tobias Hamann, Heiko Mantel, Johannes Schickel

An Evaluation of Bucketing in Systems with Non-Deterministic Timing Behavior

Yuri Gil Dantas, Richard Gay, Tobias Hamann,
Heiko Mantel, and Johannes Schickel

Department of Computer Science, TU Darmstadt, Germany
lastname@mais.informatik.tu-darmstadt.de

Abstract. Timing side-channel vulnerabilities constitute a serious threat against privacy and confidentiality of data. In this article, we study the effects of bucketing, a previously proposed mitigation technique against timing side channels. We present two implementations of bucketing that reside at the application and at the kernel level, respectively. We experimentally evaluate the effectiveness of these implementations in a setting with non-deterministic timing behavior, a practically relevant setting that has not been studied before. Our results show that the impact of non-deterministic timing behavior is substantial. The bucket boundaries cannot be established sharply and this reduces the effectiveness of bucketing. Nevertheless, bucketing still provides a significant reduction of side-channel capacity.

1 Introduction

In a side-channel attack, an adversary exploits execution characteristics of a program to deduce secret information. Timing behavior, energy consumption, EM radiation are execution characteristics on which side-channel attacks can be based. By exploiting side-channel vulnerabilities, attacks have been able to deduce sensitive information like e.g., cryptographic keys. For instance, timing attacks that are able to recover the full secret key from AES [3] and RSA [12] implementations have been developed. Timing attacks have also been developed against web applications [2,4].

Multiple techniques against timing side-channel attacks have been proposed like, e.g., predictive mitigation [22], unification [15], or cross-copying [1]. This article focuses on the bucketing technique [14]. Bucketing is the discretization of a program's execution times such that the results of each program run are returned at only a fixed number of points in time (buckets). Security-wise, bucketing aims to reduce the amount of information that an adversary can learn for a secret by reducing the possible number of observations. Performance-wise, bucketing allows one to navigate in a flexible fashion in the trade-off between the security provided and the performance overhead cost.

Bucketing has been evaluated in [14,16,8,21]. The bucketing technique has been proposed in [14], along with a theoretical bound on the amount of information about the secret key that can be extracted from a timing-channel. The

effectiveness of bucketing for a cryptographic implementation of modular exponentiation was demonstrated using this bound. In subsequent studies, it has been shown that guarantees on semantic security of cryptosystems remain valid when applying bucketing to mitigate timing side channels [16]. The same work also provided a tighter bound on the leakage of timing channels. In [8], a systematic approach is presented in order to determine the optimal instantiation of bucketing. Recently, an even tighter bound for timing-channel leakage has been proposed, that is about the square root of the previously considered bounds [21].

So far, the technique of bucketing has been studied for systems with deterministic timing behavior [14]. In such systems, each given input always leads to the same timing observation. In this article, we investigate a complementary setting, namely the effect of bucketing in systems with non-deterministic timing behavior. A better understanding for this setting is needed for clarifying the effects of bucketing in a non-deterministic setting.

Bucketing can be implemented at different levels of a layered system architecture. In our research project, we developed two implementations of bucketing: one implementation at the application level, and another one inside the operating-system kernel for reducing timing side channels in Java programs. Based on these implementations, we evaluate the effectiveness of bucketing in a non-deterministic setting. More concretely, we evaluate the effectiveness of bucketing in terms of leakage bounds and reduction of side-channel capacity. Moreover, we evaluate the efficiency of our implementations.

Our results indicate that our implementations are not able to release events instantaneously at the bucket boundary, but rather within a certain interval after the bucket boundary. This leads to a large number of possible observations that an adversary can make, increasing the leakage bound on information an adversary can learn about the secret. We provide empirical estimations of the side-channel capacity to show that our implementations can significantly reduce the channel capacity by 84% (application level) and 78% (kernel level). These numbers are similar to the reduction achieved by other established techniques for mitigating timing side channels like e.g., cross-copying, as shown in [17].

The remainder of the article is structured as follows. Section 2 briefly clarifies the concept of timing side channels. Section 3 presents an overview on bucketing. Section 4 discusses the design space options for bucketing implementations and presents implementations at both application and kernel level. Section 5 provides details on our empirical evaluation of bucketing for systems with non-deterministic timing behavior. After a discussion of the related work in Section 6, we conclude in Section 7.
All machinery needed to reproduce our results are publicly available.[1]

2 Timing Side Channels

Timing side channels go back to Kocher's seminal work on timing attacks [13]. In a timing-side channel attack, an adversary exploits correlations between the

[1] `http://www.mais.informatik.tu-darmstadt.de/assets/bucketing/machinery.tar.gz`

execution time of a program and the secrets that the program processes. By gathering multiple timing observations, such an adversary can learn information about these secrets, and, in the worst-case, extract all secrets.

In RSA, modular exponentiation (modExp for short) operation has been a classic example of a timing side-channel vulnerability. Consider for instance the implementation in Figure 1, which can be used to implement encryption and decryption operations of the RSA encryption scheme [19]. For decryption in RSA one computes $p = c^d \pmod{n}$, where c is the ciphertext, d is the private exponent, and n is the modulus.

The timing behavior of modExp reveals the Hamming weight of the private exponent. The private exponent d is processed by modExp bit by bit. When a bit is set (Line 4), modExp performs an additional multiplication and mod operation (Line 5). Execution of these instructions takes additional time. Thus, an adversary can learn the Hamming weight of the private exponent by performing multiple timing observations. Knowing the Hamming weight of the private exponent might be sufficient for making a brute force attack (i.e. by trying out all possible private exponent values with that Hamming weight) feasible.

```
1  public int modExp(int c, int d) {
2    int r = 1;
3    for (int i = 0; i < this.keySize; i++) {
4      if (d % 2 == 1)
5        r = (r * c) % n;
6      c = (c * c) % n;
7      d >>= 1; }
8    return r % n;}
```

Fig. 1: Implementation of modExp containing a timing side-channel vulnerability

3 Bucketing

Bucketing is a technique against timing side-channel attacks. The approach of bucketing is to discretize the timing behavior of a program by grouping different execution times into so called buckets. More concretely, bucketing allows only a set of possible execution times by delaying program executions to a set of observable times, the so-called bucket boundaries. For each event that occurs within the limits of a given bucket, the event is delayed until the boundary of this corresponding bucket. Note that this delay inherently causes a performance penalty. What one gains in exchange is a reduction of the amount of information that can be leaked through a timing side channel.

Bucketing has been initially studied as mitigation technique in systems with deterministic timing behavior [14]. That is, each given input always leads to the same timing observation. Thus, bucketing can reduce the number of timing observations to the number of defined buckets. The same article shows that bucketing in combination with input blinding[2] is effective in mitigating timing side channels in cryptographic implementations.

[2] Blinding [13] is a technique that decorrelates the messages from the decryption times.

In contrast to constant-time implementations like, for instance, unification [15], cross-copying [1], conditional assignment [18], or transactional branching [9], the goal of bucketing is not to completely mitigate timing side channels. Rather, bucketing bounds the information that an adversary can learn for a secret by reducing the possible number of observations he can make. This allows an adversary to infer some information about the secret, but at the same time reduces the performance overhead caused by the mitigation technique. A careful choice of the bound of information that is allowed to be leaked, however, can prevent adversaries from learning the complete key.

It has been shown that the amount of information that is leaked to an adversary through a timing side channel can be bounded by

$$log_2 \binom{\mathsf{n} + \mathsf{O} - 1}{\mathsf{n}} \tag{1}$$

where O denotes the number of different timing observations and n is the amount of measurements the adversary performed [16]. A bigger value of O increases the bound on leaked bits. Conversely, reducing O decreases the bound of leaked bits.

Example. To showcase the leakage reduction that can be achieved by bucketing, consider a hypothetical implementation of a crypto-algorithm with a 512-bit secret key. Assume a flawed implementation that leaks the Hamming weight of the secret key (e.g., the `modExp` implementation from Figure 1), and deterministic timing behavior. With these assumptions, an adversary of the non-mitigated program can at most make 512 different observations, depending on the Hamming weight of the key. Instantiating Expression (1) for this example and 153 measurements by an adversary, we get a bound on the leakage of 512 bits.

This means that, in the worst case, an adversary can determine the whole key within 153 measurements. When applying an instantiation of Expression (1) with three buckets for this example, we get a bound on the leakage of 14 bits. This means that, in order to learn the complete key, an adversary needs at least 1.64×10^{77} measurements. This illustrates how effective bucketing can be in a setting with deterministic timing behavior.

4 Implementations of Bucketing Mechanisms

This section presents two implementations of bucketing: one at the application level and one inside the operating-system kernel. These implementations meet design goals concerned with the effectiveness and efficiency of the mechanism. We also evaluate the design space w.r.t. the choice of a system level, the choice of a delay mechanism and the handling of events occurring outside the last bucket.

4.1 Terminologies and Design Goals

An implementation of bucketing shall facilitate an *effective* and *efficient* reduction of timing side channels in the target program. By effectiveness, we refer to the ability of the mechanism to reduce the number of possible observations to a limited amount of points in time. By efficiency, we refer to the overhead introduced to the target program by the mechanism.

For systems with non-deterministic timing behavior, releasing events instantaneously at the corresponding bucket boundary might not be possible, e.g., due to the non-deterministic behavior of garbage collection or just-in-time compilation in Java programs. In this case, the event will rather be delayed within a certain interval of possible observations after the bucket boundary. We refer to the width of this interval as the *precision* of the mechanism, and to the distance between the intended bucket boundary and the mean release time of our mechanism as the *bias* of the implementation. A high precision and thus a small release interval is the central aspect of the effectiveness of the mechanism.

Regarding user acceptance of security mechanisms, effectiveness alone is not sufficient. The mechanism should rather also be as transparent to end users as possible. Hence, the runtime overhead added by the mechanisms shall be as low as possible while still enforcing the desired security properties. The overhead added by a bucketing implementation has two aspects: the overhead added by the delay of events until the next bucket boundary, and effects that are induced by the implementation. The first aspect is mainly affected by the choice of bucket sizes and boundaries, and is thus induced by the mitigation technique itself. Previous work on bucketing has covered the overhead added by the mitigation technique itself [14], and how bucketing can be instantiated with minimal performance overheads [8]. In this article, we focus on the overhead added by our actual implementation. This overhead, in turn, has two main aspects. Firstly, an implementation of bucketing can add a *general overhead* to the program, for instance due to initialization steps of the mechanism. Secondly, the bias of the implementation directly adds to the perceived overhead of the mechanism.

A generic security mechanism shall be applicable to a wide range of target programs. In this work, our focus is on bucketing for Java applications. Hence, an implementation of bucketing shall abstract from the target program as much as possible, such that it can be applied to generic programs that include a timing vulnerability. However, a generic approach that can be used for a variety of programming languages can be preferable to language-specific implementations.

In summary, we identify three main design goals for an implementation of bucketing in practice: high precision, low overhead (including bias), and applicability to a wide range of programs.

4.2 Design Space

Choice of System Level. One central consideration of the implementation of the security mechanism is the system level where the mechanism is implemented. Security mechanisms can be placed directly at the kernel level of the operating-system, at the application level where the target programs of the mechanism reside, or in an intermediary middleware level that is specific to the intended application domain of target programs.

Placing our implementation of bucketing in an intermediary middleware level does not meet our design goal of applicability to a wide range of target programs. Specific middleware levels can differ between different systems, and would thus

require specialized implementations for each middleware level to be supported. In the following, we focus on the application and kernel level.

Regarding the three design goals identified in Section 4.1, we see both alternatives fitting for an implementation of bucketing. An implementation in the kernel level offers the advantage of applicability to target programs in different programming languages, as long as they can interact with the system level. An implementation at application level facilitates a better portability between systems running different operating systems and requires no modification of the underlying system. We implement bucketing for Java programs at application level using the runtime enforcement framework CliSeAu [10]. We also implement bucketing at kernel level as a reference to validate our findings.

Delay Mechanisms. Regarding the precision of a bucketing implementation, it is vital to achieve a high precision of the delay mechanism that is used for delaying events. We have considered two alternative techniques for delaying events: sleep mechanisms and busy-waiting loops.

We evaluated the precision of these techniques at both application and kernel level. For each system level, we performed 2^{19} measurements to assess the precision of sleep and busy-waiting. On the application level, busy-waiting is on average 199.55% more precise than sleep mechanisms. On the kernel level, the precision gain of busy-waiting is on average 199.85%. For this reason, we favored busy-waiting over sleep technique in our implementations.

Handling Events Outside the Last Bucket. The theory of bucketing assumes knowledge about the worst-case execution time (WCET) of a program. In practice, knowledge about this time is usually not given. In a setting with non-deterministic timing behavior, a clear boundary for WCET does not exist in general. Effects like e.g., scheduling affect the execution time of a program.

This drawback leads to the question how the last bucket boundary shall be chosen in practice, and how events that are observed outside this last bucket shall be handled by the mechanism. We leave the choice of the last bucket boundary underspecified, as this might differ between system environments or specific requirements on the mechanism for a given target program. Thus, the last bucket boundary can be instantiated by the end user of the mechanism. Regarding the aspect of handling events outside the last bucket, we see two alternatives: such events can either be released directly when they are observed, or they can be dropped completely. Releasing events directly when they are observed by the mechanism might introduce additional observations to an adversary. This can thus lead to additional leakage. Completely dropping events, however, might be unacceptable for end users of the mitigated application. We thus choose to release such events immediately for our implementations. However, adapting this behavior is straightforward and can be done easily in both implementations.

4.3 Application Level Implementation

We implement bucketing at the application level using the runtime enforcement framework CliSeAu [10]. CliSeAu is a generic framework used to harden Java

programs by dynamically enforcing security requirements. CliSeAu has a modular architecture consisting of four components: interceptor, local policy, enforcer and coordinator. For our implementation of bucketing, we focus on the interceptor and the enforcer. The interceptor component is responsible for intercepting attempts of the target program to perform security-relevant events and the enforcer component enforces the decided countermeasures on the target program. For more details about CliSeAu's architecture, we refer the interested reader to [10]. The use of a generic enforcement framework enables the support for a wide range of target programs. To instantiate our mechanism, the signature of the sensitive methods and the sizes of the buckets are provided to the framework.

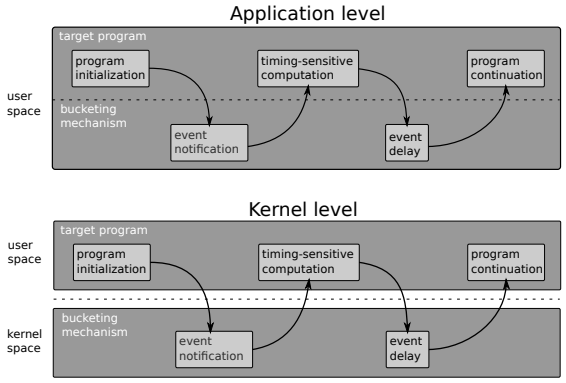

Fig. 2: High-level workflow of our bucketing implementations

The overall workflow of our implementation is depicted on the top of Figure 2. After program initialization, but before starting the timing-sensitive computation, the mechanism is notified about the start of the computation. The timing-sensitive computation is then executed, notifying the mechanism once it has been finished. The mechanism delays the event until the next bucket boundary is reached, continuing regular program execution afterwards.

The event notification step before the timing-sensitive computation is performed in CliSeAu by the interceptor component. To instantiate this interceptor component, users of the mechanism provide the method signatures of timing-sensitive computations. The event delay is performed in CliSeAu by the enforcer component. To instantiate this enforcer component, users of the mechanism provide the amount of buckets to be enforced, and their corresponding boundary times. Using these two components, the CliSeAu encapsulation process modifies the target program by inlining the bucketing mechanism to the target program. Following our design decision from Section 4.2, events outside the last bucket are released immediately without delay.

Instantiating our mechanism for a specific target program involves two aspects: the specification of the monitored methods in the target program, and the instantiation of the parameters for bucket boundaries. The first aspect is given in the form of AspectJ pointcuts, while the second aspect is implemented as instantiation of an EnforcerFactory in CliSeAu with an array containing the bucket boundaries to be enforced. In practice, an instantiation of our mechanism can be achieved in 12 LOC.

4.4 Kernel Level Implementation

Our bucketing implementation in the Linux kernel provides an interface for user space target programs to incorporate bucketing in timing-sensitive computations. The Linux kernel provides multiple interfaces that can be used from application level. Among these, regular system calls and so called virtual dynamically shared objects (vDSO) are the most interesting ones for our implementation. System calls provide developers of user space programs with the ability to interact with the kernel. However, system calls triggers a context switch from user space to kernel space, where the functionality implemented by the system call is executed. The need for this context switch can be avoided by using vDSO calls. The vDSO mechanism maps kernel methods into user space contexts, such that they can be executed directly in user space without the overhead of a context switch[3]. For this reason, we implement bucketing at the kernel level using vDSO calls.

The overall workflow of bucketing in the Linux kernel is depicted on the bottom of Figure 2. Both the event notification step and the event delay are performed as vDSO calls in the kernel. The bucketing mechanism is not inlined when using our kernel level implementation. Users of our mechanism include the vDSO calls to the mechanism directly in their target programs code[4]. For this, the event notification call is executed directly before starting the timing-sensitive computation. The call returns the current time stamp inside the Linux kernel, which is stored in a local variable in the target program. After executing the timing-sensitive computation, the event release call is executed, providing the returned initial time from the notification step. The kernel implementation then delays the event before returning to the target program. Events outside the last bucket are released immediately without delay.

Similar to our implementation at the application level, instantiating our mechanism for as specific target program involves two aspects: calling the event notification method does not involve any parameters and will only initialize our mechanism by returning the time from the kernel. When calling the event delay method, the number of buckets, their corresponding boundaries, and the initial time from the notification step are provided as arguments for the call. Hence, including the definition of a local variable for the initial time, our mechanism is instantiated with 3 LOC inside the target program.

5 Evaluation

Our empirical evaluation investigates the effectiveness of bucketing in a non-deterministic setting in terms of leakage bounds and reduction of side-channel capacity. Our evaluation compares our implementations of bucketing in terms of precision, bias and overhead.

Evaluation Setup and Metrics. All of our experiments are carried out on a desktop machine, Intel i5 3.3GHz with 4GB of memory, running Ubuntu 14.04 with kernel 4.9.18 and using OpenJDK 8.

[3] cf. Linux man-pages: `http://man7.org/linux/man-pages/man7/vdso.7.html`

[4] For Java programs, the whole procedure can be done via JNI calls.

In our experiments, we consider a vulnerable implementation of modExp (see Section 2) that can be used in RSA operations. We simulate blinding by randomizing each message before modExp. We assume a local adversary who measures the execution time of modExp using the maximum precision measurements provided by the JVM, System.nanoTime(). Following common practices by [11], our measurements consist of two phases: start-up and experimental. In both phases, we perform 2^{19} timing measurements. Note that only the results obtained in the experimental phase are considered in this article, as these measurements relate to the steady-state of modExp. For each bucket boundary, we reject outliers that lie further than three median absolute deviations from the median.

In contrast to systems with deterministic timing behavior, a program running in a system with non-deterministic timing behavior can have different execution times for the same input. Thus, we conduct multiple samples to evaluate the practical impact of our results. For the sake of space, we present three specific samples in the following sections. To evaluate the effectiveness of our implementations in reducing timing side-channels, we consider the worst-case reduction observed. Whereas to evaluate our implementations in terms of precision, bias and overhead, we consider their mean values.
In our experiments, we measure the following metrics:

- **Number of Timing Observations (O)**: The number of different timing observations (value-wise) an adversary can gather after performing timing measurements on the program.
- **Channel Capacity (CC)**: The estimation of the amount of information (in bits) leaked from the timing channel.
- **Average Response Time ($\mathsf{T}_{\mathtt{resp}}$)**: The average time a user whose request was processed by the program has to wait between the time that he sent the request and the time that he obtained the response.

5.1 Empirical Results for the Leakage Bounds

We evaluate how much bucketing can reduce the amount of bits leaked by reducing O. We measure the number of different timing observations an adversary can gather after performing 2^{19} timing measurements on modExp. We compute the bound on leaked bits presented in Section 3. As a result, we obtain a bound on how many bits an adversary can learn in this setting.

Experimental Design. We conduct experiments in two scenarios: with and without bucketing. For all experiments, we use a static RSA 1024-bit key with Hamming weight 700. When using bucketing, we instantiate our implementations with four buckets. For this, we chose the first three buckets equidistantly to each other and the last bucket as the estimated worst case execution time of modExp. Finally, we compute Expression 1 with our results.

Experimental Results. Three samples of our results are described in Figure 3. One of the samples is depicted in Figure 4. NO-BUCKETING represents our results without bucketing, whereas $\text{BUCKETING}_{\text{APPL}}$ and $\text{BUCKETING}_{\text{KRNL}}$ represent our results when using bucketing at application and kernel level, respectively.

Bucketing reduces O by 99%. Conversely, O is much larger when applying bucketing in systems with non-deterministic timing behavior. While bucketing in systems with deterministic timing behavior can reduce O to the number of defined buckets, in our case 4, BUCKETING$_{\text{APPL}}$ and BUCKETING$_{\text{KRNL}}$ reduce O to 1461 and 3737, respectively. Thus, according to Expression 1, an adversary might be able to obtain the entire key after performing 2^{19} timing measurements.

scenario	**sample 1**			**sample 2**			**sample 3**		
	O	**bound**	**reduction**	**O**	**bound**	**reduction**	**O**	**bound**	**reduction**
no-bucketing	374341	880494	-	365596	933315	-	417749	933315	-
bucketing$_{\text{APPL}}$	846	9053	99.77%	1162	11909	99.68%	1461	14495	99.65%
bucketing$_{\text{KRNL}}$	2429	22331	99.35%	2842	25489	99.22%	3737	32049	99.10%

Fig. 3: Reduction of timing observations and leakage bounds

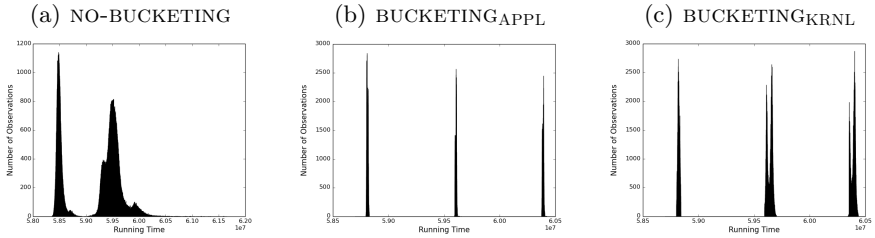

Fig. 4: Timing distributions with and without bucketing

An important question for these results is to find the cause for this large number of observations in our implementations. By taking a closer look at our results, we notice that our implementations are not precise when releasing events. That is, our implementations release events within a certain interval after the bucket boundary, as illustrated in Figures 3(b) and 3(c). This substantiates our claim, discussed in Section 4.1, that releasing events instantaneously at the bucket boundary seems not to be possible in a non-deterministic setting. A possible explanation for this effect is that activities in the CPU, e.g., scheduling and system load, can cause a latency in the response time of programs.

As a result of this imprecision, an adversary can gather a large number of observations from the program even when bucketing is applied[5]. A possible conclusion to be drawn is that bucketing is not effective in reducing timing side channels in systems with non-deterministic timing behavior. This conclusion, however, seems to be premature because the large number of observations can be caused by the properties described above.

In the following section, we investigate more closely how much information is actually leaked by our implementations of bucketing. In contrast to leakage bounds, which provides the worst case leakage that could possibly arise, this estimation has more practical significance with regard to our implementations.

[5] Omitted here for the sake of space, we also evaluated that keys with other Hamming weight (other than 700) also led to a similarly large number of observations.

5.2 Empirical Assessment of Key Indistinguishability

We estimate the reduction of the timing side-channel capacity in modExp achieved by bucketing in isolation. By isolation, we refer to an instantiation of a 1-bucketing. As in [17], we model a timing side channel as a discrete information-theoretic channel [7] with input X and output Y. The input alphabet of the channel models the space of secret inputs to a program and the output alphabet models possible timing observations. We measure the correlation between the secret inputs and possible timing observations with the Shannon's channel capacity [20], denoted $C(X;Y)$. We statistically estimate [5] the channel capacity $C(X;Y)$ from empirically collected timing observations. As a result, we compute the percentage reduction of CC achieved by our implementations of bucketing.

Experimental Design. Following the experimental design from [17], we generate two keys, namely key_1 and key_2, with different Hamming weights. The purpose of this setup is to evaluate the reduction of the timing side-channel capacity for keys with different execution times. For each of the keys, we carry out experiments with and without bucketing. When using bucketing, we instantiate a 1-bucketing (with the same bucket size for both keys). Finally, we compute the CC with the help of the leakiEst tool [6].

Experimental Results. Three samples of our results are described in Figure 5. One of the samples is depicted in Figure 6. NO-BUCKETING represents our results without bucketing, whereas $\text{BUCKETING}_{\text{APPL}}$ and $\text{BUCKETING}_{\text{KRNL}}$ represent our results when using bucketing at application and kernel level, respectively.

scenario	sample 1		sample 2		sample 3	
	CC	**reduction**	**CC**	**reduction**	**CC**	**reduction**
no-bucketing	0.4834±0.0006	-	0.4636±0.0005	-	0.4874±0.0004	-
bucketing$_{\text{APPL}}$	0.0767±0.0015	84.14%	0.0706±0.0008	84.77%	0.0733±0.0008	84.96%
bucketing$_{\text{KRNL}}$	0.063±0.0008	86.96%	0.0857±0.0009	81.51%	0.1027±0.0010	78.92%

Fig. 5: Estimated capacity of timing side channels with 95% confidence intervals

(a) NO-BUCKETING (b) $\text{BUCKETING}_{\text{APPL}}$ (c) $\text{BUCKETING}_{\text{KRNL}}$

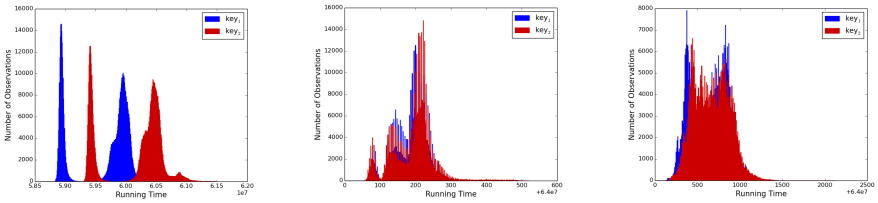

Fig. 6: Correlation between two secret keys and their timing distributions

In Figure 5(a), we can visually see that the timing distributions of key_1 and key_2 are not overlapping. Thus, the fact whether modExp has received key_1 or key_2 can be leaked via a timing side channel. In contrast, when using bucketing (see Figures 5(b) and 5(c)), the timing distributions of key_1 and key_2 are mostly

overlapping. This gives us a first hint that our implementations of bucketing were effective in removing the timing side channel in modExp.

The table in Figure 5 summarizes the results of estimating the capacity of the timing side channel using the leakiEst tool. Since we are using two different keys, the capacity of the timing side channel is 1 bit, as an adversary can at most learn one bit of information: Whether key_1 has been used or whether key_2 has been used. While for systems with deterministic timing-behavior, bucketing can reduce the capacity of timing side channels by 100%[6], our results indicate that BUCKETING$_{\text{APPL}}$ and BUCKETING$_{\text{KRNL}}$ can reduce the capacity of timing side channels by roughly 84% and 78%, respectively.

The estimated capacity of timing side channels achieved by our implementations are in the range of the ones reported in [17] for program transformation techniques. Furthermore, similar to bucketing for systems with deterministic timing behavior, our implementations significantly reduced the timing side-channel capacity. Hence, based on our results, we believe that bucketing is also effective in reducing timing side channels in systems with non-deterministic timing behavior, despite the large number of possible timing observations.

5.3 Empirical Comparison of Our Implementations

We compare our implementations in terms of precision, bias and overhead (so called general overhead in Section 4.1). Our results are described in Figure 7.

scenario	precision	bias	overhead
bucketing$_{\text{APPL}}$	[106, 488] ns	201.7 ns	383.7μs (1%)
bucketing$_{\text{KRNL}}$	[167, 1624] ns	799.8 ns	343.4μs (1%)

Fig. 7: Empirical comparison of our implementations

BUCKETING$_{\text{APPL}}$ is roughly three times more precise than BUCKETING$_{\text{KRNL}}$ when releasing events. This difference directly affects the number of timing observations one can gather from the program. BUCKETING$_{\text{APPL}}$ is also more efficient when releasing events. While BUCKETING$_{\text{APPL}}$ has a bias of 201.7 ns, BUCKETING$_{\text{KRNL}}$ has a bias of 799.8 ns. Thus, on average, BUCKETING$_{\text{KRNL}}$ releases events four times slower than BUCKETING$_{\text{APPL}}$. In both cases, we believe that the use of JNI calls to enforce bucketing at the kernel level was the reason for such a difference. On the other hand, BUCKETING$_{\text{APPL}}$ added slightly ($\sim$40μs) more overhead than BUCKETING$_{\text{KRNL}}$. The reason why this happened is unclear to us at this moment.

6 Related Work

Program transformation mechanisms like unification [15], cross-copying [1], or conditional assignment [18] aim to completely mitigate timing side channels introduced by critical conditionals (i.e., conditionals whose timing behavior is

[6] This is derived from the assumption that bucketing can reduce the number of observations to the number of defined buckets.

directly affected by the value of a secret). In this constant-time approach, the program is modified such that critical conditionals take the same execution time for all secret inputs. Thus, each mitigated program run takes the WCET of the unmitigated program. To reduce the runtime overhead of constant-time mitigation, several approaches that incorporate a tradeoff between security guarantees and runtime overhead have been proposed. Examples for such approaches include bucketing [14] and predictive mitigation of timing channels [22]. *Predictive mitigation* of timing channels, as proposed in [22], offers a tradeoff between security and performance by using predicted schedules for events. If the predicted schedule is met for event observations by the mitigated program, events are delayed according to the current schedule, as this does not provide information to an adversary. If the schedule is violated, however, the schedule is adapted dynamically and events are delayed to meet this adapted schedule. The tradeoff between security and overhead can be chosen by selecting a tailored adaptation strategy for the schedule, which is called penalty policy.

In contrast to existing work in the area of bucketing, we are the first to provide an empirical evaluation of the effectiveness of bucketing in systems with non-deterministic timing behavior. Previously, the effectiveness of bucketing has been evaluated based on leakage bounds in systems with deterministic timing behavior. We do not consider optimal choices of the instantiation of bucketing, as we are interested in the effectiveness of our implementation rather than efficient choices in the security-performance tradeoff. We are not the first to evaluate the effectiveness of timing side-channel mitigation techniques empirically in general. For instance, [17] provides an empirical evaluation of different program transformations in Java programs. Their evaluation compares the effectiveness and efficiency of different program transformations, enabling developers to choose a fitting transformation for their security requirements.

7 Conclusion

This article investigated the impact of non-deterministic timing behavior on bucketing. Our results show that the impact is substantial. Our bucketing implementations are not able to release events sharply at the bucket boundary, but rather within a certain interval after the bucket boundary. This leads to a large number of possible observations that an adversary can make, increasing the leakage bound on information an adversary can learn about the secret. Nevertheless, we provided empirical estimations of the side channel capacity to show that our implementations can reduce the channel capacity by roughly 84% (application level) and 78% (kernel level). These numbers are similar to the reduction achieved by other established techniques for mitigating timing side channels like e.g., cross-copying, as shown in [17]. Based on these results, we believe that the large number of observations can be caused by activities in the CPU, e.g., scheduling. This observation indicates that future work towards tighter leakage bounds for non-deterministic timing behavior are desirable. A tighter bound for timing channel leakage has been recently proposed [21]. Computing this bound with our results seems to be computationally expensive. Thus, experiments using this bound as well as evaluation of other algorithms are left to future work.

In this paper, we studied the effectiveness of bucketing as a countermeasure against timing side channels. Whether bucketing opens other possibilities for attacks was outside scope. This might be a direction for future work, as other security mechanisms have been exploited by attackers to mount attacks.

Acknowledgment. This work was funded by the DFG as part of project Secure Refinement of Cryptographic Algorithms (E3) in CRC 1119 CROSSING.

References

1. J. Agat. Transforming out Timing Leaks. In *POPL'00*, pages 40–53, 2000.
2. M. R. Albrecht and K. G. Paterson. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. In *EUROCRYPT'16*, pages 622–643, 2016.
3. D. J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.
4. A. Bortz and D. Boneh. Exposing Private Information by Timing Web Applications. In *WWW'07*, pages 621–628, 2007.
5. K. Chatzikokolakis, T. Chothia, and A. Guha. Statistical measurement of information leakage. In *TACAS'10*, pages 390–404, 2010.
6. T. Chothia, Y. Kawamoto, and C. Novakovic. A Tool for Estimating Information Leakage. In *CAV'13*, pages 690–695, 2013.
7. T. M. Cover and J. A. Thomas. *Elements of Information Theory, 2. ed.*
8. G. Doychev and B. Köpf. Rational Protection against Timing Attacks. In *CSF'15*, pages 526–536, 2015.
9. T. Rezk G. Barthe and M. Warnier. Preventing Timing Leaks Through Transactional Branching Instructions. *ENTCS'06*, 153(2):33–55, 2006.
10. R. Gay, J. Hu, and H. Mantel. CliSeAu: Securing Distributed Java Programs by Cooperative Dynamic Enforcement. In *ICISS'14*, pages 378–398. Springer, 2014.
11. A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *OOPSLA'07*, pages 57–76, 2007.
12. M. S. Inci, B. Gülmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES'16*, pages 368–388, 2016.
13. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO'96*, pages 104–113, 1996.
14. B. Köpf and M. Dürmuth. A Provably Secure and Efficient Countermeasure against Timing Attacks. In *CSF'09*, pages 324–335, 2009.
15. B. Köpf and H. Mantel. Transformational Typing and Unification for Automatically Correcting Insecure Programs. *Int. J. Inf. Sec.*, 6(2–3):107–131, 2007.
16. B. Köpf and G. Smith. Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In *CSF'10*, pages 44–56. IEEE, 2010.
17. H. Mantel and A. Starostin. Transforming Out Timing Leaks, More or Less. In *ESORICS'15*, pages 447–467, 2015.
18. D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC'15*, volume 3935, pages 156–168. Springer, 2005.
19. A. Shamir R.L. Rivest and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *CACM'78*, 21(2):120–126, 1978.
20. C. E. Shannon. A Mathematical Theory of Communication. *ACM SIGMOBILE MC2R'01*, 5(1):3–55, 2001.
21. D. M. Smith and G. Smith. Tight Bounds on Information Leakage from Repeated Independent Runs. In *CSF'17*, pages 318–327, 2017.
22. D. Zhang, A. Askarov, and A. C. Myers. Predictive Mitigation of Timing Channels in Interactive Systems. In *CCS'11*, pages 563–574, 2011.