Transforming Out Timing Leaks, More or Less

Heiko Mantel and Artem Starostin

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

Abstract We experimentally evaluate program transformations for removing timing side-channel vulnerabilities wrt. security and overhead. Our study of four well-known transformations confirms that their performance overhead differs substantially. A novelty of our work is the empirical investigation of channel bandwidths, which clarifies that the transformations also differ wrt. how much security they add to a program. Interestingly, we observe such differences even between transformations that have been proven to establish timing-sensitive noninterference. Beyond clarification, our findings provide guidance for choosing a suitable transformation for removing timing side-channel vulnerabilities. Such guidance is needed because there is a trade-off between security and overhead, which makes choosing a suitable transformation non-trivial.

1 Introduction

Side channels are unintended communication channels that transmit information during the execution of programs. Running time [34, 15, 4], power consumption [35], EM radiation [26, 49], cache behavior [48], and other characteristics can cause side channels. Side channels might reveal information about secrets processed by a program, and this makes them a serious security concern. Timing side channels are particularly critical since they can be exploited remotely [15, 4].

The idea of program transformations, in general, dates back to the seventies [33] and since then has attracted a lot of attention for improving programs, e.g., [16, 12, 5]. More specifically, a spectrum of program transformations has been proposed for removing timing side-channel vulnerabilities [2, 47, 13, 38]. The objective of such transformations is to improve the security of programs. That a program is secure wrt. timing side channels can be formalized by a timing-sensitive noninterference-like property (see, e.g., [2]). That a transformation is sound wrt. its objective can then be shown by proving that each transformed program satisfies the property based on a timing-sensitive program semantics [2].

The objective of our research project was to improve the understanding of program transformations for eliminating timing side-channel vulnerabilities. We wanted to better understand how much security is added by such transformations at which costs, in practice. Hence, we chose an experimental approach. In our study we focused on four well-known source-to-source transformations: cross-copying [2], conditional assignment [47], transactional branching [13], and unification [38]. Each of these transformations is transparent in the sense that it

To appear in:
G. Pernul et al. (Eds.): ESORICS 2015 — Part I, LNCS 9326

does not change a sequential program's input/output behavior. Hence, the only negative consequence of these transformations is the overhead that they induce.

Our experimental results clarify that all four program transformations reduce the capacity of timing side channels. These capacity reductions are substantial, but they differ between the transformations. Regarding negative consequences, our experimental results show that all four program transformations cause some performance overhead. The worst-case overhead substantially differs between the transformations, ranging from 18 to 372 percent in our experiments.

Previously, the effectiveness of program transformations for removing timing side-channel vulnerabilities was evaluated mostly analytically. In [2], [13], and [38], it is proven that cross-copying, transactional branching, and unification, respectively, establish timing-sensitive noninterference. In [47], it is proven that conditional assignment establishes the program counter security (PC-Security). The only prior experimental study of the effectiveness of transformations is the investigation of cross-copying in [3]. The overhead of program transformations also was evaluated mostly analytically, based on the code-size blow-up wrt. the definitions of transformations. The only prior experimental study of the overhead induced by transformations is the investigation of conditional assignment in [47].

In contrast to most prior work, we perform our evaluation empirically. We measure the running time of baseline and transformed programs in a series of experiments. From these experimental results, we estimate the performance overhead induced by a transformation by computing the percentage increase of a program's mean running time caused by the transformation. We estimate the effectiveness of a transformation by computing the percentage reduction of the timing side-channel capacity in a program achieved by the transformation. We run all our experiments on a contemporary laptop using realistic Java programs.

Our observation, which might be surprising, is that there are substantial differences in the capacity reduction even between transformations that have been proven before to establish fairly similar definitions of timing-sensitive noninterference. This suggests that analytical investigations of the security established by such program transformations are not yet satisfactory wrt. practice.

In summary, the two main novel contributions of this article are

- the quantification of the positive and negative consequences of different program transformations based on experiments, and
- the clarification of the trade-off between performance overhead and security in this context.

In addition, we provide guidance for selecting a suitable transformation by exploiting our results of the performance and security evaluations in combination.

The article is structured as follows. In Section 2, we define the class of timing side channels relevant for this article. In Section 3, we recall the aforementioned transformations and explain our implementations of them. In Section 4, we introduce our benchmark programs and our experimental setup. In Sections 5 and 6 we present the performance and security evaluation, respectively. In Section 7 we analyze the performance-security trade-off. After a discussion of related work in Section 8, we conclude in Section 9.

2 Timing Side Channels

An illustrative example [34] of a timing side channel can be found in the square-and-multiply modular exponentiation. Such exponentiation is used, e.g., during private-key operations in RSA [50] for computing $R = y^k \bmod n$, where n is public, y can be eavesdropped by the attacker, and k is the secret key. A vulnerable Java implementation containing a timing side channel is given in Figure 1.

```
public int modExp(int y, int k) {
  int 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;
}
```

Figure 1. Square-and-multiply modular exponentiation.

The secret key is stored in integer parameter k. It is processed bitwise starting from the least significant bit. Each bit of k is tested. If the current bit is set, extra multiplication and modulo operations are performed (highlighted lines in Figure 1). Since these extra operations are performed only for the set bits of the secret key, the running time of this implementation varies depending on the number of the set bits. More concretely, the running time encodes the Hamming weight of the secret key. Therefore, the Hamming weight of the secret key is leaked through the timing behavior in one run.

This example illustrates how a conditional statement may result in a timing side channel. If the Boolean condition of a conditional statement contains secret information, then the resulting timing side channel leaks secret information. We will refer to conditional statements that may result in timing side channels leaking secret information as *critical conditionals*.

Previously proposed program transformations for removing timing side channels [2, 47, 13, 38] aim at eliminating timing side channels that result from critical conditionals, like the one in the above example program. This is the class of timing side channels on which we focus in this article.

3 Program Transformations

We consider four transformations: cross-copying [2], conditional assignment [47], transactional branching [13] and unification [38]. Their original definitions from the respective articles assume special statements like skip, dummy assignments, etc. Such statements are not available in real-world programming languages by default. In order to analyze program transformations in practice one first needs to implement the missing special statements. These implementations are not obvious because for each special statement there is a spectrum of design decisions.

The four transformations were defined for different programming languages. For instance, transactional branching was defined for an object-oriented programming language in [13], while unification was defined for a simple language with conditionals and loops in [38]. For our comparison, we use a language that provides all features that are in the intersection of the languages in [2, 47, 13, 38].

This resulting language is a high-level programming language that restricts bodies of critical conditionals to contain only assignments to variables or fields of primitive data types or arrays, and other conditional statements.

3.1 Cross-Copying

Cross-copying [2] pads the branches of critical conditionals in order to equalize the timing behavior of both branches. Technically, cross-copying appends a sequence of dummy statements that shall mimic the timing behavior of one branch to the respective other branch, hence the name "cross-copying". The inserted dummy statements perform the same computations as the statements that shall be mimicked, but dummy statements do not update program variables that are relevant for the program's behavior.

Realization in [2] Padding is realized with the help of a special statement SkipAsn x e. It shall take the same time to execute as the assignment x := e, but that does not change the value of x. SkipAsn-versions of all assignments in one branch of a critical conditional are appended to the other branch, and vice versa. For instance, the critical conditional from Figure 1 is transformed to

```
if (k % 2 == 1) { r = (r * y) % n; } else { SkipAsn r ((r * y) % n); }
```

Our Implementation in Java We implement SkipAsn by assignments to dummy variables. For each statement SkipAsn x e that needs to be inserted, we introduce a dummy field xSkip assuming xSkip is not present in the original program. We implement then SkipAsn x e by xSkip = e. Such implementation is transparent because assignments to dummy fields do not affect the values in the original computation while introducing the desired delays in the running time.

3.2 Conditional Assignment

Conditional assignment [47] removes critical conditionals, so that both branches are consecutively executed. Boolean conditions of the removed conditionals are encoded directly in the assignments from both branches of the original code. The encoding is done with the help of bit masks and bitwise logical operators.

Realization in [47] Function Mask(b) is used for encoding a boolean condition b of a critical conditional. It satisfies Mask(false) = 0 and Mask(true) = 2^l–1, where l is the length in bits of the variables assigned under the critical conditional. Suppose that in a program, x is assigned e_t if b evaluates to true, and e_f if b evaluates to false. Then, in the transformed program, x is assigned (m & e_t) | (˜m & e_f), where m = Mask(b) and &, |, and ˜ are bitwise conjunction, disjunction, and negation. For instance, the critical conditional from Figure 1 is transformed to

```
r = (Mask(k % 2 == 1) & ((r * y) % n)) | (˜Mask(k % 2 == 1) & r);
```

Our Implementation in Java In [47], it is shown that MASK can be implemented in C without conditional statements by defining MASK(b) as $-b$. Such implementation is not suitable for Java because type casting from booleans to integers is not allowed in Java. We came up with a different solution: MASK(a == b) is implemented for 32-bit integers as ~(((a−b)>>31) | ((b−a)>>31)), where >> is the sign-extending right shift. Such implementation is correct because Java uses two's complement integer numbers, and the check whether two integers a and b are equal is equivalent to checking $\neg(((a-b)<0)\vee((b-a)<0))$.

3.3 Transactional Branching

Transactional branching [13] leverages a transaction mechanism for cross-copying. Each branch of a critical conditional is wrapped in a transaction and sequentially composed with the respective other branch. The transaction of the original branch is committed, while the transaction of the cross-copied branch is aborted.

Realization in [13] Three transaction primitives are used. BEGINT starts a new transaction. ABORTT aborts a transaction dismissing all changes made since BEGINT. COMMITT commits a transaction making all changes since BEGINT effective. The original branch is wrapped by the pair BEGINT-COMMITT. The cross-copied branch is wrapped by the pair BEGINT-ABORTT. For instance, the critical conditional from Figure 1 is transformed to

```
if (k % 2 == 1) { BEGINT; ABORTT; BEGINT; r = (r * y) % n; COMMITT; }
else { BEGINT; r = (r * y) % n; ABORTT; BEGINT; COMMITT; }
```

Our Implementation in Java We implement transaction primitives by methods that operate on copies of variables that are not yet committed. For each assignment x := e under a critical conditional we introduce a field xCopy assuming that xCopy is not present in the original program. We implement then BEGINT as xCopy = x, ABORTT as x = xCopy, and leave the body of COMMITT empty. Such implementation is correct because it straightforwardly realizes the required functionality of the transaction primitives.

3.4 Unification

Unification [38] is similar to cross-copying in the sense that dummy statements are added to the branches of critical conditionals in order to equalize the timing behavior of both branches. In contrast to cross-copying, these dummy statements might be inserted into the branches instead of being appended only at the end of branches. A unification algorithm is used to determine where dummy statements need to be inserted into each branch, hence the name "unification". Unification can be viewed as an optimization of cross-copying that inserts never more, but often fewer dummy statements into a program.

Realization in [38] In [38], unification assumes a program semantics in which execution of every statement consumes one time unit, but its adaptation to a more fine-grained timing-sensitive program semantics is straightforward. Padding is realized in [38] using the special statement SKIP. It has no effect on the values of variables, but its execution consumes one time unit. For instance, the critical conditional from Figure 1 is transformed to

```
if (k % 2 == 1) { r = (r * y) % n; } else { SKIP; }
```

Note that the advantage of unification over cross-copying does not become apparent in this example, because the critical conditional in the original program lacks an else-branch.

Our Implementation in Java In our implementation of unification, we use the same dummy statements as in our implementation of cross-copying. Such implementation is transparent because assignments to dummy fields do not affect the values in the original computation while introducing the desired delays.

4 Our Benchmark Programs and Experimental Setup

An existing suite of benchmark Java programs that contain timing side-channel vulnerabilities would be an ideal candidate for an empirical evaluation of program transformations for removing such vulnerabilities. To the best of our knowledge, there is unfortunately no such suite. That is why, we identify meaningful candidates for benchmark programs ourselves. We choose four programs: (i) square-and-multiply modular exponentiation from RSA [50], (ii) computation of a share's value [2], (iii) Kruskal's algorithm for calculating the minimum spanning tree (MST) of a graph [40], and (iv) modular multiplication from the IDEA cipher [41]. These four programs should not be seen as a complete benchmark that is sufficient to investigate transformations in full detail. However, since these programs come from different domains and have different degree of sophistication, they offer themselves as meaningful candidates for our experiments.

4.1 Our Benchmark Programs

Modular Exponentiation Program modExp is the square-and-multiply modular exponentiation discussed in Section 2. The security concern is that the Hamming weight of the secret key k is leaked via a timing side channel.

Share's Value Program shareValue computes the total market value of a specified share form the user's portfolio. In [2], similar program was used to illustrate timing side channels. The portfolio is represented by two arrays, ids and qty, that store identifiers of shares and the number of corresponding shares possessed by the user, respectively. Which shares are possessed by the user is a secret. Method `public int shareValue(int[] ids, int[] qty)` computes the total market value of a specified share from the portfolio. The security concern is that the fact whether the user possess the specified share is leaked via a timing side channel.

Kruskal's Algorithm Program kruskal implements Kruskal's algorithm [40] for calculating the minimum spanning tree of a graph. Kruskal's algorithm is used among others for compression of database queries and responses to them [29].

In case a secret is stored in a database, both queries and responses may contain secret information. Method **public int**[] kruskal(**int**[] g) computes the MST for graph g represented by its adjacency array. The security concern is that the number of graph's vertices is leaked via a timing side channel.

Modular Multiplication Program mulMod16 is a modular multiplication from the IDEA cipher's [41] implementaion in cryptographic library FlexiProvider [1].The encryption and decryption of this IDEA's implementation use mulMod16 several times for computing with the secret key. Method **private int** mulMod16(**int** a, **int** b) implements multiplication modulo $2^{16}+1$ for operands a and b. The security concern is that 16 bits of the secret key leak via a timing side channel. Corresponding timing side-channel attacks have been reported [32, 43].

4.2 Our Experimental Setup

We run all experiments on a typical laptop, a Lenovo ThinkPad T510 with Intel Core i7 CPU @2.67GHz×4 and 4Gb RAM under Ubuntu 12.04 LTS with OpenJDK 64-Bit Server VM. We measure the running time of programs in nanoseconds using System.nanoTime(). We want to stay close to the program semantics in which the transformations have been originally defined. In particular, we want to avoid aggressive compiler optimizations that might revert transformations. Because of that we disable the JIT compilation. This might be seen as a simplification of a practical environment, however the main goal of this research project is to empirically evaluate theoretical concepts of different program transformations and to clarify the relationship between them. It is not the goal of this research project to fully solve the problem of timing side channels in practice.

5 A Performance Evaluation

Our goal is to quantify the performance overhead induced by program transformations in practice. Estimating performance of Java programs in a statistically sound fashion requires a careful experimental design and analysis of the obtained data. We guide our decisions for such a design and analysis by the principles of statistically rigorous Java performance evaluation by Georges et al. [28].

5.1 Experimental Design

We estimate the running time of Java programs by random sampling. We draw each sample of the running time from a different invocation of the Java VM. This is necessary because the running time samples drawn from the same invocation will not be independent. We measure the running time of a program directly after the invocation of the Java VM, i.e., we do not perform any warm-up computations. It has been recognized [28] that because of the JIT compilation the performance of Java programs may improve after certain amount of warm-up computation is made. We however excluded the JIT compilation from our setup.

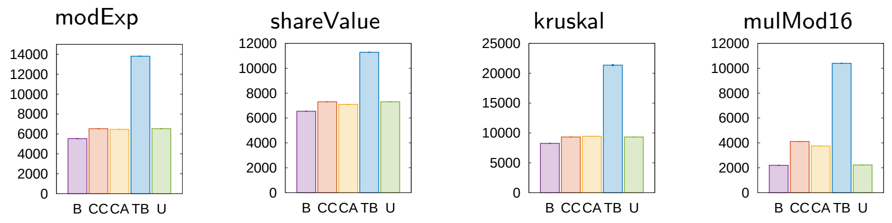
To estimate the running time of a program, we first generate a vector of random inputs. We run the program on each input in a freshly invoked Java VM. We measure the running time of the program within each Java VM invocation in nanoseconds using System.nanoTime(). The measured time value constitutes a sample of the running time. From all collected samples, we reject outliers that lie further than three median absolute deviations from the median.

5.2 Experiments and Experimental Results

We apply each of the 4 transformations to each of the 4 benchmark programs. By that we obtain 17 unique programs: 4 baseline and 13 transformed ones. We obtain 13 unique transformed programs instead of 16 because the resulting programs for cross-copying and unification coincide for modExp, shareValue, and kruskal. Next, we perform the timing measurements for these 17 programs.

The inputs to modExp are pairs of random integers. The inputs to shareValue are pairs of arrays of random integers. Each array has 10 elements. The inputs to kruskal are random graphs. Each graph has 7 vertices and 7 edges. That is, each input is an array of 15 integers: The first element stores the number of vertices, and the next 14 elements store 7 edges as the pairs of source and target vertices. The inputs to mulMod16 are pairs of random integers.

We collect 1000 samples of the running time for each baseline and transformed programs. From these samples we compute 95% confidence intervals [11] for the estimated mean running time. The results are presented in Figure 2.



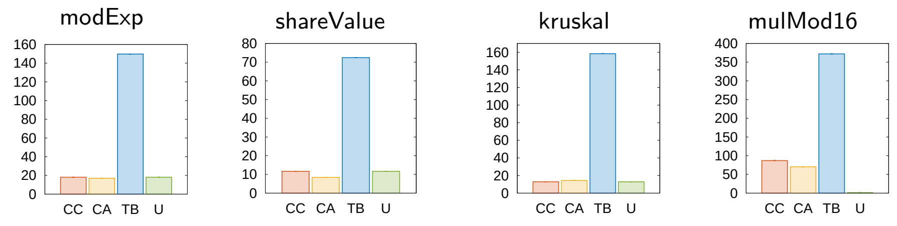
	modExp	shareValue	kruskal	mulMod16
baseline (B)	5528.56±8.67	6544.11±7.65	8264.75±12.09	2201.58±3.96
cross-copying (CC)	6530.77±8.3	7306.83±5.79	9326.25±9.87	4116.23±5.42
cond. assign. (CA)	6468.12±6.93	7095.51±6.62	9450.69±10.56	3751.69±5.43
trans. bran. (TB)	13808.48±10.17	11284.31±11.15	21348.65±92.77	10393.93±10.97
unification (U)	6530.77±8.3	7306.83±5.79	9326.25±9.87	2231.09±3.93

Figure 2. Estimated mean running time, in ns, 95% confidence intervals.

5.3 Our Findings in the Performance Evaluation

In order to clarify how much overhead is introduced by transformations, we use the estimated mean running time to compute the percentage increase of the running time due to each transformation. The result of this is given in Figure 3.

We observe that program transformations generally introduce some performance overhead. The observed overhead substantially differs between the transformations. Altogether, the observed overhead varies from 1 to 372 percent. The worst-case overhead of transformations among different benchmark programs varies from 18 to 372 percent. The experimental results suggest that transactional branching introduces the largest overhead that varies from 72 to 372 percent. We observe moderate difference between the overhead introduced by cross-copying and conditional assignment. For mulMod16 we observe substantial difference between unification and all other transformations. In this case, unification introduces only a marginal overhead of about 1 percent.



	modExp	shareValue	kruskal	mulMod16	worst case
cross-copying (CC)	18.13	11.66	12.84	86.97	86.97
cond. assignment (CA)	16.99	8.43	14.35	70.41	70.41
trans. branching (TB)	149.77	72.43	158.31	372.11	372.11
unification (U)	18.13	11.66	12.84	1.34	18.13

Figure 3. Performance overhead based on the estimated mean running time, in %.

Comparison with Findings in [47] The only prior experimental study of the overhead induced by transformations is the investigation of conditional assignment by Molnar et al. in [47]. The experiments were done on three programs implemented in C. The experimental results in [47] indicate a much larger overhead for conditional assignment than the one observed in our experiments. The worst case overhead observed in [47] is about 480 percent. Interestingly, in [47] a modular exponentiation from RSA and a modular multiplication from IDEA are also used as benchmark programs. For these programs, the overhead observed in [47] is about 150 and 200 percent, respectively. We, however, observe an overhead of only 17 and 70 percent for our versions of these programs, respectively. Note that the versions of these programs in [47] and in our work originate from different cryptographic libraries and are implemented in C and Java, respectively.

6 A Security Evaluation

Our goal is to quantify the effectiveness of program transformations in practice. In the spirit of Millen [46], we model a timing side channel as a discrete information-theoretic channel [21] 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 [51], denoted $C(X;Y)$. We statistically estimate [17] the channel capacity $C(X;Y)$ from empirically collected timing observations. To quantify the positive effects of a transformation we compute the percentage reduction of the timing side-channel capacity achieved by the transformation.

6.1 Experimental Design

For each benchmark program we design the following experiment to which we will refer as the *distinguishing experiment*. We generate two distinct secret input values for a program. Our security concern is that the fact whether the program has received the first or the second secret input value is leaked via a timing side channel. For each of the two secret input values we repeatedly run the program. For each run we freshly invoke the Java VM. We measure the running time of the program within each Java VM invocation in nanoseconds using System.nanoTime(). The resulting value of the time measurement constitute a sample of the running time. From all collected samples, we reject outliers that lie further than three median absolute deviations from the median. We augment each sample with a Boolean variable that stores whether the sample resulted from the first or from the second secret input value. We pass the list of such augmented samples into the procedure for statistical measurement of information leakage [17]. This procedure estimates the capacity $C(X;Y)$ of the timing side channel using iterative Blahut-Arimoto algorithm [14, 8].

6.2 Experiments

Similarly to our performance evaluation, we run our security evaluation experiments on 4 baseline and 13 transformed programs. We run a distinguishing experiment for each of these 17 programs.

Two distinct secret inputs for each of the programs are generated as follows.

In modExp, the timing side channel of our interest results from a critical conditional with the Boolean condition over parameter k. Hence, we supply two different secret inputs to k: fixed integers with the Hamming weight of 5 and of 25, respectively. The other parameter of modExp receives a fixed integer.

In shareValue, the timing side channel of our interest results from a critical conditional with the Boolean condition over parameter ids. Hence, we supply two different secret inputs to ids: an array of 10 fixed integers that does not contain the value representing the user's specified share, and an array of 10 fixed integers that contains at one element a value representing the user's specified share. The other parameter of shareValue receives an array of 10 fixed integers.

In kruskal, the timing side channel of our interest results from a critical conditional with the Boolean condition depending on parameter g. Hence, we supply two different secret inputs to g: an array encoding a fixed graph with 5 vertices and 7 edges, and an array encoding a fixed graph with 7 vertices and 7 edges.

In mulMod16, the timing side channel of our interest results from a critical conditional with the Boolean condition over parameter a. Hence, we supply two different secret inputs to a: a fixed integer whose 16 least significant bits are all zeros, and a fixed integer whose 16 least significant bits contain ones and zeros. The other parameter of mulMod16 receives a fixed integer.

We collect 10000 samples of the running time for each of the two secret inputs for each baseline and each transformed version of benchmark programs.

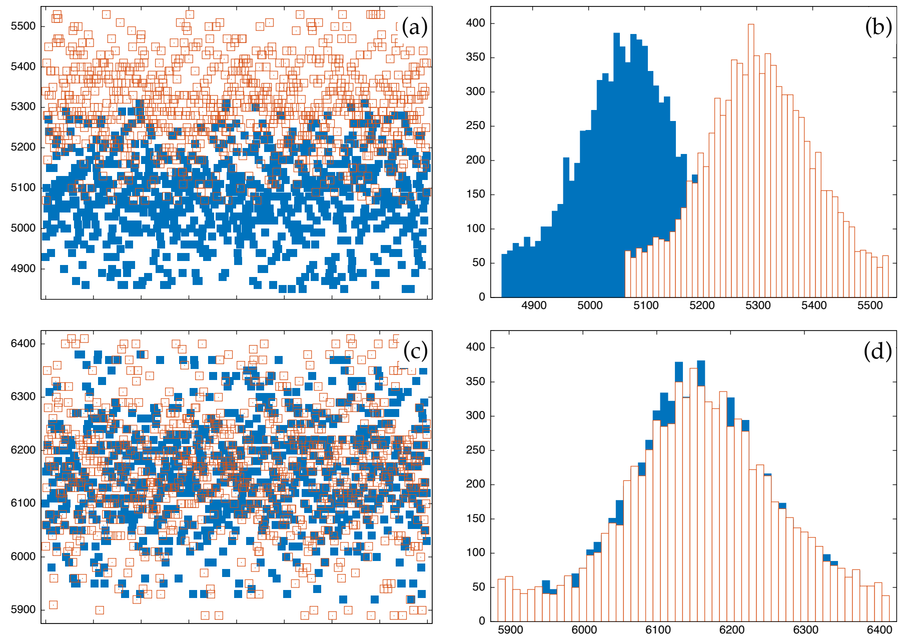


Figure 4. Running time values and frequencies of their occurrence in the distinguishing experiment for modExp in the baseline and cross-copied versions.

6.3 Experimental Results

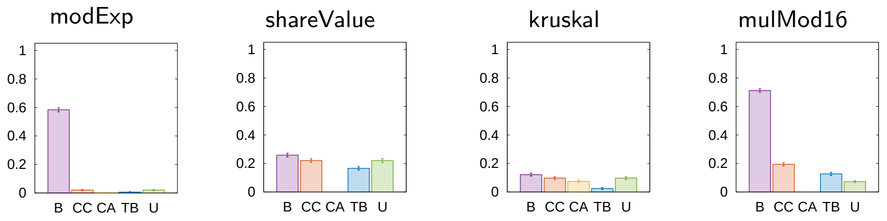
Already just by visualizing the collected samples of the running time one can get a first impression about timing side channels in each program and about the effects of program transformations on these timing side channels.

Figure 4a depicts a portion of the collected running time samples for the baseline version of modExp. Blue (filled) boxes correspond to the first 800 running time samples that resulted from executing modExp on the secret input with the Hamming weight of 5. Red (unfilled) boxes correspond to the first 800 running time samples that resulted from executing modExp on the secret input with the Hamming weight of 25. Figure 4b depicts the frequency with which different running time samples occurred in the experiment. Again, blue (filled) and red (unfilled) bars correspond to the samples that resulted from executing modExp on the secret inputs with the Hamming weights of 5 and 25, respectively. On both Figures 4a and 4b we can clearly observe differences in the running time values that correspond to two different secret input values. This gives us a hint that modExp indeed contains a timing side channel.

Similarly, Figure 4c depicts a portion of the collected running time samples for modExp transformed with cross-copying. Figure 4d depicts the frequency with which different running time samples occurred in the experiment. Blue and red (filled and unfilled, respectively) correspond to the running time samples that resulted from executing the transformed program on the secret inputs with the

Hamming weights of 5 and 25, respectively. In contrast to Figures 4a and 4b, we cannot observe much difference in the running time values that correspond to two different secret input values. This gives us a hint that cross-copying was effective in removing the timing side channel in modExp.

From the collected samples we estimate the capacity of the timing side channels using a procedure for statistical measurement of information leakage [17]. The resulting estimated capacity is depicted in Figure 5. Since in our distinguishing experiments the size of the secret is 1 bit, the maximal possible capacity of the timing side channel in each program is also 1 bit.



	modExp	shareValue	kruskal	mulMod16
baseline (B)	0.5833±0.0142	0.2586±0.0132	0.1216±0.0104	0.7115±0.0121
cross-copying (CC)	0.0202±0.004	0.2204±0.0134	0.0974±0.0097	0.1943±0.0115
cond. assign. (CA)	0.0007±0.0014	0.0±0.0008	0.0743±0.0087	0.0±0.0007
trans. bran. (TB)	0.0062±0.0034	0.1657±0.012	0.0244±0.0066	0.1266±0.0114
unification (U)	0.0202±0.004	0.2204±0.0134	0.0974±0.0097	0.0727±0.0069

Figure 5. Estimated capacity of timing side channels, in bits, 95% confidence intervals.

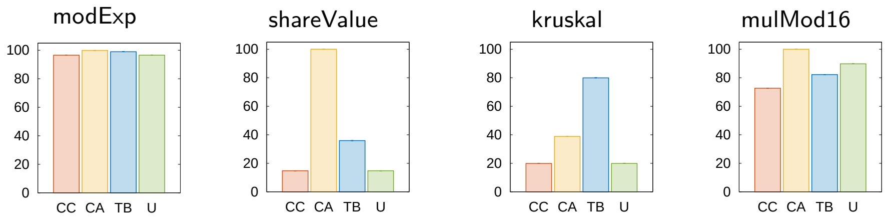
6.4 Our Findings in the Security Evaluation

The results of our experiments show that executing each benchmark program opens timing side channels that have various capacities. The experimental results also show that all program transformations in all experiments reduce the capacity of timing side channels, i.e., all considered transformations have positive effects wrt. side-channel mitigation. In order to clarify how large these positive effects of transformations are, we use the estimated capacity of timing side channels to compute the percentage reduction of the side channel's capacity due to each transformation. The result of this is given in Figure 6.

We observe that program transformations generally reduce the capacity of timing side channels, and that the observed reduction substantially differs between the transformations. Altogether, the observed reduction varies from about 15 to 100 percent. We also observe that the reduction of the capacity of timing side channels varies between cross-copying, transactional branching, and unification. These transformations have been previously proven in [2], [13], and [38] to establish respective definitions of timing-sensitive noninterference.

The transformation "conditional assignment" has been proven in [47] to establish PC-Security. We observe that, in shareValue and mulMod16, conditional assignment completely removes timing side channels, and, in modExp, it achieves a 99.88% reduction of the estimated timing side-channel capacity. For these three programs, conditional assignment removes timing side channels more effectively than the other transformations. One might wonder: Why is conditional assignment so much worse for kruskal, achieving a reduction of only 38.9% and being outperformed by transactional branching? We investigated this question and suspect that the remaining timing side-channel capacity in kruskal is caused by the recursive function find (see Figure 9 in the appendix).

Our experimental results clarify that, in practice, there are differences in the effectiveness of program transformations for removing timing side-channel vulnerabilities. The differences are substantial, and therefore our results indicate that there is still much to be understood about such transformations. Under which conditions should a program developer prefer one transformation over another? Can a program developer maximize the positive effects of a transformation by his programming style, and, if yes, how? Such questions will require answers until we fully understand how to use program transformations for writing programs that are free from timing-side channel vulnerabilities.



	modExp	shareValue	kruskal	mulMod16
cross-copying (CC)	96.54	14.77	19.9	72.69
cond. assign. (CA)	99.88	100.0	38.9	100.0
trans. branching (TB)	98.94	35.92	79.93	82.21
unification (U)	96.54	14.77	19.9	89.78

Figure 6. Reduction of the estimated capacity of timing side channels, in %.

Comparison with Findings in [3] The only prior experimental study of the effectiveness of transformations is the investigation of a Java bytecode implementation of cross-copying by Agat in [3]. This investigation had a qualitative nature and did not consider bandwidths of timing side channels. The experiments were done on synthetic benchmark programs. In contrast to our findings, no significant timing differences for the transformed programs have been observed. There might be several reasons for that: The transformation was implemented in Java bytecode, and different experiments, programs, and a setup were used.

7 Navigating in the Performance-Security Trade-off

Usually security comes at a price. Our evaluation of the overhead introduced by four program transformations for removing timing side-channel vulnerabilities shows that these transformations are no exception. But what is the relationship between the security and its price?

In this section we attempt to explore this relationship for the considered program transformations. In Figure 7 we plot together the results of our performance

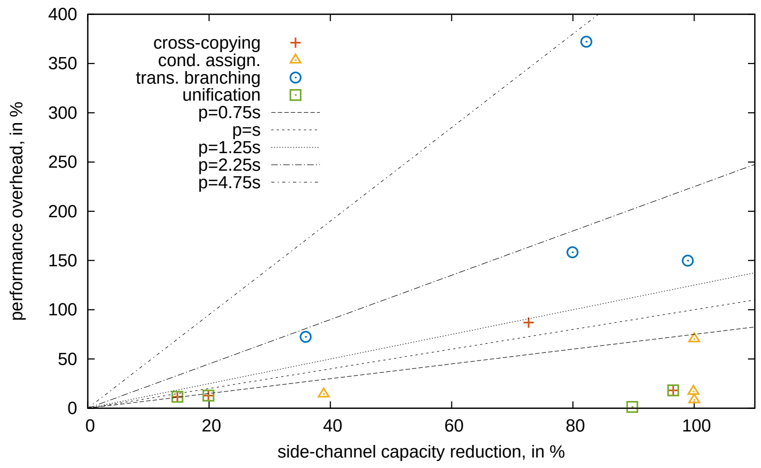


Figure 7. Analyzing performance-security requirements for transformations.

and security evaluations. The ordinate denotes the values of the performance overhead from Figure 3. The abscissa denotes the values of the side-channel capacity reduction from Figure 6. Red crosses, yellow triangles, blue circles, and green boxes correspond to cross-copying, conditional assignment, transactional branching, and unification, respectively. There are four markers of each marker type. Each marker corresponds to an experiment with one benchmark program.

We are interested in analyzing which transformations satisfy a performance-security requirement of the form "We are willing to pay α percent in performance overhead for 1 percent of side-channel capacity reduction" for different values of α. Let p denote the performance overhead in percent, and let s denote the side-channel capacity reduction in percent. Equation $p = \alpha s$ represents the above performance-security requirement. In Figure 7 we plot beams that satisfy the equation $p = \alpha s$ for different values of α. Whenever all four markers of the same marker type lie below the beam for particular α, the transformation that corresponds to this marker type satisfies the performance-security requirement for this α. We vary α from 0 to 5 with the step 0.25.

Our experimental results suggest: 1) Conditional assignment satisfies $p = 0.75s$. 2) Unification satisfies $p=s$. 3) Cross-copying satisfies $p=1.25s$. 4) Transactional branching satisfies $p=4.75s$. (In three cases, it satisfies $p=2.25s$.)

We conclude that conditional assignment satisfies our performance-security requirement of interest for the smallest value of α among all transformations. Furthermore, the above list allows us to identify the ordering between the transformations wrt. how expensive is the security offered by them. This list can serve as an initial guidance for reducing the search space of candidate program transformations that one may want to deploy in practice.

One weakness of the considered requirement is that it suggests that a transformation not impacting performance, but only very slightly decreasing the side-channel capacity might be considered superior to any other transformation. This weakness can be overcome by requiring in addition all transformations to achieve a minimum threshold in reduction of the side-channel capacity. Naturally, further performance-security requirements may also be of one's interest. In this section, we illustrate how one can use our experimental results for analyzing how good different program transformations satisfy such requirements.

8 Related Work

There is a large body of work on the analysis of side channels from the attacker's perspective, e.g., [34, 32, 35, 26, 49, 48, 15, 4, 19]. Timing side channels have been for the first time exploited by Kocher to attack an implementation of RSA [34].

A successful attack against an implementation proves that the implementation is vulnerable. On the contrary, timing-sensitive noninterference-like properties have been used to express that an implementation is secure wrt. timing side channels [2, 30, 13, 38, 23]. Noninterference-like properties express very strong security, which usually implies that an attacker cannot gain any information about given secrets. In practice, however, some leakage might be unavoidable.

Quantitative theories of information-flow security allow one to limit how much information is actually leaked [52]. In the eighties, Millen [46] proposed to use the Shannon's channel capacity [51] for quantifying the capacity of covert channels. Later, attention was attracted by the development of new leakage measures that more closely express the danger of real attacks, most notably min-entropy [52] and g-leakage [7]. Generalizing the Shannon's capacity, a theory of channel capacity applicable to g-leakage has also been recently proposed [6].

For quantitative analysis of side channels, in general, Köpf and Basin [36] present an information-theoretic model of side-channel attacks that allows quantification of the information revealed to an attacker. Macé et al. [44] propose an approach for information-theoretic evaluation of side-channel resistant logic styles. Standaert et al. [53] present a framework for analysis of side-channel attacks that enables comparisons of different implementations wrt. side channels.

For quantitative analysis of timing side channels, Köpf and Backes [10] propose an approach for quantifying resistance to unknown-message side-channel attacks and use this approach to assess the resistance of cryptographic implementations against timing attacks. Köpf and Smith [39] derive leakage bounds for blinded cryptography under timing attacks. Doychev et al. [24] present a tool for automatic derivation of upper bounds on the cache side-channel leakage in x86 binaries, including cache-related side channels that are based on timing.

Yet, there seems to be a deficit of reports on *empirical* quantitative evaluation of timing side channels. We are aware only of the work by Cock et al. [19] who present an empirical evaluation of timing side channels on the seL4 microkernel. The results of our own research project contribute to this line of research.

Related to side channels, the problem of covert channels [42] has also attracted a lot of attention. For covert channels that are based on timing, there are reports on their informal [55], analytical [45], and empirical [27, 19] analysis.

Besides program transformations [2, 47, 13, 38], there is a spectrum of other techniques for controlling timing side channels. Hu [31] proposes to reduce timing channels by adding noise to the observable timing signal. Kocher [34] proposes blinding that unpredictably changes the correlation between the secret input of a cryptographic operation and its observable running time. Chevallier-Mames et al. [18] propose side-channel atomicity, a method to convert a cryptographic algorithm into an algorithm protected against simple side-channel attacks. Köpf and Dürmuth [37] improve blinding to allow a choice between the strength of the security guarantee and the resulting performance overhead. Svenningsson and Sands [54] present a method for controlled declassification of the side-channel leakage. Coppens et al. [20] propose to remove timing side channels by a transformation in a compiler backend. Askarov et al. [9] introduce black-box mitigators for controlling timing side channels in a system by delaying the system's outputs. Zhang et al. [56] leverage this approach for a programming language. Crane et al. [22] propose automated software diversity to mitigate cache side channels.

While performance costs of side-channel mitigation are generally addressed in the literature, e.g., in [47, 37, 20, 23, 56], a trade-off between the performance and security in this context is explored to a lesser extent. Köpf and Dürmuth [37] study such a trade-off for their countermeasure. Di Pierro et al. [23] investigate such a trade-off for a probabilistic variant of cross-copying, but only analytically. Doychev and Köpf [25] propose a game-theoretic approach for finding cost-effective configurations for countermeasures against side channels.

9 Conclusion

We presented the first systematic empirical evaluation of source-to-source transformations for removing timing side-channel vulnerabilities wrt. security and overhead. Our experimental results suggest that there are substantial differences between the transformations both in the introduced performance overhead and in the achieved reduction of timing side-channel capacities. In prior work, such transformations were analyzed mostly theoretically. However, it was speculated that some of the transformations are of unclear practical significance due to their potential inefficiency [54] or ineffectiveness [38]. In this research project, we obtain objective numbers that allow one to clarify such concerns wrt. one's own criteria of efficiency and effectiveness. Beyond this clarification, our findings provide guidance for choosing a suitable program transformation. Such choice is non-trivial because of the trade-off between security and performance.

Our work deepens the understanding about the effectiveness and efficiency of program transformations for removing timing side-channel vulnerabilities in practice, but this is only a first step in the empirical evaluation of such transformations. As future work, we will experimentally investigate effects of JIT compilation on program transformations. We also plan to consider alternative implementations of transformations as well as alternative measures of leakage.

Acknowledgements We thank Boris Köpf, David Sands, and the anonymous reviewers for valuable comments. We thank Patrick Metzler for help in the early phase of this work. This work has been partially funded by the DFG as part of project E2 within the CRC 1119 CROSSING and by CASED (`www.cased.de`).

References

1. FlexiProvider (Version 1.7p7). `http://www.flexiprovider.de`, 2013.
2. J. Agat. Transforming out Timing Leaks. In *POPL'00*, pages 40–53. ACM, 2000.
3. J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Chalmers University of Technology, 2000.
4. N. J. AlFardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *S&P'13*, pages 526–540. IEEE, 2013.
5. J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software Verification for Weak Memory via Program Transformation. In *ESOP'13*, LNCS 7792, pages 512–532. Springer, 2013.
6. M.-S. Alvim, K. Chatzikokolakis, A. McIver, C. Morgan, C. Palamidessi, and G. Smith. Additive and Multiplicative Notions of Leakage, and Their Capacities. In *CSF'14*, pages 308–322. IEEE, 2014.
7. M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring Information Leakage Using Generalized Gain Functions. In *CSF'12*, pages 265–279. IEEE, 2012.
8. S. Arimoto. An Algorithm for Computing the Capacity of Arbitrary Discrete Memoryless Channels. *IEEE Trans. on Inf. Theory*, 18(1):14–20, 1972.
9. A. Askarov, D. Zhang, and A. C. Myers. Predictive Black-Box Mitigation of Timing Channels. In *CCS'10*, pages 297–307. ACM, 2010.
10. M. Backes and B. Köpf. Formally Bounding the Side-Channel Leakage in Unknown-Message Attacks. In *ESORICS'08*, LNCS 5283, pages 517–532. Springer, 2008.
11. M. Baron. *Probability and Statistics for Computer Scientists*. CRC Press, 2006.
12. G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure Multi-Execution through Static Program Transformation. In *FMOODS/FORTE'12*, LNCS 7273, pages 186–202. Springer, 2012.
13. G. Barthe, T. Rezk, and M. Warnier. Preventing Timing Leaks Through Transactional Branching Instructions. In *QAPL'05*, pages 33–55. Elsevier, 2006.
14. R. E. Blahut. Computation of Channel Capacity and Rate-Distortion Functions. *IEEE Trans. on Inf. Theory*, 18(4):460–473, 1972.
15. B. B. Brumley and N. Tuveri. Remote Timing Attacks Are Still Practical. In *ESORICS'11*, LNCS 6879, pages 355–371. Springer, 2011.
16. R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *J. of the ACM*, 24(1):44–67, 1977.
17. K. Chatzikokolakis, T. Chothia, and A. Guha. Statistical Measurement of Information Leakage. In *TACAS'10*, LNCS 6015, pages 390–404. Springer, 2010.
18. B. Chevallier-Mames, M. Ciet, and M. Joye. Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. *IEEE Trans. on Computers*, 53(6):760–768, 2004.
19. D. Cock, Q. Ge, T. C. Murray, and G. Heiser. The Last Mile: An Empirical Study of Timing Channels on seL4. In *CCS'14*, pages 570–581. ACM, 2014.
20. B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *S&P'09*, pages 45–60. IEEE, 2009.
21. T. M. Cover and J. A. Thomas. *Elements of Information Theory, 2. ed.* Wiley, 2006.
22. S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *NDSS'15*. The Internet Society, 2015.

23. A. Di Pierro, C. Hankin, and H. Wiklicky. Probabilistic Timing Covert Channels: To Close or Not to Close? *Int. J. of Inf. Sec.*, 10(2):83–106, 2011.
24. G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security'13*, pages 431–446. USENIX, 2013.
25. G. Doychev and B. Köpf. Rational Protection Against Timing Attacks. In *CSF'15*. IEEE, 2015.
26. K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In *CHES'01*, LNCS 2162, pages 251–261. Springer, 2001.
27. R. Gay, H. Mantel, and H. Sudbrock. An Empirical Bandwidth Analysis of Interrupt-Related Covert Channels. In *QASA'13*, 2013.
28. A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *OOPSLA'07*, pages 57–76. ACM, 2007.
29. P. B. Guttoski, M. S. Sunyé, and F. Silva. Kruskal's Algorithm for Query Tree Optimization. In *IDEAS'07*, pages 296–302. IEEE, 2007.
30. D. Hedin and D. Sands. Timing Aware Information Flow Security for a JavaCard-like Bytecode. *El. Notes in Th. Comp. Science*, 141(1):163–182, 2005.
31. W.-M. Hu. Reducing Timing Channels with Fuzzy Time. In *S&P'91*, pages 8–20. IEEE, 1991.
32. J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. In *ESORICS'98*, LNCS 1485, pages 97–110. Springer, 1998.
33. D. Knuth. Structured Programming with Go to Statements. *ACM Computing Surveys*, 6(4):261–301, 1974.
34. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO'96*, LNCS 1109, pages 104–113. Springer, 1996.
35. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO'99*, LNCS 1666, pages 388–397. Springer, 1999.
36. B. Köpf and D. A. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *CCS'07*, pages 286–296. ACM, 2007.
37. B. Köpf and M. Dürmuth. A Provably Secure and Efficient Countermeasure against Timing Attacks. In *CSF'09*, pages 324–335. IEEE, 2009.
38. B. Köpf and H. Mantel. Transformational Typing and Unification for Automatically Correcting Insecure Programs. *Int. J. of Inf. Sec.*, 6(2–3):107–131, 2007.
39. B. Köpf and G. Smith. Vulnerability Bounds and Leakage Resilience of Blinded Cryptograph under Timing Attacks. In *CSF'10*, pages 44–56. IEEE, 2010.
40. J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proc. of the American Math. Soc.*, 7(1):48–50, 1956.
41. X. Lai. *On the Design and Security of Block Ciphers.* PhD thesis, ETH Zürich, 1992.
42. B. W. Lampson. A Note on the Confinement Problem. *Commun. ACM*, 16(10):613–615, 1973.
43. A. Lux and A. Starostin. A Tool for Static Detection of Timing Channels in Java. *J. of Crypt. Eng.*, 1(4):303–313, 2011.
44. F. Macé, F.-X. Standaert, and J.-J. Quisquater. Information Theoretic Evaluation of Side-Channel Resistant Logic Styles. In *CHES'07*, LNCS 4727, pages 427–442. Springer, 2007.
45. H. Mantel and H. Sudbrock. Comparing Countermeasures against Interrupt-Related Covert Channels in an Information-Theoretic Framework. In *CSF'07*, pages 326–340. IEEE, 2007.
46. J. K. Millen. Covert Channel Capacity. In *S&P'87*, pages 60–66. IEEE, 1987.

47. 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'06*, LNCS 3935, pages 156–168. Springer, 2006.
48. D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA'06*, LNCS 3860, pages 1–20. Springer, 2006.
49. J.-J. Quisquater and D. Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *E-Smart'01*, LNCS 2140, pages 200–210. Springer, 2001.
50. R. L. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
51. C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948.
52. G. Smith. On the Foundations of Quantitative Information Flow. In *FOSSACS'09*, LNCS 5504, pages 288–302. Springer, 2009.
53. F.-X. Standaert, T. Malkin, and M. Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In *EUROCRYPT'09*, LNCS 5479, pages 443–461. Springer, 2009.
54. J. Svenningsson and D. Sands. Specification and Verification of Side Channel Declassification. In *FAST'09*, LNCS 5983, pages 111–125. Springer, 2009.
55. J. C. Wray. An Analysis of Covert Timing Channels. In *S&P'91*, pages 2–7. IEEE, 1991.
56. D. Zhang, A. Askarov, and A. C. Myers. Language-Based Control and Mitigation of Timing Channels. In *PLDI'12*, pages 99–110. ACM, 2012.

A Source Code of Benchmarks

```
public int shareValue(int[] ids, int[] qty) {
  shareVal = 0;
  int i = 0;
  while (i < ids.length) {
    int id = ids[i];
    int val = lookupVal(id) * qty[i];
    if (id == SPECIAL_SHARE)
      shareVal = shareVal + val;
    i++;
  }
  return shareVal;
}
```

Figure 8. Benchmark program shareValue, the critical conditional is highlighted.

```
public int[] kruskal(int[] g) {
  int[] mst = new int[g.length];
  par = new int[g.length];
  for (int i = 0; i < par.length; ++i) {
    mst[i] = −1;
    par[i] = i;
  }
  int idx = 0;
  for (int i = 1; i < g.length; i += 2) {
    int src = find(g[i]);
    int tgt = find(g[i + 1]);
    if (src != tgt) {
      mst[++idx] = src;
      mst[++idx] = tgt;
      par[src] = tgt;
    }
  }
  mst[0] = idx / 2 + 1;
  return mst;
}

private int find(int x) {
  if (par[x] != x)
    return find(par[x]);
  return x;
}
```

Figure 9. Benchmark program kruskal, the critical conditional is highlighted.

```
private int mulMod16(int a, int b) {
  int p;
  a &= mulMask;
  b &= mulMask;
  if (a == 0) {
    a = mulModulus − b;
  } else if (b == 0) {
    a = mulModulus − a;
  } else {
    p = a ∗ b;
    b = p & mulMask;
    a = p >>> 16;
    a = b − a + (b < a ? 1 : 0);
  }
  return a & mulMask;
}
```

Figure 10. Benchmark program mulMod16, the critical conditional is highlighted.