A Comparative Study across Static and Dynamic Side-Channel Countermeasures

Yuri Gil Dantas, Tobias Hamann, and Heiko Mantel

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

Abstract. Timing side-channel attacks remain a major challenge for software security, in particular for cryptographic implementations. Multiple countermeasures against such attacks have been proposed over the last decades, including static and dynamic approaches. Although such countermeasures have been extensively studied in the literature, previous evaluations have mostly relied on simplified system settings. In this article, we provide a comparative evaluation of the effectiveness of both static and dynamic countermeasures in a realistic setting for Java programs. Our experimental setup considers the effects of the non-deterministic timing behavior introduced by the Java VM, in particular involving just-in-time compilation (JIT). Our empirical results indicate that such countermeasures vary heavily on how much they can reduce information leakage, and show that negative effects of non-deterministic timing behavior on their effectiveness are substantial.

1 Introduction

One particular class of timing side-channel vulnerabilities that is frequently exploited by adversaries is caused by conditionals that are dependent on secret data [9]. In this case, a timing side channel is introduced when the if branch of a secret-dependent conditional takes a different time to be executed then the else branch. An adversary can exploit this to deduce information about the secret. For cryptographic implementations, for instance, it has been shown that attacks can, in the worst case, leak the entire secret key [8].

In order to mitigate timing side channels caused by such conditionals, one can modify program behavior to reduce information leakage via the channel. For this article, we consider two classes of such program modifications, which we refer to as static and dynamic transformations. Conceptually, *static transformations*, like cross-copying [1] or conditional assignment [16], aim to completely remove timing side-channel vulnerabilities by modifying the source code of the target program. *Dynamic transformations*, like bucketing [10] or predictive timing mitigation [19], in contrast, delay program events at runtime up to well-defined points in time to reduce the amount of information leaked by the target program.

Previous evaluations of both static and dynamic transformations have been mostly carried out in simplified settings. One of the most prevalent simplifications is the assumption of deterministic timing behavior (e.g., [10], [2]). In a

Published in:
N. Zincir-Heywood et al. (Eds.): FPS 2018, LNCS 11358, pp. 173-189, 2019.
© Springer Nature Switzerland AG 2019
The final authenticated version is available online at
https://doi.org/10.1007/978-3-030-18419-3_12

system with deterministic timing behavior, each input always results in the same timing observation. The impact of non-deterministic timing behavior on bucketing has been investigated in [4]. In that study, two implementations of bucketing that reside at the application and kernel level were developed for reducing timing side channels in Java programs. Empirical results indicated that indeed both implementations performed comparably worse than previous evaluations in settings with deterministic timing behavior. To the best of our knowledge, such an evaluation has not been carried out for predictive timing mitigation. A previous study on different static transformations [14] has compared four well-known static transformations for Java in a simplified setting with just-in-time compilation (JIT) disabled. To date, the impact of non-deterministic timing behavior on different static transformation techniques has not been investigated as rigorously.

The goal of this article is to provide a comparative study on the effectiveness of static and dynamic transformations in a more practical setting that considers the non-deterministic timing behavior introduced by JIT. In particular, we provide an answer to the following research question: *'How do static and dynamic transformations compare to each other in terms of reduction of side-channel leakage?'*. To this end, we evaluate implementations of two static transformations (cross-copying and conditional assignment), and two dynamic transformations (bucketing and predictive timing mitigation). For the static transformations, we evaluate the implementations presented in [14] in a setting with JIT enabled. For the dynamic transformations, the results of [4] indicated that application-level implementations are more effective in reducing information leakage than kernel-level implementations. In order to better understand the effects of different implementation strategies on the application level for such transformations, we present an implementation that manually modifies the target program to enforce bucketing. We compare this implementation to the bucketing implementation presented in [4] that is using a generic enforcement mechanism. In addition, we present and evaluate two implementations of predictive timing mitigation on the application-level, using the same implementation strategies as for bucketing.

Our results indicate that the impact of non-deterministic timing behavior on side-channel countermeasures can be substantial. This impact reduces the effectiveness of both static and dynamic techniques when compared to simplified settings, e.g., assuming deterministic timing behavior. The reduction of effectiveness seems to be more severe for static techniques, and cross-copying seems to be especially affected. While evaluations of cross-copying in simplified settings indicated a reduction of information leakage by roughly 96% [14], the reduction is substantially smaller for our experiments – achieving, on average, a reduction of only 4.48%. The impact on conditional assignment, in contrast, is also clearly negatively affected, but still ensures an average reduction of roughly 87% (in contrast to over 99% in simplified settings [14]). Dynamic transformations seem to be more promising in settings with non-deterministic timing behavior, achieving an average reduction of over 90% in our experiments[1].

[1] We provide all implementations and experimental results online:
`https://drive.google.com/file/d/1CHfHD6Huo2Wp2y_ZQb7OgsKeNxiuhxB3/view?usp=sharing`

2 Timing Side Channels

In a timing side-channel attack, an adversary exploits the timing behavior of a program to deduce secret information. Timing side channels have been a long-standing problem for software security, going back to the work of Kocher [9]. A classical example of a timing-side channel vulnerability can be found in the square-and-multiply implementation of modular exponentiation. Modular exponentiation (modExp for short) is an operation that can be used to compute $p = c^d \ (mod\ n)$. This is especially relevant in RSA implementations, where c is the ciphertext, d the secret key, and n the modulus. In a nutshell, the running time of the square-and-multiply implementation of modExp is dependent on the Hamming weight of the secret key, thus leaking private information to an adversary.

The algorithm of the square-and-multiply implementation of modExp is illustrated in Figure 1. The timing behavior of modExp depends on the secret d since Line 5 is executed more often when more bits of d are set (condition of Line 4). This enables adversaries to learn the Hamming weight of d by measuring the running time of modExp. If an adversary knows the Hamming weight of the secret key, the brute force search space is reduced, opening the possibility of deducing the entire secret key.

```
1  input: c, d, n;
2  r ← 1;
3  for i = 1 to length(d) do
4  |   if d % 2 == 1 then
5  |   |   r ← (r * c) % n;
6  |   end
7  |   c ← (c * c) % n;
8  |   d ← d ≫ 1;
9  end
10 return r % n;
```

Fig. 1: Algorithm of modExp

Statistical Estimation of Information Leakage. A side channel can be modeled as an information-theoretical channel with input alphabet X and output alphabet Y, where X and Y are random variables [15]. Intuitively, X models the possible secret inputs that a program can process, while Y models the possible (timing) observations an adversary can gather through the side channel. The leakage that occurs via the side channel can be measured by the notion of mutual information of X and Y. Mutual information describes the amount of information that Y contains about X, and is calculated as the difference between the (Shannon) entropy [17] and the conditional (Shannon) entropy.

The channel capacity $C(X;Y)$ [17] is defined as the worst-case (i.e., maximal) mutual information across all prior distributions. We use the notion of channel capacity to evaluate the effectiveness of the static and dynamic transformations considered in this article. To this end, we use the leakiEst tool [3] that provides statistical estimations of the channel capacity based on provided sample runs of a program for different inputs.

3 Static Transformations

Static transformation techniques for mitigating timing side channels like, e.g., cross-copying [1] and conditional assignment [16] aim at mitigating side channel

vulnerabilities introduced by secret-dependent conditionals. In such settings, a timing side channel can occur when the if branch of a conditional takes a different execution time than the else branch. A special instance of this problem are secret-dependent conditionals that only consist of an if branch, and an empty else branch. Static transformation techniques modify the program code of a target program such that the executions of all branches in secret-dependent conditionals take the same time. In this section, we discuss two static transformation techniques, namely cross-copying and conditional assignment.

3.1 Cross-Copying

The approach of the cross-copying [1] technique is to add dummy statements resembling the complete corresponding other branch at the end of each branch of secret-dependent conditionals. The goal of this technique is to ensure that each branch takes the same execution time, because the statements of both branches will be executed. Cross-copying can be seen as a special case of unification [11], a similar technique that can add dummy statements at arbitrary points in each branch, and can thus lead to less dummy statements being added. In this article, we consider modExp that contains a secret-dependent conditional without an else branch. Hence, we investigate the simpler cross-copying rather than unification.

Previous work on the evaluation of different static transformation techniques [14] has shown how it is possible to implement cross-copying in Java programs. We leverage this implementation for our comparative experiments.

For the example presented in Figure 1, cross-copying modifies the conditional starting in Line 4 as depicted in Figure 2. The cross-copying technique adds a dummy variable (r_d) to the program, performing the same computation in both branches of the conditional. Note that only in the if branch the result is applied to the local variable that is used to calculate the return value. In the else branch, the result of the computation is assigned to the dummy variable, which is expected to take the same execution time, but will not affect the return value of the algorithm. Hence, this implementation strategy for cross-copying is transparent.

```
1  input: c, d, n;
2  r ← 1;
3  for i = 1 to i = length(d) do
4      if d % 2 == 1 then
5          r ← (r * c) % n;
6      else
7          r_d ← (r * c) % n;
8      end
9      c ← (c * c) % n;
10     d ← d ≫ 1;
11 end
12 return r % n;
```

Fig. 2: modExp after cross-copying

3.2 Conditional Assignment

Conditional assignment [16] aims to mitigate timing side channels caused by an assignment to a local variable that affects the program state, and is dependent on secret information. The conditional assignment technique ensures that all computations that might occur based on the secret-dependent conditional are

executed, and assign the desired value of these computed values to the variable afterwards using a bitmask that chooses the desired value of the computation. This approach eliminates secret-dependent conditionals, as the conditional is completely encoded by the masking process.

```
1  input: c, d, n;
2  r ← 1;
3  for i = 1 to i = length(d) do
4  |   r' ← (r * c) % n;
5  |   m = Mask(d % 2 == 1);
6  |   r ← (m & r') | (~m & r);
7  |   c ← (c * c) % n;
8  |   d ← d ≫ 1;
9  end
10 return r % n;
```

Fig. 3: modExp after conditional assignment

For the example presented in Figure 1, conditional assignment modifies the conditional starting in Line 4 as depicted in Figure 3. Instead of adding dummy assignments as for cross-copying, conditional assignment ensures that both the updated value r' (Line 4) from the if branch and the unchanged r are used for computing the new value of r (Line 6). The assignment to r is performed by masking the updated result and the non-updated result based on the original condition. The `Mask` function used in Line 5 is supposed to return $2^l - 1$ in the case that $d \% 2 == 1$ holds (the condition for the if branch to be taken), where l is the bitlength of the variable r. Correspondingly, the `Mask` function is supposed to return 0 in the case that $d \% 2 != 1$ holds (the condition for the else branch to be taken). Hence, the computation result that should not affect the return value of the algorithm is masked out, making conditional assignment transparent.

4 Dynamic Transformations

Dynamic transformation techniques for mitigating timing side channels monitor program behavior during runtime and react to the monitored behavior dynamically in order to reduce or prevent information leakage via a timing channel. In this section, we discuss implementations of two such dynamic transformation techniques, namely bucketing [10] and predictive timing mitigation [19].

4.1 Bucketing and Predictive Timing Mitigation in an Nutshell

The goal of both bucketing and predictive timing mitigation is to reduce the amount of information leakage via timing side channels at runtime. In contrast to the static transformations presented in Section 3, the goal of both techniques is not to completely remove timing side channels. They rather aim at reducing the amount of information that is leaked via the side channel, thus limiting the information about the secret that an adversary can learn. From a high-level perspective, both approaches delay sensitive program events to well-defined points in time, thus reducing the amount of possible distinct timing observations. It has been shown that reducing the number of possible distinct observations directly reduces the upper bound of possible information leakage via a timing channel

(see, e.g., [12]). By adjusting this delay, both approaches allow a navigation in the tradeoff between security and performance.

To achieve the reduction of possible timing observations that an adversary can get, the two approaches follow different strategies. The bucketing technique discretizes the timing behavior of a protected program by delaying events to a set of predefined points in time, the so called bucket boundaries. Each event that occurs within the interval of a certain bucket is delayed up to the corresponding bucket boundary of that bucket. The approach of predictive timing mitigation, in turn, is to provide a certain prediction schedule that observed events shall adhere to. In case an event is observed before the next point in time that is scheduled – the so called quantum –, the event is delayed up to that quantum. In case the program violates the schedule, the schedule is adapted dynamically, penalizing the program for not adhering to the schedule. The time frame in which a given schedule is adhered to by the program is called an epoch. Penalizing the target program by updating the schedule thus starts a new epoch.

4.2 Implementations

Previous work on bucketing has investigated the effects of different implementation techniques for the bucketing mechanism on the security guarantees provided by these implementations [4]. That work provided first evidence that the choice of system layer where the implementation is placed can have a direct effect on the provided security by the implementation. In this article, we investigate two implementation strategies at the application layer for both bucketing and predictive timing mitigation: instantiations of the generic enforcement framework CliSeAu [6] for the two approaches, and manual implementations that are inlined into the protected program. We assume that program operations that cause a timing leak are located in specific methods. Hence, our implementations monitor invocations to these methods and delay program execution after each such method invocation. Our goal is to evaluate the effects of these different implementation strategies on the same system level.

Implementations of dynamic transformations using CliSeAu. CliSeAu is a generic framework for enforcing security requirements for Java programs on the application level at runtime. Conceptually, CliSeAu encapsulates a target program into a so called enforcement capsule that consists of four additional components responsible for the enforcement: the interceptor, the coordinator, the local policy, and the enforcer. In this work, we follow the approach of [4], and focus on the interceptor component and the enforcer component in our implementation. The interceptor is responsible for intercepting security-relevant program events from the target program. Based on these intercepted events, a decision how to handle the events is made. This decision is then enforced by the enforcer component. For implementing dynamic side-channel mitigation techniques, the interceptor component is used to determine the start time of timing-sensitive computations, while the enforcer component is used to delay events based on the delay strategy by a given mitigation technique.

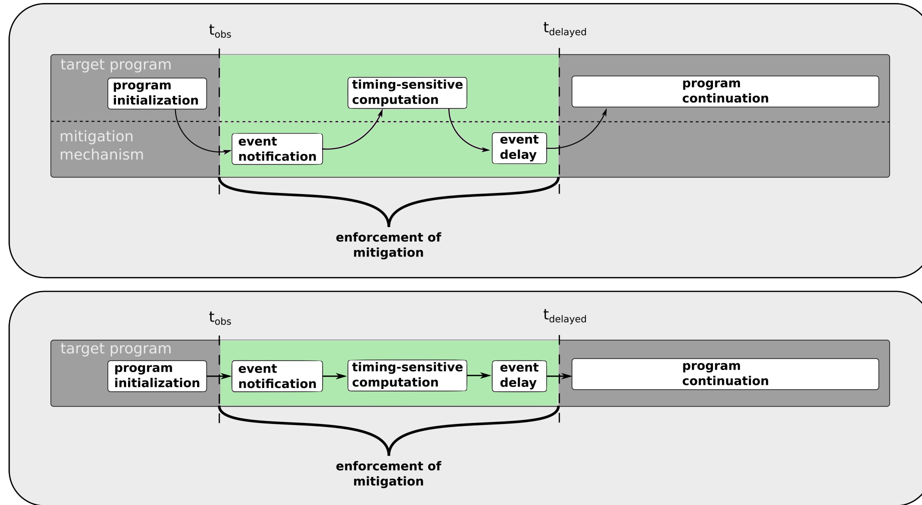

Fig. 4: Visualization of implementation strategy using a generic framework (upper part), and inlining the mitigation (lower part)

This implementation strategy for dynamic side-channel mitigation has been used to implement the bucketing mechanism in CliSeAu [4]. As a point of comparison with predictive timing mitigation and static transformations, we reuse the CliSeAu implementation of bucketing in this article. In a nutshell, the implementation provides the CliSeAu components for enforcing bucketing for Java programs, making it sufficient to declare the method signatures of timing-sensitive computations and to provide the amount and placement of buckets to instantiate it for a given target program. For more details, we refer the interested reader to the original work [4].

We present an implementation of predictive timing mitigation in CliSeAu that builds on the implementation of bucketing in CliSeAu. The overall workflow of any dynamic transformation technique that delays program events at runtime can be seen in Figure 4. In particular, the upper part of Figure 4 shows the workflow when using a generic mechanism like CliSeAu. Just before a timing-sensitive computation is about to start (at time t_{obs}), the mechanism is notified, and program execution continues with the computation. After the computation, the mechanism is notified that the computation has been finished. Based on the delay strategy, the mechanism then delays further program execution to reduce the amount of distinct possible timing observations (up to time $t_{delayed}$). For predictive timing mitigation, there are two cases how the mechanism can delay program events based on the current schedule. In case the timing-sensitive computation finished before the next scheduled quantum, the mechanism delays further program execution up to that quantum and the program continues regularly. In case the computation does not finish before the next scheduled quantum, the mechanism adapts the schedule based on a predefined penalty function, i.e., a

new epoch is started. The mechanism delays further program execution until the newly scheduled quantum, and keeps the new schedule afterwards. In order to instantiate the predictive timing mitigation implementation, users of the mechanism provide the method signatures of the methods containing timing-sensitive computations, and provide the initial schedule as well as the penalty function.

Manually inlined implementations. Instantiating a generic mechanism for mitigation techniques offers an increased level of reusability of the security solutions, as the instantiation can be specialized to a variety of target programs with relatively few implementation overhead. The clear separation of the target program and the enforcement mechanism also decouples the development of the mitigation mechanism, making it possible to treat security as an orthogonal aspect. However, such generic mechanisms inherently introduce a runtime overhead and might even decrease the precision of the mitigation technique. Our goal in this article is to evaluate the effects of the two implementation strategies empirically. We therefore provide manually inlined implementations of both bucketing and predictive timing mitigation. The overall workflow of such inlined implementations is depicted in the lower part of Figure 4. Conceptually, the manually inlined versions of the mitigation strategies are the same as described for the instantations of CliSeAu. In contrast, the inlined implementations contain all code used for the mitigation inside the target program, avoiding splitting the mitigation into multiple components that need to communicate with each other. This increases implementation effort, as developers have to ensure that the mitigation code is located at all occurrences of timing-sensitive computations manually. On the other hand, the avoidance of component interaction and close coupling of the target program and the mitigation code might lead to increased security guarantees and program performance. We provide an empirical evaluation of these effects in the next section.

5 A Comparative Security Evaluation

The goal of this section is to report and discuss the findings of our evaluation regarding the static and dynamic transformations presented in Sections 3 and 4.

We provide all implementations of the transformation techniques, and all experimental results online.[2]

5.1 Experimental Setup, Metric, and Design

Setup. We conduct all experiments on a 3.7GHz Intel Xeon E3-1240 server with 16GB of RAM running Ubuntu 16.04 LTS with kernel 4.4.0, and OpenJDK 8. Furthermore, all experiments are conducted with JIT enabled using the so-called tiered optimization mode. We apply the static and dynamic transformations investigated in this article to an implementation of modExp in Java[3].

[2] https://drive.google.com/file/d/1CHfHD6Huo2Wp2y_ZQb7OgsKeNxiuhxB3/view?usp=sharing

[3] The modExp implementation considered in this article is the same used in [14].

We consider a passive adversary who locally measures the running time of modExp using System.nanoTime(). Each measurement consists of a timing observation value collected by this adversary. We consider an adversary who can observe the execution time of the target program, but not the program's internal state. In particular, he cannot observe internal communication events used in the implementations of bucketing and predictive timing mitigation.

We conduct 100 samples for each mitigation technique to evaluate the practical impact of our results. For each sample, we start with a warm-up phase of 2^{19} measurements that is discarded in the results. We chose this number in order to reach the steady-state of JIT before collecting measurements for our experiments. This approach follows the best practices proposed in [7] for Java performance evaluations. Subsequently, we start with an experimental-phase of 2^{19} measurements that is kept in the results as these measurements relate to the steady-state of modExp. To evaluate the effectiveness of static and dynamic transformations in reducing timing side channels, we consider the mean and worst-case values of our sample distributions. From all collected samples, we reject outliers that lie further than 1 absolute standard deviation from the mean.

Metric. Following the methodology used in [14],[4], we consider channel capacity as our metric to determine the effectiveness of static and dynamic transformations. That is, we measure the correlation between secret inputs and their timing distributions in order to estimate the amount of information (in bits) that might be leaked via a timing side channel.

Design. We conduct the so-called distinguishing experiments (as in [14],[13]) for two distinct secret input values, namely key_1 and key_2. Both keys have 32 bits (with Hamming weights 5 and 25, respectively[4]). Using distinguishing experiments, we can identify a side-channel vulnerability that enables adversaries to distinguish whether either key_1 or key_2 is used by modExp. In our experiments, we generated such distinct keys and conducted experiments with and without any transformation. As a result, we can quantify the channel capacity with the help of an off-the-shelf information leakage estimation tool named leakiEst [3].

5.2 Static Transformations

Static transformations have been evaluated in a simplified setting with JIT disabled in previous work [14]. Our goal in this article is to investigate the impact of non-deterministic timing behavior on static transformations. To this end, we quantify the effectiveness of cross-copying and conditional assignment in terms of channel capacity.

Figures 5 and 6 illustrate our experimental results. Figure 6 shows the histogram of the scenario where no transformation (BASELINE) is applied to modExp. Figures 5a and 5b show the histograms of the scenarios where modExp is using cross-copying (CC) and conditional assignment (CA), respectively. The overall

[4] We also considered keys with other Hamming weights than 5 and 25, but this is outside the scope of this article.

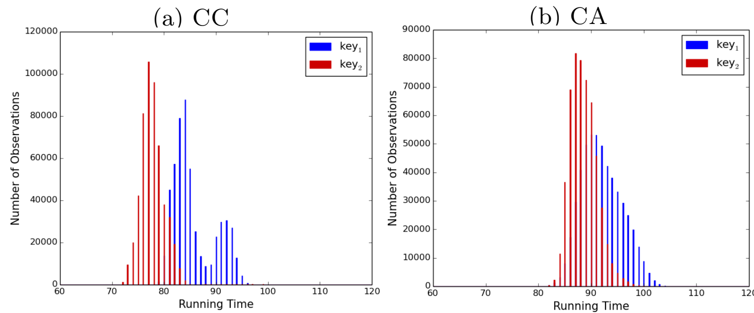

Fig. 5: Timing distributions of modExp after applying static transformations

results regarding the effectiveness of cross-copying and conditional assignment are presented in Figure 7.

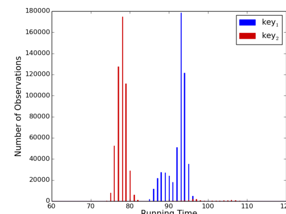

Fig. 6: BASELINE

We observe that when modExp is not using any transformation (Figure 6) the timing distributions of key_1 and key_2 can be clearly distinguished. That is, the histograms indicate that an adversary can distinguish whether modExp is using key_1 or key_2 via a timing side channel. When modExp is using cross-copying, we can observe that the distributions are scarcely overlapping each other. On the other hand, when modExp is using conditional assignment, the distributions are nearly overlapping. These results give us a first hint that cross-copying is not as effective as conditional assignment in a setting with JIT enabled.

Our results are quantified in terms of the channel capacity for the baseline, and both static transformations. For the transformations, we also consider the capacity reduction achieved by the techniques. We quantify both the mean channel capacity across our experimental samples (with 95% confidence intervals), and the worst-case capacity we have observed in the samples. We can observe that cross-copying and conditional assignment clearly differ on how much they can reduce the side-channel vulnerability in modExp. While cross-copying can only reduce the channel capacity by roughly 4.5% for the mean case, conditional assignment achieves a reduction of almost 90%. For the worst-case we observed in our experimental samples, cross-copying can only reduce the capacity by 2.35%, while conditional assignment achieves roughly 73% of reduction. These results substantiate those illustrated in Figure 5, where e.g. the timing distributions of key_1 and key_2 are scarcely overlapping when cross-copying is applied to modExp.

Our results indicate that the impact of the non-deterministic timing behavior introduced by JIT is substantial on cross-copying. In a setting with JIT disabled, as shown in [14], cross-copying is able to reduce the side-channel vulnerability in modExp by 96%. On the other hand, the impact of non-deterministic timing behavior on conditional assignment is clearly observable (the results of [14] have

scenario	channel capacity		reduction	
	mean	worst-case	mean	worst-case
baseline	0.9546±0.0042	0.9997	-	-
CC	0.9119±0.0042	0.9762	4.48%	2.35%
CA	0.1209±0.0065	0.2698	87.33%	73.01%

Fig. 7: Estimated capacity of timing side channels after static transformations

shown a reduction of 99.88% in a setting without JIT), but conditional assignment still seems to be quite effective in systems with non-deterministic timing behavior. While some impact of non-deterministic timing behavior on the effectiveness of static transformations are to be expected, we were surprised by the extent of the impact on cross-copying. Our investigations on the implementation of cross-copying have shown that indeed the mitigated branches look the same, also on the bytecode level. Possible explanations for the poor reduction achieved by cross-copying include factors like branch prediction, garbage collection, or system load. However, most of these factors apply also to conditional assignment, making it hard to find possible reasons for the difference in the achieved reductions. The conditional assignment technique completely eliminates conditionals by relying on bitmasks. Cross-copying, on the other hand, still includes conditionals in the mitigation – but they are designed to take the same execution time for each branch. We believe that this difference might be a key factor for the difference between both techniques.

5.3 Dynamic Transformations

With regard to dynamic transformations, our goal is to investigate the impact of non-deterministic timing behavior on different implementations of bucketing and predictive timing mitigation at the application level. To this end, we quantify the effectiveness of such transformations in terms of channel capacity.

Instantiation of Transformations. Following [4], we conduct experiments using bucketing in isolation. By isolation, we refer to an instantiation of a 1-bucketing. We set the same bucket size for both keys_1 and key_2. Note that this bucket size is greater than the expected worst case running time for either key. Events with running time greater than this bucket size are classified as outliers and, thus, they are released directly by the mechanism. The nature of the implementations of bucketing and predictive timing mitigation is different (as explained in Section 4.2), but in order to enable a comparison between such implementations, we instantiated predictive timing mitigation as follows. We set the initial quantum to the same value as the bucket size. In order to avoid effects of penalization, we choose the penalty function that directly releases events that are not arriving on time (classified as outliers), and stay within the same epoch without adapting the schedule.

We consider scenarios where modExp is using bucketing and predictive timing mitigation. We use subscripts to refer to when e.g., bucketing is manually inlined into modExp ($\text{BUCKETING}_{\text{INL}}$) and implemented using a generic framework

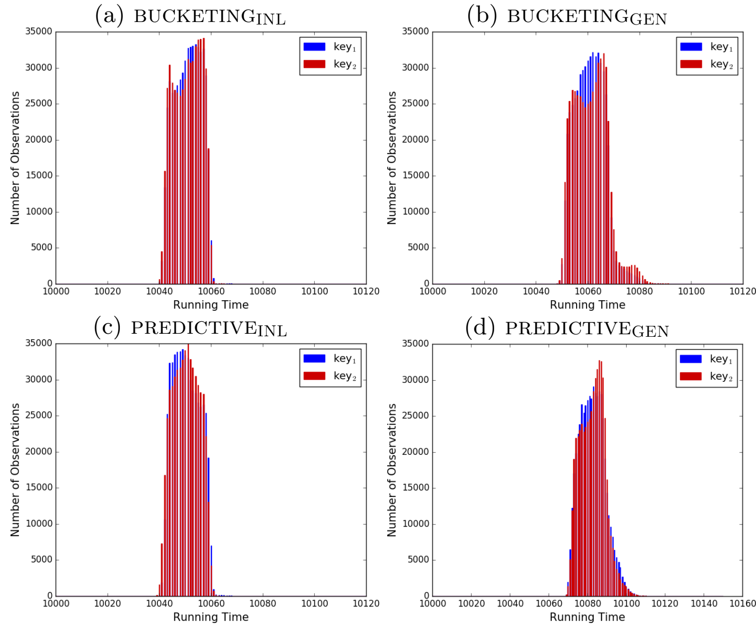

Fig. 8: Timing distributions of modExp after applying dynamic transformations

(BUCKETING$_{\mathrm{GEN}}$). The histograms of such scenarios are depicted in Figure 8. The histogram of the BASELINE is the same illustrated in Figure 6. The overall results regarding the effectiveness of bucketing and predictive timing mitigation are described in Figure 9.

The histograms of bucketing (Figures 8a and 8b) and predictive timing mitigation (Figures 8c and 8d) indicate there are no substantial difference between the timing distributions of key_1 and key_2. That is, their distributions are mostly overlapping. These results suggest successful mitigation of the timing side-channel vulnerability observed in Figure 6.

Our experimental results are quantified in terms of channel capacity in Figure 9. We quantify both the mean channel capacity across our experimental samples (with 95% confidence intervals), and the worst-case capacity we have observed in the samples. Our results show that both bucketing and predictive timing mitigation achieve a very high reduction regarding the side-channel vulnerability in modExp, regardless of the mean or worst-case results. One important observation is that there is a slight difference in the effectiveness of bucketing and predictive timing mitigation when implemented inlined and using a generic mechanism. For instance, considering the worst-case reduction, BUCKETING$_{\mathrm{INL}}$ and PREDICTIVE$_{\mathrm{INL}}$ can, respectively, reduce the channel capacity by roughly 95% and 94%, while BUCKETING$_{\mathrm{GEN}}$ and PREDICTIVE$_{\mathrm{GEN}}$ can, respectively, reduce the channel capacity by 91% and 92%.

scenario	channel capacity		reduction	
	mean	worst-case	mean	worst-case
baseline	0.9546±0.0042	0.9997	-	-
bucketing$_{\mathbf{INL}}$	0.0137±0.0010	0.0484	98.56%	95.16%
bucketing$_{\mathbf{GEN}}$	0.0354±0.0022	0.0891	96.29%	91.09%
predictive$_{\mathbf{INL}}$	0.0117±0.0010	0.0626	98.77%	93.74%
predictive$_{\mathbf{GEN}}$	0.0260±0.0017	0.0798	97.28%	92.02%

Fig. 9: Estimated capacity of timing side channels after dynamic transformations

Our experimental results reflect the fact that generic mechanisms can (although slightly) reduce the effectiveness of mitigation techniques. However, it is not clear to us whether BUCKETING$_{\text{GEN}}$ or PREDICTIVE$_{\text{GEN}}$ leak any additional information related to the secret key. This difference can happen due to additional overhead (e.g., communication between internal components) caused by generic mechanisms. Clarifying whether such information is related to the secret key appears to be an interesting direction for future work.

Regarding the results achieved by bucketing and predictive timing mitigation, we are not surprised by their similarities. Both transformations delay sensitive program events up to well-defined points in time. Furthermore, we instantiated predictive timing mitigation in a comparable fashion with bucketing.

Overall, our results show that both predictive timing mitigation and bucketing are effective in systems with non-deterministic timing behavior. On the other hand, our results also show that neither predictive timing mitigation nor bucketing (confirming the results from [4]) are able to completely close the timing side channel. Possible explanations for these results are activities in the CPU, e.g., system load, that can cause a latency in the response time of programs [4].

5.4 Comparison between Static and Dynamic Transformations

This section summarizes our answer to the following research question: *'How do static and dynamic transformations compare to each other in terms of reduction of side-channel leakage?'*. In this section, we consider the worst-case reduction observed in our experiments. The summary of our results are given in Figure 10. The blue bars illustrate cross-copying and conditional assignment as static transformations, and the green bars illustrate both implementation strategies for bucketing and predictive timing mitigation as dynamic transformations.

The first important observation is that all transformations are affected by our setting with non-deterministic timing behavior. In a system with deterministic timing behavior, each given input would always lead to the same timing observation. In such systems, all transformations investigated in this article are able to reduce the side-channel capacity by 100%. Our experimental results indicate that this is not true for systems with non-deterministic timing behavior, since neither transformation is able to reduce the side-channel capacity by that factor.

The second important observation is that the impact of non-deterministic timing is much more substantial on static than dynamic transformations. The

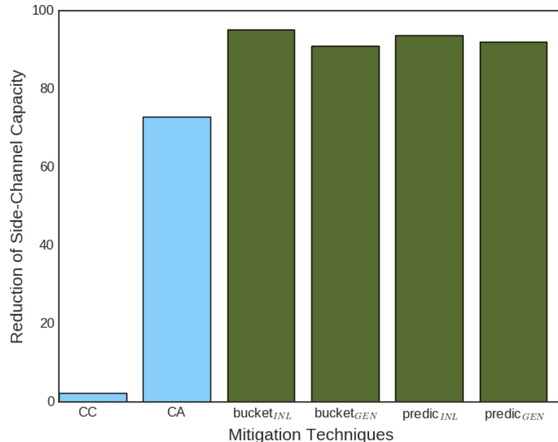

Fig. 10: Overall comparison between static and dynamic transformations

dynamic transformations – bucketing and predictive timing mitigation – performed well in reducing the side-channel vulnerability in modExp. Regardless of the implementation strategy, both transformations performed well comparably by achieving reductions higher than 90%. In contrast, the static transformations – conditional assignment and cross-copying – performed worse in comparison.

Regarding cross-copying, the impact is substantial. Our experiments suggest that cross-copying can poorly reduce the channel capacity by 2.35%. This result is a huge drawback in comparison to the other transformations. Furthermore, in comparison to [14], which investigated cross-copying in a scenario with JIT disabled, the effectiveness of cross-copying falls from 96% to 2.35%. The impact is lower on conditional assignment, but still observable. Our experiments suggest that conditional assignment can reduce the channel capacity by 73%. This result is, however, a drawback in comparison to the results from [14]. That is, with JIT enabled, the effectiveness of conditional assignment falls from 99.88% to 73%.

In summary, our experimental results suggest that dynamic transformations are more effective than static transformations in our setting with JIT enabled.

6 Related Work

Mantel and Starostin [14] have investigated the trade-off between security and overhead of four well-known static transformations, namely cross-copying, conditional assignment, transactional branching, and unification. Their experimental results showed that such transformations differ w.r.t how much security and overhead they add to the program. Our work differs from [14] in how we evaluate static transformations. In [14], all experiments were conducted in a simplified Java environment (with JIT disabled). In contrast, we evaluate the impact of the non-deterministic timing behavior introduced by JIT on conditional assignment and cross-copying, showing that this impact is indeed substantial. Evaluating other techniques (like e.g., unification) in this setting are left for future work.

The effectiveness of dynamic transformation techniques has been evaluated from different perspectives. Bucketing has been originally presented for systems with deterministic timing behavior [10]. Subsequent studies of bucketing have established an algorithm for optimal bucket placement strategies in the tradeoff between security and performance [5]. The original work on bucketing included an information-theoretical upper bound on the leakage of the mitigation that is based on the number of distinct timing observations that an adversary can get, and the number of experiments the adversary obtains. Tighter bounds for this leakage have been presented (e.g., [12], [18]). All of these bounds, however, assume that events can be released sharply at the bucketing boundaries.

Dantas et al. [4] have investigated the impact of non-deterministic timing behavior on bucketing. To this end, they provided two implementations of bucketing that reside at the application level and kernel level. Experimental results indicated that their implementations are not able to release events sharply at the bucket boundary. This led to a large number of observations that an adversary can gather via a timing side channel, and thus to a very high amount of information leakage predicted by the theoretical leakage bounds. Their experimental results also provided the first evidence that the choice of system layer where bucketing is placed can have a direct effect on the provided security by bucketing. That is, their experimental results indicated that bucketing implemented at the application level provides more security (w.r.t. leakage bounds and channel capacity) to Java programs than bucketing implemented at the kernel level. Our work builds on top of [4], providing an empirical investigation of two implementation strategies for bucketing at the application level.

The technique of predictive timing mitigation has been presented in [19] as a generalization of predictive black-box mitigation [2]. In [19], Zhang et al. leverage the black-box mitigation model to web applications. More concretely, they developed a server-side wrapper to mitigate timing leaks from web applications. The predictive black-box model assumes that it can precisely control when events are released by the model. As for bucketing, this assumption does not necessarily hold in systems with non-deterministic timing behavior. Our work empirically investigates predictive timing mitigation for non-deterministic timing behavior to enable a empirical evaluation with bucketing and static transformations.

7 Conclusion

We presented a comparative study on the effectiveness of static and dynamic transformations in a realistic setting with non-deterministic timing behavior. Our results are particularly interesting in three aspects: (1) We show that such transformations differ on how much they can reduce the timing side-channel capacity. (2) We show that the impact of non-deterministic timing behavior is substantial on static transformations, especially on cross-copying. (3) We show that dynamic transformations manually inlined into the target program are more effective (although slightly) than implementations using generic mechanisms.

An interesting direction for future work is a comparative study on the performance overhead caused by static and dynamic transformations. In particular,

it could be interested to investigate the applicability of such transformations in resources-constrained settings like e.g., IoT devices.

Acknowledgments. This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP. This work has been co-funded by the DFG as part of the project Secure Refinement of Cryptographic Algorithms (E3) within the CRC 1119 CROSSING.

References

1. J. Agat. Transforming out Timing Leaks. In *POPL'00*, pages 40–53, 2000.
2. A. Askarov, D. Zhang, and A. C. Myers. Predictive Black-Box Mitigation of Timing Channels. In *CCS'10*, pages 297–307, 2010.
3. T. Chothia, Y. Kawamoto, and C. Novakovic. A Tool for Estimating Information Leakage. In *CAV'13*, pages 690–695, 2013.
4. Y.G. Dantas, R. Gay, T. Hamann, H. Mantel, and J. Schickel. An Evaluation of Bucketing in Systems with Non-Deterministic Timing Behavior. In *SEC'18*, pages 323–338, 2018.
5. G. Doychev and B. Köpf. Rational Protection against Timing Attacks. In *CSF'15*, pages 526–536, 2015.
6. R. Gay, J. Hu, and H. Mantel. CliSeAu: Securing Distributed Java Programs by Cooperative Dynamic Enforcement. In *ICISS'14*, pages 378–398, 2014.
7. A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *OOPSLA'07*, pages 57–76, 2007.
8. 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.
9. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO'96*, pages 104–113, 1996.
10. B. Köpf and M. Dürmuth. A Provably Secure and Efficient Countermeasure against Timing Attacks. In *CSF'09*, pages 324–335, 2009.
11. B. Köpf and H. Mantel. Transformational Typing and Unification for Automatically Correcting Insecure Programs. *Int. J. Inf. Sec.*, 6(2–3):107–131, 2007.
12. B. Köpf and G. Smith. Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In *CSF'10*, pages 44–56. IEEE, 2010.
13. H. Mantel, J. Schickel, A. Weber, and F. Weber. How Secure Is Green IT? The Case of Software-Based Energy Side Channels. In *ESORICS'18*, 2018.
14. H. Mantel and A. Starostin. Transforming Out Timing Leaks, More or Less. In *ESORICS'15*, pages 447–467, 2015.
15. Jonathan K Millen. Covert Channel Capacity. In *S&P'87*, pages 60–66, 1987.
16. 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.
17. C. E. Shannon. A Mathematical Theory of Communication. *ACM SIGMOBILE MC2R'01*, 5(1):3–55, 2001.
18. D. M. Smith and G. Smith. Tight Bounds on Information Leakage from Repeated Independent Runs. In *CSF'17*, pages 318–327, 2017.
19. D. Zhang, A. Askarov, and A. C. Myers. Predictive Mitigation of Timing Channels in Interactive Systems. In *CCS'11*, pages 563–574, 2011.