Tool Support for Step-Wise Analysis and Mitigation of Timing Channels in Java

by

Patrick Metzler

January 31, 2013

Examiner: Prof. Dr.-Ing. Heiko Mantel
Advisor: Dr. Artem Starostin
modified version
Erklärung

Hiermit versichere ich gemäß der Allgemeinen Prüfungsbestimmungen der Technischen Universität Darmstadt (APB) §23 (7), die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 31. Januar 2013

______________________________
Patrick Metzler
Abstract

Timing side channels—an infamous class of security vulnerabilities resulting in leakage of confidential information through differences in the running time of a system—constitute an active research topic since the mid 1990’s. Mitigation techniques for such side channels include program transformations intended to remove any dependence of a program’s execution time on its confidential inputs. However, results by Coppens, Verbauwheide, Bosschere, and Sutter [CVBS09] show that some of the proposed program transformations do not account for microarchitectural influences and therefore leave a program vulnerable to timing attacks. This thesis investigates three selected program transformations presented in the literature [Aga00, MPSW05, BRW06] in the context of programs executed on a Java virtual machine. By the help of information theoretic results of Chatzikokolakis, Chothia, and Guha [CCG09], we are able to quantitatively assess the effect of each program transformation on information leakage.

In particular, we developed a sample-based analysis framework in order to characterize the timing behavior of a given Java program and quantitatively estimate the induced information leakage. We present four case studies including non-trivial Java programs with side channels which are transformed by the selected program transformations and analyzed by our framework. We evaluate the effects of each program transformation with respect to mitigation of information flow and induced execution time overhead. Additionally, we manually replace certain program fragments in a step-wise manner, allowing conclusions about which fragments contribute to a timing channel.

Our results include an empirical confirmation of the inappropriateness of two of the three selected program transformations for Java virtual machines with just-in-time compilers. Furthermore, we identify timing channels caused by recursion and other general program elements.
# Contents

1 Introduction  
   1.1 Related Work .............................................. 2

2 Preliminaries  
   2.1 Side Channel Finder ........................................... 5  
   2.2 Prior Work of the Author ..................................... 6  
   2.3 Program Transformations  
      2.3.1 Cross-Copying .......................................... 7  
      2.3.2 Conditional Assignment .............................. 8  
      2.3.3 Transactional Branching ............................ 8  
      2.3.4 Unification .......................................... 9  
   2.4 Problem Statement ........................................... 9

3 Analysis Framework for Timing Channels  
   3.1 Preliminaries on Information Theory .................... 11  
   3.2 Modeling Timing Channels as Discrete Memoryless Channels  
      3.3 Anonymity Engine ......................................... 14  
   3.4 Timing Analysis ............................................. 15  
   3.5 Attacker Model ............................................. 18  
   3.6 Workflow for Using the Framework ....................... 19

4 Case Studies  
   4.1 Case Study 1: Modular Exponentiation ................... 23  
   4.2 Case Study 2: ShareCount ................................ 26  
   4.3 Case Study 3: Kruskal’s Algorithm ..................... 30  
   4.4 Case Study 4: IDEA ......................................... 37  
   4.5 Summary .................................................... 41

5 Conclusion  
   5.1 Summary of Experiment Results ......................... 43  
   5.2 Modifying a JVM JIT Compiler .......................... 44  
   5.3 Summary of Contributions ................................ 47  
   5.4 Design Decisions ........................................... 48

A Source Code and Experiment Results  
   A.1 Case Study 1: Modular Exponentiation .................. 57  
   A.2 Case Study 2: ShareCount ................................ 61  
   A.3 Case Study 3: Kruskal’s Algorithm ..................... 67
Chapter 1

Introduction

When considering systems with security requirements, substantial security flaws may be introduced by side channels—observable properties of the physical implementation of a system which are not included in the system’s model, e.g., timing behavior or power usage. A timing side channel (in the remainder simply timing channel) is present if information about the confidential data processed by a system can be obtained by observing its execution time. Timing channels have been an active research topic since the mid 1990’s [Koc96]. A timing attack measures the execution time of a system, usually many times, and draws conclusions about the confidential data processed. Such attacks have been reported on both hard- and software level and on different cryptographic algorithms (e.g., [DKL+98, BR05, BT11, LS11]).

In the domain of software, possible mitigation techniques of timing channels lie at the source code level, where program transformations are applied such that the execution time of different branches becomes indistinguishable [Aga00, BRW06, KM05], or such that branches are avoided completely [MPSW05]. Timing channels may also be introduced at microarchitectural level [CVBS09], where influences such as instructions with varying execution time or branch predictors [AScKK06] can be identified as a cause for timing channels. This may make a program vulnerable to a timing attack even if no timing channel is visible at the source code level. Another non-trivial source of timing channels are virtual machines, e.g., a Java virtual machine (JVM), where programs can be both interpreted or executed as native code using a just-in-time (JIT) compiler. Here, interpretation may hide timing channels while they become observable if the program is executed as native code, as it is the case in [LS11]. As these examples indicate, it is in general challenging to qualitatively determine the presence or absence of a timing channel in a program.

Alternative approaches use quantitative information flow theory in order to derive upper bounds for the information flow through a side channel. Known metrics for such computations rely on concepts such as, e.g., Shannon entropy or channel capacity [Sha48]. The relation between inputs and outputs of a program is derived statically, e.g., using a model checker [BKR09], or dynamically, e.g., by profiling or simulating [KB11].

A related sampling-based technique, developed by Chatzikokolakis, Chothia, and Guha [CCG10], uses mutual information to measure the anonymity of a system’s secret inputs. The technique was implemented in the Anonymity Engine
This thesis uses Anonymity Engine in order to practically evaluate different program transformations reported in the literature, namely cross-copying [Aga00], conditional assignment [MPSW05], and transactional branching [BRW06]. We will analyze, at the example of Java programs, which transformations achieve their goal of mitigating timing channels in practice. In order to perform the analysis, we developed a framework for empirical analysis of the timing behavior of a program and quantifying the information leaked. We will apply our analysis on four case studies, containing Java programs of different conceptual complexity which we transform on the source code level.

Such an analysis can be thought of as an assistance during the implementation of a program processing confidential data. Following a step-wise software engineering approach, the current implementation of a program is first analyzed. The result supports the decision at which spots in the program and to which extent mitigation techniques are applied. Reanalyzing the transformed program facilitates finding a tradeoff between the amount of the acceptable leakage and a potential performance drawback introduced by the transformation.

Outline. Following a brief discussion of related work, we proceed by introducing into the work preceding this thesis, describing the program transformations from the literature we will consider, and stating our hypothesis. The third chapter describes our measuring framework and the quantitative analysis. We introduce four case studies, investigate them with respect to the program transformations, and present experimental results in chapter four. Finally, in chapter five, we state our conclusion, give an overview of most important design decisions, and summarize our observations.

1.1 Related Work

Besides the aforementioned literature, we provide an overview of related work from two perspectives: first, modeling, detection, and mitigation of timing channels and second, quantification of information flow through a timing channel. Further related are works concerned with the empirical confirmation of timing-based communication channels and related influences by hardware architectures.

Formal methods for timing channel security. In the context of programming languages, early notions of noninterference and corresponding type systems for programs were developed by Volpano and Smith, e.g., in [SV98]. Related to this approach, several security conditions and corresponding program transformations for mitigation of information leakage have been proposed. They include the works of Agat [Aga00], Sabelfeld, and Sands [SS00], Köpf and Mantel [KM05], Molnar, Piotrowski, Schultz, and Wagner [MPSW05], and Barthe, Rezk and Warnier [BRW06]. While [SV98] requires conditionals with a confidential guard to be executed atomically, the approach of [Aga00] and [SS00] renders this requirement superfluous through cross-copying the low program slices of both branches. A further development is provided in [KM05], achieving observational equivalence by unification and thereby avoiding nested transformations. In contrast to these four approaches, [MPSW05] introduces a security condition capturing the observability of the exact control flow of a program;
accordingly, their program transformation eliminates certain branchings on con-
fi-dential data completely and replaces the branches by conditional assignments,
instead of making them observationally equivalent and leaving the control flow
structure unchanged. A step towards a richer language containing objects and
exceptions is taken by [BRW06] through transactional branching. A further
refinement of the underlying model for timing behavior of program instructions
was presented by Hedin and Sands [HS05]; here, the duration of an instruc-
tion may depend on both its arguments and the history of previously executed
instructions, superceding models where, e.g., all instructions require the same
amount of time.

Further notions of timing channels, independent of programs but considering
systems as a black box is given by Askarov, Zhang, and Myers in [AZM10], where
a mitigation of timing channels is achieved by delaying output of the system such
that output is observable only in certain time slots.

Quantitative information flow. As summarized by Smith [Smi09], informa-
tion theory and various measures built on Shannon entropy are used to quan-
titatively analyze information flow. Backes, Köpf, and Rybalchenko [BKR09]
quantify the information contained in an equivalence relation on secret inputs
induced by a program, assuming an attacker capable of observing the exact con-
trol flow of the program. Parametrized in the observable properties of a program
is the method of Köpf and Basin [KB11], who model an adaptive timing channel
attack and provide a tailored measure of resistance against such attacks.

Further related work. The observations made during the author’s previous
and current work which have lead to the main hypothesis of this thesis corre-
spond to the work of Coppens, Verbauwhede, Bosschere, and Sutter [CVBS09]
on microarchitectural effects on timing channels, in particular the influences of
conditional jumps.

Regarding the technical aspects of this thesis, Gay [Gay08] is likewise con-
cerned with the empirical quantification of communication over timing based
communication channels, although focused on interrupt-based covert channels.
Chapter 2

Preliminaries

This chapter introduces the work preceding this thesis. We describe the observations of unexpected timing behavior of the IDEA cipher implemented in Java, which motivated a further analysis of timing channels in Java programs. Furthermore, a brief description of the program transformations investigated as timing channel mitigation techniques during our case studies is given. We conclude this chapter by stating our hypothesis about the effects of these program transformations on timing channels.

2.1 Side Channel Finder

In [LS11], Lux and Starostin present the Side Channel Finder, a tool which can be used to detect control flow related side channels in Java programs. Here, a program is considered to contain a potential timing channel if it contains a control flow branching that depends on confidential data according to a given security policy. Control flow branchings occur, e.g., at conditionals, loops, or polymorphic method calls. In the following, we will refer to an arbitrary control flow branching simply as branching. Side Channel Finder analyzes programs at the source code level, accepting a program if it is free of such branchings and reporting the problematic location if a program contains an untypable branching.

In order to demonstrate Side Channel Finder, it was applied on a Java implementation of the block cipher IDEA, contained in the cryptographic library FlexiProvider [Fle03]. This side channel, reported originally in [KSWH98], occurs in implementations of IDEA where a multiplication is skipped for certain inputs. Side Channel Finder correctly detects a branching dependent on the secret key in the modular multiplication implemented in method mulMod16, depicted in listing 2.1. Here, formal parameter a holds a part of the ciphertext and formal parameter b holds a part of the secret key. Since a depends on the secret key from previous rounds, the branching in line 7 indeed constitutes a potential timing channel. In order to determine whether this potential timing channel can be exploited and used to derive information about the secret key, the first part of the ciphertext-only attack of [KSWH98] was implemented. This attack records, for sufficiently many encryptions, execution time and resulting ciphertext. These ciphertexts are clustered, where all ciphertexts of a specific cluster
have the last 16 bits in common, which are the result of a modular multiplication (corresponding to the last execution of method `mulMod16`). The common 16 bits of the partition comprising the lowest average execution time are assumed to be the result of a multiplication where the first multiplicand equals zero. This is justified by the implementation of method `mulMod16` which reduces otherwise costly computations to a single subtraction in this special case, cf. the branch in line 8. Since the second multiplicand consists of bits 70–85 of the secret key, it is possible to compute this part of the key if the correct cluster is chosen. A more detailed description of this attack is given in [LS11].

The experiment result showed the attack on the FlexiProvider implementation to be successful. However, an unexpected modification of the attack was necessary in order to derive the correct key from the measurement, as a higher execution time was observed in cases where the smaller branch in line 8 was taken. Furthermore, attacks were only successful with enabled JIT compiler in the Java virtual machine. These circumstances were further investigated by the author prior to this thesis.

### 2.2 Prior Work of the Author

In order to further examine the IDEA experiment described above, the author was concerned with evaluating the effect of timing channel mitigation techniques on the attack. In a first step, the experiment was repeated on method `mulMod16` transformed with cross-copying [Aga00], conditional assignment [MPSW05], and transactional branching [BRW06]. This showed a successful attack for cross-copying and transactional branching, while conditional assignment prevented a successful attack if implemented strictly without branchings; implementing it even with no more branchings than on the expression level (by use of the ternary `_?_:` operator) lead to a successful attack.
As described above, general preconditions for a successful attack were both an enabled JIT compiler and a modification of the attack. The original attack in [KSWH98] is based on the assumption that skipping computations during modular multiplication leads to a shorter execution time. Therefore, in a direct application of the theoretical attack to the FlexiProvider implementation, the cluster holding the lowest average execution time is selected and associated to the branch in line 8 of listing 2.1. In contrast to this assumption, selecting the cluster with the highest average execution time, which constitutes the mentioned modification, lead to a successful attack.

In conjunction with the fact that the execution frequency of the first branch (line 8) is considerably lower than the execution frequency of the third branch (beginning at line 12), these observations lead to the assumption that the processor’s branch predictor caused an extended execution time of the first branch and by this rendered the control flow observable. Indeed, further experiments revealed a correlation between the execution time of IDEA and the number of branch mispredictions.

We conclude that multiple aspects of a real system, both at hardware level, e.g., through branch predictors, and at software level, e.g., through JIT compilers, may introduce or amplify timing channels which are not closed by certain program transformations. This agrees with theoretical considerations made by Acı́ncmez, Seifert, and Koč [AScKK06] and the finding of Coppens, Verbauwhede, Bosschere, and Sutter [CVBS09]. To which extent this applies to selected program transformations will be investigated in this thesis, where the program transformations described in the following section are considered.

2.3 Program Transformations

During each case study, we will apply the program transformations cross-copying, conditional assignment, and transactional branching to the source code of a Java program. The timing behavior of those programs will then be analyzed and compared. The source code of all untransformed and transformed programs occurring in the case studies are listed in the appendix, starting from page 56. The following gives an overview of the transformations we consider and discusses how we adapted these transformations to Java programs.

2.3.1 Cross-Copying

Cross-copying, presented by Agat [Aga00], is a program transformation defined for a simple while language. It contains the special statement \texttt{skipAsn}, which has the same timing behavior as a regular assignment statement, but has no effect on the memory. For example, the program

\begin{verbatim}
if h then x := 1
\end{verbatim}

would be transformed to

\begin{verbatim}
if h then x := 1 else skipAsn x 1.
\end{verbatim}

An implementation of this transformation was done by Agat for Java bytecode [Aga01], using the \texttt{pop} bytecode instruction to realize the \texttt{skipAsn} statement. However, it was designed assuming that the bytecode is entirely interpreted, i.e., no JIT compiler is involved. We consider this to be an unrealistic
restriction, but use cross-copying nevertheless, to investigate whether it amplifies or mitigates a timing channel. In absence of a better solution, we use assignments to dummy variables which do not affect original computations in order to implement the \texttt{skipAsn} statement.

### 2.3.2 Conditional Assignment

Molnar et al. [MPSW05] introduced conditional assignment as a technique to secure programs against adversaries capable of observing the exact control flow of the program. It removes branchings, such that both branches are subsequently executed. Program semantics are preserved by transforming the branching condition into a bit mask in order to compute values dependent on this condition using logical bit-wise operators. Function \texttt{mask} converts a boolean expression, i.e., the branch condition \( c \), into an \texttt{int} value with all bits set or unset, respectively. Whenever an assignment depends on a secret, the value assigned is calculated by

\[
\text{cond}(m,a,b) \equiv (m \& b) | (\neg m \& a), \quad \text{where } m \equiv \text{mask}(c),
\]

which returns \( b \) if \( c \) evaluates to true and \( a \) otherwise. Applying conditional assignment to the example program on the left yields the program on the right:

```plaintext
int p = 0; int r = 0;
if ( c ) { p = q; }
else { r = s; }
```

```plaintext
int p = 0; int r = 0;
m = mask(c);
p = cond( m, q, p );
r = cond( m, s, r );
```

While \texttt{mask} can be easily implemented without branchings in languages such as C (\texttt{mask}(c)\equiv-\neg c), it is not as straightforward in Java: type casting from \texttt{boolean} to \texttt{int} is not allowed. To overcome this, a technique developed by the author in his previous work is used. We reduce boolean expressions of integer comparisons to \texttt{a>b} for integers \( a, b \) and define \texttt{mask} by \((b-a)>>31\). This technique relies on the fact that Java uses two’s complement representation for integers and \texttt{>>} is a sign-extending shift. Although this implementation of \texttt{mask} covers only a subset of possible boolean expressions, it is sufficient for the programs considered in this work. In comparison to cross-copying and transactional branching, a unique property of conditional assignment is the complete elimination of branching statements (on source code level), as long as \texttt{mask} is implemented without branchings. Our implementation of \texttt{mask} meets this requirement, thus enables us to preserve this property. The corresponding security condition introduced by Molnar et al. is called program counter security (PC-security) and expresses indistinguishability of program executions for an observer with knowledge of all values of the program counter during the program execution.

### 2.3.3 Transactional Branching

Transactional branching, proposed by Barthe, Rezk, and Warnier [BRW06], takes a step towards support of richer languages. Indeed, it supports methods and even exceptions, but requires the language to provide a transaction functionality. An exemplary transformation is given by the following.
if e then c1;
    else c2;

if e then beginT; c2; abortT; beginT; c1; commitT
    else beginT; c1; abortT; beginT; c2; commitT

Here, abortT ends a transaction dismissing all changes made since beginT. Accordingly, commitT ends a transaction making all changes since beginT effective. A virtual machine appears to be well-suited to provide such a transaction interface. However, for Java standard edition, such interfaces exist only for large-scaled databases, which seemed inappropriate to use in our case studies. Therefore, we implement beginT, abortT, and commitT ourselves through methods keeping copies of variables not committed yet.

2.3.4 Unification
Köpf and Mantel introduced unification [KM06], concerning the problem of code explosion which arises with cross-copying and is inherited by transactional branching. In contrast to cross-copying, where nested branches induce an exponential code growth, nested branches can be omitted with unification. Since case studies selected for this thesis do not contain nested branches that depend in confidential data, unification is not in the scope of this work.

2.4 Problem Statement
In the context of the author’s previous work, our goal is to analyze the effects of program transformations on a wider range of Java programs. In particular, we will investigate to which extent each individual program transformation mitigates a timing channel and which amount of performance tradeoff it causes.

Given the observations made with the IDEA attack, it seems that not all of the previously described program transformations give an appropriate degree of security against timing attacks. For the special case of the IDEA implementation of FlexiProvider, our previous work demonstrated that timing channels may be caused by factors hardly detectable by a static analysis such as hard- or software particularities. We have attempted to capture those factors by choosing an empiric evaluation in order to assess for a given Java program whether it contains a timing channel. This approach allows us to consider run-time behavior of the program which is not visible during a static analysis, e.g., the frequency a certain branch is executed, and specific timing behavior of the environment. One drawback of profiling analyses is the absence of generality—an assertion by a profiling analysis is only valid for one particular environment.

Concerning program transformations on Java programs as countermeasures against timing channels, cases might occur where even no transformation closing a particular timing channel is known. In such a case, it might be desirable to mitigate timing channels partially such that a certain security requirement is met. As an example, it might be permissible for a system to leak a secret key if a reasonable lower bound can be found for when the key is leaked completely and must be replaced by a new key. In order to enable the approach of partial mitigation, quantifying the information flow through a side channel can be used. In contrast to a binary approach which either accepts or rejects a
program depending on whether leakage can be found, quantitative information flow analysis enables us to compare different mitigation techniques with respect to their effect and performance.

We expect that applying the three considered program transformations to further programs with timing channels gives analogous results to the example of IDEA. In particular, our hypothesis is that, given a program containing a timing channel, neither transforming it with cross-copying nor with transactional branching closes the timing channel, while applying conditional assignment does.
Chapter 3

Analysis Framework for Timing Channels

This chapter presents our approach for a quantitative analysis of timing channels. The analysis is divided into a timing analysis and a subsequent quantitative information flow analysis. For the latter, we deploy Anonymity Engine \cite{Cho08}, a tool approximating the information flow of a system based on observation samples. The timing analysis collects such samples by profiling the program under analysis whose information leakage through timing behavior shall be investigated. After refining the sampled data, it is passed to Anonymity Engine which estimates the information flow according to the observations.

We begin by giving basic concepts of information theory like entropy, mutual information, and communication channels. These notions will then be used to establish a model of timing channels. Subsequently, we will describe the timing analysis and its interaction with Anonymity Engine. We will discuss the assumed observer capabilities and conclude by proposing a workflow for the application of our analysis framework.

3.1 Preliminaries on Information Theory

Information theoretic channels provide well-known means for quantifying information flow from a sender to a receiver. The concept of channels can also be applied to the quantitative analysis of side channels, as done by, e.g., \cite{Mil87}. Likewise, we will model the information flow from a program’s input to observations of its running time as a discrete memoryless channel. The following briefly discusses the information theoretic notions necessary to establish our model; definitions used here are extracts of \cite[chapter 2 and 7]{CT06} by Cover and Thomas.

We begin by clarifying our notation of basic probability theoretic notions and define discrete channels.

Notations. Discrete random variables are denoted by capitals $X$ with co-domains (or alphabets) $\mathcal{X}$ and probability mass functions $p_X(x) := P(X = x)$ for $x \in \mathcal{X}$. The distribution function of $Y$ given $X$ is written as $p_{Y|X}(y|x) := P(Y = y|X = x)$. The joint distribution function of random variables $X$ and
Y is written as \( p_{X,Y}(x,y) := P(X = x \land Y = y) \). The number of elements contained in a set \( S \), i.e., its cardinality, is denoted by \(|S|\).

**Definition 3.1** (Discrete channel [Sha48]). A discrete channel, denoted by \((\mathcal{X}, p_{Y|X}(y|x), \mathcal{Y})\), consists of two finite sets \( \mathcal{X} \) and \( \mathcal{Y} \) and a collection of probability mass functions \( p_{Y|X}(y|x) \), one for each \( x \in \mathcal{X} \), such that for every \( x \) and \( y \), \( p_{Y|X}(y|x) \geq 0 \), and for every \( x \), \( \sum_y p_{Y|X}(y|x) = 1 \), with the interpretation that \( \mathcal{X} \) is the input and \( \mathcal{Y} \) is the output of the channel and that \( \mathcal{X} \) and \( \mathcal{Y} \) are the input and output alphabets, respectively. The collection \( p_{Y|X}(y|x) \) is called channel transition matrix.

A discrete channel is **memoryless** if the probability distribution of the output depends only on the input at that time and is conditionally independent of previous channel inputs or outputs.

The *entropy* of a random variable \( X \) is a measure of the amount of information contained in \( X \), or, in other words, the uncertainty about \( X \). The *conditional entropy* of a random variable \( Y \) given \( X \) expresses the remaining amount of information provided by \( Y \) if the value of \( X \) is known.

**Definition 3.2** (Entropy and conditional entropy [Sha48]). The entropy \( H(X) \) of a discrete random variable \( X \) is defined by

\[
H(X) = - \sum_{x \in \mathcal{X}} p_X(x) \log p_X(x).
\]

The conditional entropy \( H(Y|X) \) is defined as

\[
H(Y|X) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p_{X,Y}(x,y) \log p_{Y|X}(y|x).
\]

Mutual information \( I(X;Y) \) expresses the amount of information shared by \( X \) and \( Y \).

**Definition 3.3** (Mutual information [Sha48]). The mutual information between two discrete random variables \( X \) and \( Y \) is defined as

\[
I(X;Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p_{X,Y}(x,y) \log \frac{p_{X,Y}(x,y)}{p_X(x)p_Y(y)}.
\]

The value of \( I(X;Y) \) not only depends on the channel itself, i.e., the transition probabilities \( p_{Y|X}(y|x) \), but also on \( p_X \). The *channel capacity* is the maximal possible mutual information for a given channel.

**Definition 3.4** (Channel capacity [Sha48]). The channel capacity of a discrete memoryless channel is defined as

\[
C = \max_{p_X} I(X;Y).
\]

**Lemma 3.1** (Properties of entropy, mutual information and capacity).

1. \( H(X) \geq 0 \)
2. \( H(X) \leq \log |\mathcal{X}| \). \( H(X) = \log |\mathcal{X}| \) if and only if \( X \) is distributed uniformly over \( \mathcal{X} \).
3. $H(X|Y) \leq H(X)$

4. $I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$

5. $0 \leq I(X;Y) \leq H(X) \leq \log |X|$

6. $0 \leq C \leq \min\{\log |X|, \log |Y|\}$

Property 4 supports the intuition that the mutual information $I(X;Y)$ is the information gain, or reduction of uncertainty, about the one random variable after knowing the value of the other. Hence, mutual information is a notion of information flow through a channel: after knowing the output $Y$, the receiver has gained $I(X;Y)$ of information about the input $X$.

Example (Binary symmetric channel). The binary symmetric channel is given by input and output alphabets $\mathcal{X} = \mathcal{Y} = \{0, 1\}$ and the transition matrix

$$p_{Y|X}(y|x) = \begin{pmatrix} 1 - p & p \\ p & 1 - p \end{pmatrix}.$$  

We have $I(X;Y) = H(Y) - H(p) \leq 1 - H(p)$, where $H(p)$ denotes the binary entropy function and is defined by $-p \log(p) - (1-p) \log(1-p)$. For $p_X(0) = p_X(1) = \frac{1}{2}$, $I(X;Y) = 1 - H(p)$, hence $C = 1 - H(p)$.

3.2 Modeling Timing Channels as Discrete Memoryless Channels

Given the prerequisites from the previous section, we are now able to establish a model of timing channels as discrete memoryless channels. Our goal is to model the information flow from secret program inputs to an observer of the program’s timing behavior. This model should match an analysis which experimentally determines, for a given program and a set of program inputs, the observable execution time for each input. The information flow to the observer is then approximated as the channel capacity.

Input alphabet. Concerning the channel input, we are seeking an input alphabet $\mathcal{X}$ denoting the set of possible secrets about the program input. Among other options, we choose $\mathcal{X} = I$, where $I$ denotes a non-empty set of secret program inputs. An alternative, more general choice is $\mathcal{X} = I/\sim_p$, where $\sim_p \subseteq I \times I$ is an equivalence relation on (secret or public) program inputs and $I/\sim_p$ is the set of equivalence classes with respect to $\sim_p$. For example, if $i_1$ and $i_2$ are two integer inputs, $i_1 \sim_p i_2$ could denote that $i_1$ and $i_2$ have the same Hamming weight, i.e., the same number of set bits in their binary representation. Another example would be suitable where a single input has a public and a secret part; here, $i_1 \sim_p i_2$ could denote that the secret parts of $i_1$ and $i_2$ are equal. Obviously, this choice subsumes the previous one as $\sim_p$ can be chosen to be equality. As the difference between both options does not affect our analysis, we define $\mathcal{X} = I$ and leave it to the user of the framework to distinguish between the interpretation of $x \in \mathcal{X}$ as a representative of an equivalence class $[x] \in I/\sim_p$ or as a plain input $x \in I$. Note that the user of the framework is left the option to specify $I$ as the set of all possible inputs as well as to specify $I$ as a proper
subset of all program inputs. The latter could be chosen with the intention to reduce the time required by the analysis, however restricting, at the same time, the validity of the analysis result to only this subset of inputs.

**Output alphabet.** As our goal is to analyze information leaked through timing behavior, we let the output $Y$ model timing observations. We assume these timing observations to be discrete, which seems to be appropriate since we regard clocked digital systems. Let the random variable $T$ denote the number of time units elapsed between the invocation of the program with input $X$ and its termination. One possible definition of the channel output is $Y = T$. However, this choice probably leads to a large output alphabet, as various factors aside the program influence $T$, e.g., the scheduler and concurrently running programs, cache misses and hardware interrupts. Repeated executions of the same deterministic program, with the same inputs, may consequently each take a different time, hence, in the general case, we would have $|Y| > |X|$ for $Y = T$. As we will see later, the sample size necessary for a statistically solid analysis result linearly increases with increasing $|Y|$. In the interest of a feasible sample size, we therefore reduce $|Y|$ such that $|Y| = |X|$. This is achieved by an interpretation function $f : T \rightarrow Y$ grouping possibly multiple observed time values $t \in T$ to a single channel output $y \in Y$. We call this interpretation *partitioning* and the elements of $Y$ *partitions*, with the intuition that the time span between the shortest and longest observed execution times is divided into partitions and an observed time value is interpreted as the partition it lies in.

Due to lemma 3.1, 6, the reduction of $|Y|$ to $|X|$ does not affect the capacity $C$. Following the same argument, we do not choose $|Y| < |X|$ in order to prevent underapproximating $C$. Note that, although our choice of $|Y|$ does not affect $C$, inauspiciously defining $f$ could reduce $C$. Accordingly, the analysis result will only be valid under the assumption that the observer is not able to find an alternative interpretation function resulting in a higher capacity.

**Channel transition matrix.** The channel transition matrix corresponds to the timing behavior of both the program and the system it is executed on. By measuring execution times for each input, the timing analysis experimentally approximates, for each program individually, the channel transition matrix, i.e., for each input $x \in X$ and output $y \in Y$ the probability $p_{Y|X}(y|x)$.

**Assumptions.** As we use discrete memoryless channels as system models, the output $Y$ may only depend on the last realization of $X$. Hence, the timing behavior must neither depend on previous runs of the system, nor on other influences besides the program input. Nevertheless, we can model systems on which, e.g., concurrently running processes additionally influence a program’s execution time, if we assume that this influence can be neglected after repeated measures.

### 3.3 Anonymity Engine

Anonymity Engine [CCG09, Cho08] is a tool analyzing the quantitative information flow from the inputs to the outputs of a system. It models systems as discrete memoryless channels and calculates both mutual information
and an approximation of the channel capacity using the Blahut-Arimoto algorithm \cite{Ari72, Bla72}. It takes as input an unordered list of observations \( o \in X \times Y \) for channel input \( X \) and output \( Y \). From these observations, Anonymity Engine first estimates the channel transition matrix as

\[
p_{Y|X}(y|x) = \frac{\text{number of occurrences of } (x,y)}{\text{number of occurrences of } x}.
\]

This is followed by the calculation of mutual information and the approximation of capacity for these transition probabilities. With our timing analysis, we create such observations and use Anonymity Engine to analyze the information flow.

Given a channel transition matrix, mutual information is directly computable, while the channel capacity may, in general, only be approximated, e.g., by the Blahut-Arimoto algorithm as in Anonymity Engine. Nevertheless, as the channel transition matrix on which both calculations base is already an estimate of the actual transition probabilities, the mutual information calculated by Anonymity Engine is as well an approximation of the actual value. Moreover, the calculation of the approximation of capacity contains two sources of error, the estimation of transition probabilities and the approximation by the Blahut-Arimoto algorithm. Due to the results of Chatzikokolakis, Chothia and Guha about the distribution of mutual information, it is possible to assess the error of the approximations \cite{CCG09}. They prove the distributions of the estimated mutual information \( \hat{I}(X;Y) \) (and likewise of the estimated capacity \( \hat{C} \)) to have the mean

\[
I(X;Y) + O\left(\frac{1}{n^2}\right),
\]

where \( I(X;Y) \) denotes the actual mutual information of the analyzed system and \( n \) denotes the number of observations. Furthermore, the variance is bound above by

\[
\frac{|X||Y|}{n}.
\]

This upper bound increases linearly with the sizes of input and output alphabets and decreases linearly with the number of observations. It indicates the possible size of the input alphabet and the number of observations required for an analysis within the error tolerance of the user.

### 3.4 Timing Analysis

In order to derive the input for the information flow analysis through Anonymity Engine, i.e., the channel transition matrix for a given program, we provide a Java tool implementing the timing analysis. Additionally, this tool reports the overall average execution time allowing an assessment of the program’s performance. The timing analysis tool takes three inputs:

- a Java program \( P \),
- an algorithm, called \textit{input generator}, which generates a list of program inputs such that each \( x \in X \) occurs (possibly multiple times) in this list, and
Timing Analysis

- Input generation
- Profiling
- Partitioning
- Performance analysis

Anonymity Engine

- Distribution calculation
- Blahut-Arimoto algorithm

Figure 3.1: Overview of the analysis procedure. A program along with a corresponding input generator is evaluated by the timing analysis which produces a list of observations $o \in X \times Y$ relating program inputs $x \in X$ to time partitions $y \in Y$. Subsequently, Anonymity Engine estimates the channel transition matrix matching the observations, calculates the mutual information $\hat{I}(X;Y)$, and approximates the channel capacity $\hat{C} = \max_{p_X} \hat{I}(X;Y)$.
• the number of secret inputs $|\mathcal{X}|$ (which equals the number of partitions $|\mathcal{Y}|$),

along with three measurement parameters $r$, $w$, and $c$ which will be explained in the following. The specified program is executed with program inputs calculated by the input generator. For each input, the execution time of the program under analysis is measured and observations $o \in \mathcal{X} \times \mathcal{Y}$ are generated as input to Anonymity Engine.

**Analysis procedure.** Figure 3.1 illustrates the analysis procedure. In a first step, the input generator is executed which yields a list of $n$ program inputs. The second step consists of profiling the program with the given input. This is realized by selecting subsequently each input in the list. The inputs are either processed in a random order or a fixed order given by the input generator. For a single input, the program under analysis is executed for a number of rounds which is specified by the parameter $r$. The measured execution time is recorded for each round. These first two steps are repeated for a specified number of times $w$ with the intention to warm-up the system. Most notably, these warm-up passes increase the probability of a more intensive optimization by the JVM’s JIT compiler. Accordingly, the first two steps are executed $w+1$ times, where the results of the first $w$ executions are discarded. The first step is repeated as well, instead of generating the input only once, as the input generation may contain random elements. Following the last pass of input generation and profiling, the timing measurements are evaluated and partitioned. The evaluation consists of calculating the arithmetic mean of all values below or equaling the $c$-quantile of each round, with measurement parameter $c$: For each profiling round, the list of measurement results $t_1, \ldots, t_r$ is sorted in ascending order. For this list, the round-mean

$$\bar{t} := \frac{1}{\left\lceil \frac{r}{c} \right\rceil} \left( t_1 + \cdots + t_{\left\lceil \frac{r}{c} \right\rceil} \right),$$

disregarding higher measurements, is derived. The arithmetic mean of all round-means $\bar{t}_1, \ldots, \bar{t}_r$ in turn constitutes the result of the performance analysis

$$\bar{T}_P(w, r) := \frac{1}{r} \sum_{i=1}^{r} \bar{t}_i.$$

Consecutively, each round-mean $\bar{t}$ is assigned a partition $f(\bar{t}) \in \mathcal{Y}$, yielding pairs $(x, f(\bar{t})) \in \mathcal{X} \times \mathcal{Y}$ which are called observations. The list of all observations is subsequently passed to Anonymity Engine which estimates the channel transition matrix $p_{Y|X}(y|x)$ matching these observations. From these probabilities, the mutual information $\hat{I}(X; Y)$ is computed and the capacity approximation $\hat{C}$ is derived using the Blahut-Arimoto algorithm. If the information flow for the distribution of $X$ used during the analysis (which is implicitly specified by the generated input) is of interest, the mutual information is the appropriate analysis result. Otherwise, if the information flow for arbitrary input distributions should be found, the approximated capacity constitutes the analysis result. The analysis result in bits for program $P$ will be denoted by $I_P(w, r)$ and $C_P(w, r)$ for mutual information and capacity, respectively, depending on the measurement parameters $w$ and $r$. By $I(\cdot, \cdot)$, we will denote the mutual information analysis result for an unspecified program.
Partitioning. The partitioning function $f : \mathcal{X} \to \mathcal{Y}$ was designed so that it allows a first evaluation of the analysis framework, however considering, at the same time, that it is beyond the main focus of this work. Therefore, we refrained from implementing a sophisticated classification algorithm or distance measure. As a simplification, we parametrize the partitioning function with the input distribution $p_X$. Note that this does not imply that the observer has knowledge of the input distribution if he or she can otherwise mimic the partitioning function of the analysis.

Specifically, we define the partitioning function as follows: Let $\bar{t}_1, \ldots, \bar{t}_n$ denote the list of all round-means in ascending order. Furthermore, let $x_1, \ldots, x_m = \mathcal{X}$ and let $y_1, \ldots, y_m = \mathcal{Y}$ denote inputs and partitions. For input $x_i$, we abbreviate the expected frequency $p_X(x_i) \cdot n$ as $q_i$, hence we expect $q_i$ timing observations corresponding to input $x_i$. The partitions are defined according to

$$y_i := \left\{ j \in \mathbb{N} \mid \sum_{k=1}^{i-1} q_k < j \leq \sum_{k=1}^{i} q_k \right\}.$$  

Finally, each round-mean will be assigned the partition

$$f(\bar{t}_i) := \begin{cases} y_1 & \text{if } i \in y_1 \\ y_2 & \text{if } i \in y_2 \\ \vdots \\ y_m & \text{if } i \in y_m \end{cases},$$

hence, after partitioning, each partition is associated with a specific number of timing observations according to the distribution of $X$. With this definition, all partitions are non-empty, regardless of the values $\bar{t}_1, \ldots, \bar{t}_n$. In particular, it is possible that $\bar{t}_i = \bar{t}_j$ and $f(\bar{t}_i) \neq f(\bar{t}_j)$ hold at the same time.

Therefore, mutual information is over-approximated in some cases and an observer is imitated who guesses for some inputs with equal timing observations that they are caused by different inputs. This behavior could be undesirable in cases of a low capacity, however, it is not straightforward to map equal round-means into the same partition while still filling partitions according to the the input distribution.

Technically speaking, this partitioning violates the assumption that the channel output must not depend on previous runs of the system. However, we expect the partition boundaries to be approximately constant for a given system and hence could also be calculated in advance.

### 3.5 Attacker Model

Given the channel definition along with the analysis description, we can fix the corresponding observer capabilities:

1. The observer is capable of measuring the time span between the program invocation and termination as accurate as possible using the time source of the Java virtual machine.
2. The observer has knowledge of the input alphabet $|\mathcal{X}|$ and the channel transition matrix $p_{Y|X}(y|x)$. As a consequence, he or she is able to determine, for all $x \in \mathcal{X}$, the probability $p_{Y|X}(f(t)|x)$ that input $x$ lead to the timing observation $t \in \mathcal{T}$.

3. The observer is capable of partitioning timing observations $t \in \mathcal{T}$ ‘as good as’ the partitioning function $f$ used by the analysis framework, but not ‘better’. In other words, the observer is capable of computing a partitioning function $f'$ leading to the same conditional entropy $H(Y|X)$ (and hence the same capacity) as $f$ and all partitioning functions $f''$ computable by the observer lead to a conditional entropy $H'(Y|X) \leq H(Y|X)$.

For example, these observer capabilities match a setting where the observer derives timing observations by reading the time counter provided by the Java virtual machine via `System.nanoTime()` at the beginning and end of the program execution, knows the exact timing characteristics of the program and the system on which it is executed, and uses a partitioning method similar to the partitioning of the analysis.

### 3.6 Workflow for Using the Framework

Finally, we sketch an exemplary procedure for designing an experiment based on our analysis framework given a program $P$ whose information leakage shall be analyzed.

1. Specify the confidential and public parts of the program input space.

2. Specify the list of program inputs of length $n$ along with the measurement parameters $w$ and $r$ according to the following constraints:

   - $\frac{|\mathcal{X}|^2}{n}$ is an acceptable upper bound for the variance of the analysis result $I_P(w, r)$ for mutual information respectively $C_P(w, r)$ for the capacity.
   - The number of unique confidential inputs $|\mathcal{X}|$ either corresponds to the real number of confidential inputs in the program’s original environment or is an acceptable large subset of it.
   - $n = \sum_{i=1}^{|\mathcal{X}|} p_X(x_i)k$, for $k \in \mathbb{R}$ and $\forall 1 \leq i \leq |\mathcal{X}| : p_X(x_i)k \in \mathbb{N}$, i.e., the frequency of occurrence of each input in the input list corresponds to the input distribution $p_X$. Note that the public part of an input, if any, may change with repetitions of the same secret input part or may stay unchanged with each occurrence of an input.
   - The specified ranges for parameters $w$ and $r$ reflect the original environment of the program under analysis. For example, choose low values for $w$, e.g., 0 or 1, if the JVM of the program’s original environment is always freshly started for an execution of the program. In contrast, choose high values for $w$, if the program under analysis is executed numerous times without re-starting the JVM which usually implies that the program is well optimized by the JIT compiler. Choose $r$ according to the expected accuracy of the observer’s measurements.
• The expected execution time of a single analysis, dominated by the total number of program executions $n \cdot w \cdot r$, matches its time constraint.

3. Design an input generator according to the choice made above by implementing the according abstract Java class of the analysis tool.

4. Empirically determine a suitable value for the measurement parameter $c$ in order to reduce noise caused by processes concurrently running with the analysis tool.

5. Execute the analysis for all specified pairs of $w$ and $r$. 
Chapter 4

Case Studies

We introduce four case studies in order to illustrate the practical applicability of our analysis framework and evaluate our hypotheses. Each case study contains a program implemented in Java. These programs contain branchings depending on confidential inputs, i.e., potential timing channels. During each case study, the three program transformations—cross-copying, conditional assignment, and transactional branching—are applied to the program, where we use the following notation:

<table>
<thead>
<tr>
<th>Resulting program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untransformed</td>
</tr>
<tr>
<td>Cross-copying</td>
</tr>
<tr>
<td>Conditional assignment</td>
</tr>
<tr>
<td>Transactional branching</td>
</tr>
</tbody>
</table>

It will be evaluated how these transformations influence the timing behavior with respect to the amount of information leaked, using the methodology described in the previous chapter. We write $I_P(w, r)$ for the analysis result for program $P$ with $w$ warm-up passes and $r$ rounds per sample, i.e., the estimated mutual information between input and timing observations. For each experiment, we fix $w \in \{0, 5, 10\}$ and let $r$ range from 1 to 500, i.e., one experiment consists of 500 analyses with a fixed number of warm-up passes and an increasing number of rounds.

In regard to a straightforward comparison between transformed and untransformed programs, it is favorable that the untransformed program indeed contains a timing channel and furthermore that an observer can learn the complete input through this channel. Additionally, a uniform interpretation of analysis results helps comparing different case studies. Hence, we will define the following notion of information leakage. We introduce two properties of programs, the property of being leaking and being strongly leaking categorizing the severity of information leakage. In order to account for outliers in an experiment consisting of $n$ analyses, we consider, for $\gamma \in [0, 1]$, no more than the $\gamma n$ analyses showing the highest mutual information. Furthermore, we distinguish between analysis results $I_P(w, r)$ equaling the maximum of $H(X)$ and analysis results lying above a certain threshold $\epsilon H(X)$, $\epsilon \in [0, 1]$.

**Definition 4.1** ($\gamma$-leaking programs). Let $\epsilon, \gamma \in \{x \in \mathbb{R} : 0 \leq x \leq 1\}$. A program $P$, analyzed in an experiment with input $X$ and $\rho$ different values of
measurement parameter \( r \), is \( \gamma \)-leaking if

\[
P(w, r) \geq \epsilon H(X)
\]

for at least \( \lceil \gamma \rho \rceil \) different values of \( r \). \( P \) is strongly \( \gamma \)-leaking if it is \( \gamma \)-leaking and \( \epsilon = 1 \). A program is non-\( \gamma \)-leaking if and only if it is not \( \gamma \)-leaking.

We will use these properties to express, intuitively speaking, that a \( \gamma \)-leaking program \( P \) leaks a considerable amount of information and that a strongly \( \gamma \)-leaking program leaks all available information during sufficiently many analyses with the intuition that a program contains a timing channel if and only if it is \( \gamma \)-leaking. Clearly, a strongly \( \gamma \)-leaking program is also \( \gamma \)-leaking.

This notion of leakage might not be well-suited to compare arbitrary programs, however, it serves its purpose in uniformly comparing the case studies presented in this thesis. Thus, we provide values for \( \epsilon \) and \( \gamma \) which allow a reasonable interpretation of the results we obtained, but do not claim these values to be generally appropriate. For all case studies, we choose \( \gamma = \epsilon = \frac{1}{4} \).

Given the precondition that an untransformed program strongly \( \gamma \)-leaks, we render more precisely our hypotheses about the effects of program transformations on timing channels. As described in section 2.4, we expect conditional assignment to be the only program transformation which closes a timing channel in the presence of a JIT compiler. Our expectation is justified by the fact that conditional assignment is the only PC-secure transformation (cf. section 2.3.2). This leads to our hypotheses:

**Hypothesis 4.1.** For a strongly \( \gamma \)-leaking program \( P \), cross-copying and transactional branching do not close the timing channel, i.e., \( P_{CC} \) and \( P_{TB} \) are \( \gamma \)-leaking programs.

**Hypothesis 4.2.** For a strongly \( \gamma \)-leaking program \( P \), conditional assignment mitigates the timing channel, i.e., \( P_{CA} \) is not \( \gamma \)-leaking.

Furthermore, we expect that with an increasing number of measurements, corresponding to \( n \cdot w \cdot r \), the influence of noise is reduced, therefore \( I_P(w, r) \) increases.

**General experiment setup.** All experiments were conducted on an AMD E-300 APU with 1.3GHz and 4GB main memory. As operating system we used Ubuntu / GNU Linux, kernel version 3.2.0-29-generic together with the Java HotSpot 64-Bit Server virtual machine, version 1.7.0 update 7. The system was booted with the standard configuration except that the graphical system Xorg was disabled. The execution time of the respective program under analysis is measured using `System.nanoTime()`. This time measurement method provides the highest precision natively available in the Java Standard Edition. Its precision depends on the Java virtual machine and is higher than 300 ns with the setup used in the case studies presented here.

Listing [4.1] illustrates how the analysis described in chapter 3 is used to construct the experiments for our case studies. In line [6], `java` starts the 64bit JVM in the server mode (specified by `-server -d64`) and executes the analysis for a single combination of parameters. The subsequent parameters are passed to the analysis tool, where \( w \) and \( r \) specify the number of warm-up passes and measurement rounds, respectively, while the experiment specific parameters consist
Listing 4.1: Shell script specifying the experiment procedure.

```bash
#!/bin/bash

function experiment {
    for i in {0..5..10}; do
        for j in {1..500..1}; do
            sleep 0.5
            java -server -d64 -jar sta.jar -w$i -r $ \\
                ⟨experiment specific parameters⟩
        done
    done
}
}
```

of the program under analysis, the input generator, the number of partitions (corresponding to the number of distinct inputs), and the number of samples (pairs of input and output) to generate. Measurement parameter \( c \) is assigned its standard value \( c = \frac{1}{5} \). The \texttt{sleep} command in line 5 delays the experiment execution by 0.5 seconds before each analysis which is intended to reduce noise caused by, e.g., hardware latencies.

4.1 Case Study 1: Modular Exponentiation

Our first case study consists of a square-and-multiply algorithm for modular exponentiation. This algorithm was studied in the context of side channels in various works, e.g. by Kocher [Koc96]. It can be used, e.g., in implementations of cryptographic algorithms like RSA for an efficient exponentiation in a residue class ring. Here, it is used during decryption to compute \( m = c^d (mod \ n) \), where \( c \) denotes the ciphertext and \( d \) the secret key. Thus, an implementation must not leak the value of \( k \) through its timing behavior.

We implement the square-and-multiply algorithm in Java (depicted in listing 4.2), as an adaption of the code given in [Koc96] and denote the program by \( E \). The secret key is represented by the formal parameter \( d \) of type \texttt{int} and successively shifted to the right (line 7) while during each loop iteration, the least significant bit is tested (line 4). If it is set, an additional multiplication and modulo operation is performed (line 5).

Therefore, the conditional branching constitutes a potential timing channel, which can be automatically detected by a static analysis, like, e.g., in [LS11], since the branching depends on the value of \( d \). The channel leaks, up to any noise, the Hamming weight of the confidential key in one execution of the algorithm. Furthermore, Kocher describes an attack that guesses all bits of the key provided that multiple measurements are possible.

We denote our implementation of modular exponentiation by \( E \) and the programs resulting from applying cross-copying, conditional assignment, and transactional branching by \( E^{CC} \), \( E^{CA} \), and \( E^{TB} \), respectively. The listings of these programs are given in appendix A.1.

Experiment. In order to investigate the leakage through the conditional branching, inputs of different Hamming weights are created. We restrict our-
Listing 4.2: Modular exponentiation

```java
public void exp(long m, int d) {
    long r = 1;
    for (int i = 0; i < 32; i++) {
        if (d % 2 == 1)
            r = (r * m) % module;
        m = m * m % module;
        d >>= 1;
    }
    result = (int) (r % module);
}
```

selves to eight distinct inputs in order to obtain a feasible scope of the experiment. The Hamming weight of distinct input $i$ is given by $h(i) = 12 + i$, $i \in \{0, \ldots, 7\}$; the bit permutation of each input is randomly chosen. Each distinct input occurs 50 times in the input set, yielding a total of 400 inputs, which results in an upper bound for the analysis result’s variance of $\frac{|X|^2}{n} \approx 0.04 \cdot H(X)$. The maximal possible information flow, i.e., the maximal value of $I_E(w, r)$ we can expect, is given by $H(X) = 3$.

The results of the information flow analysis for modular exponentiation for 10 warm-up passes, i.e., $w = 10$, and rounds from 1 to 500, i.e., $r = 1, \ldots, 500$, are illustrated in figure 4.1. The remaining results for $w = 0$ and $w = 5$ are provided in appendix A.1 on pages 59 and following. The abscissa denotes parameter $r$ and the ordinate denotes the analysis result $I_E(w, r)$, i.e., the mutual information between inputs and observations.

First, we observe that our implementation of modular exponentiation is indeed a strongly $\gamma$-leaking program (in the case of $w = 10$), as $I_E(10, r)$ reaches the maximum of 3 bit 317 times. Second, transactional branching similarly results in a constantly high information flow, but slightly mitigated compared to $P$, hence $E^{TB}$ is a $\gamma$-leaking program. Third, results for cross-copying and conditional assignment show a similarly low information flow where $I_E^{CC}(10, r), I_E^{CA}(10, r) \leq 0.4$. Further investigating the single measurement results revealed that the value of 0.4 corresponds to the information flow calculated if all inputs lead to exactly the same observation, i.e., the execution time was independent of the inputs. The fact that $I_E(w, r) = 0.4$ rather than $I_E(w, r) = 0$ in this case is due to an implementation detail of the partitioning method; see chapter 3.4 for a more detailed discussion. These results approve hypothesis 4.2. However, hypothesis 4.1 is only partially approved as $E^{CC}$ is not $\gamma$-leaking, while we expected $E^{CC}$ to be a $\gamma$-leaking program since it still contains a conditional branching on confidential data. One speculative explanation is that the branch predictor adapted sufficiently accurate such that miss-predictions are not observable.

Examining the results for $w = 0$, we observe that $I_E(0, r)$ and $I_E^{TB}(0, r)$ increase monotonously with increasing $r$, which matches our expectation that a higher number of measurements leads to a higher information flow.

Figure 4.2 depicts the performance analysis results for $w = 10$ and $r = 1, \ldots, 500$. The remaining results for $w = 0$ and $w = 10$ are provided in appendix A.1 on page 60 and following. The abscissa denotes the average exe-
Figure 4.1: Mutual information $I_{E}(10, r)$ in bits with 10 warm-up passes for modular exponentiation untransformed ($E$), transformed with cross-copying ($E^{CC}$), conditional assignment ($E^{CA}$), and transactional branching ($E^{TB}$).

Figure 4.2: Average execution times $\bar{T}_{E}(10, r)$ in nano seconds with 10 warm-up passes for modular exponentiation untransformed ($E$), transformed with cross-copying ($E^{CC}$), conditional assignment ($E^{CA}$), and transactional branching ($E^{TB}$).
Listing 4.3: Share calculation example

```java
class ShareCalculationExample {
    public void count(int[] noDB, int[] valueDB) {
        long shareValue = 0;
        int i = 0;
        while (i < noDB.length) {
            int no = noDB[i];
            if (no == SHARE) {
                int value = valueDB[i];
                shareValue = shareValue + value;
            }
            i++;
        }
        result = shareValue;
    }
}
```

cution time \(T_P(w, r)\), i.e., the arithmetic mean of all round-means \(\bar{t}_1, \ldots, \bar{t}_{400}\).
For the approximately 10 smallest values of \(r\), \(T_P(10, r)\) ranges up to 6509 ns and decreases in an order of magnitude due to an ameliorating optimization through the JIT compiler. For higher values of \(r\), \(T_P(10, r)\) stabilizes and we have \(\bar{T}_{ECC}(10, r) > \bar{T}_{ECA}(10, r) > \bar{T}_{EAB}(10, r) > \bar{T}_E(10, r)\), with only a minor difference between the average execution time for cross-copying and conditional assignment. Transactional branching induces at the same time a relatively low performance and information flow reduction, while cross-copying and conditional assignment close the timing channel but result in a higher performance tradeoff.

### 4.2 Case Study 2: ShareCount

This case study is an adaption of an example given by Agat (Figure 1 of [Aga00]). The program, denoted by \(S\), iterates over two `int` arrays, the first of which, `noDB`, containing identifiers of shares and the second, `valueDB` providing values of shares. The program calculates the accumulated value of a specific share. During each iteration, the identifier at the current position in `noDB` is compared to the identifier `SHARE` of the share we are interested in (line 6). Only if the identifiers match, the according value is accumulated (line 8), which opens a potential timing channel.

We consider the number of occurrences of `SHARE` to be secret and specify the values of `noDB` as confidential input, which lets a static program analysis detect the branching in line six as a potential timing channel. Cross-copying, conditional assignment, and transactional branching are applied on this branching and the corresponding programs \(S_{CC}, S_{CA},\) and \(S_{TB}\) are given in appendix A.2.

**Experime**nt. Using our analysis, we conduct an experiment with four different inputs which vary mainly in the number of elements matching the branching condition. Each input consists of the arrays `noDB` and `shareDB` of length \(10^4\). For \(i \in \{0, 1, 2, 3\}\), `noDB` of input \(i\) contains \(i \times 10\) occurrences of `SHARE`. All remaining values as well as the ordering inside each input were randomly chosen. Each input occurs 25 times in the input set, which is hence of size 100 and
leads to an upper bound for the analysis result’s variance of \( \frac{|X|^2}{n} \approx 0.08 \cdot H(X) \). Since there are four different inputs, the maximal possible mutual information equals \( H(X) = 2 \).

The experiment results for \( w = 10 \) are given in figure 4.3 and those for \( w = 0 \) and \( w = 5 \) can be found in appendix A.2 on pages 63 and following. As intended, \( S \) is a strongly \( \gamma \)-leaking program. Surprisingly, none of the transformations closes this timing channel; \( S^{CC} \) and \( S^{CA} \) are \( \gamma \)-leaking programs and \( S^{TB} \) is even a strongly \( \gamma \)-leaking program which contradicts hypothesis 4.2. As our goal is to measure the effect of the transformed conditional branching and \( P^{CA} \) does not contain this branch, the result deserves further investigation. Our first conjecture is that the JIT compiler reintroduced the branch eliminated by conditional assignment. Investigating the assembly code produced by the JIT compiler reveals that this is not the case and that the method contained no branches depending on the values of \texttt{noDB} as intended.

Further investigating this issue, we re-conduct the experiment with a changed input set. It contains again four distinct inputs, however all elements of \texttt{noDB} are set to zero. Hence, the branch condition is never satisfied (as \texttt{SHARE} \( \neq 0 \)) and we expect the program to run in constant time, up to noise. The result for the changed input set however showed that \( S, S^{CC}, S^{CA} \), and \( S^{TB} \) again are \( \gamma \)-leaking programs. We conclude that the running time does not solely depend
Figure 4.4: Mutual information \( I(10, r) \) in bits with 10 warm-up passes for ShareCount with a single input array untransformed (\( S \)), transformed with cross-copying (\( S^{CC} \)), conditional assignment (\( S^{CA} \)), and transactional branching (\( S^{TB} \)) on the entries of \texttt{noDB}.

One remaining difference between distinct inputs are the memory locations they are read from: the analysis implementation stores distinct inputs in distinct arrays and refers to these arrays for repetitions of an input. Our conjecture is that caching behavior causes different delays for access to each of the four input arrays. In order to determine whether this influences the running time of the program, we subsequently change the measurement procedure such that the program reads the input always from the same memory location. The original and new measurement procedures are depicted in listings A.9 and A.10, respectively.

Figure 4.4 shows the result for this changed measurement procedure. After this change, \( S \) and \( S^{TB} \) are still \( \gamma \)-leaking programs, however not strongly \( \gamma \)-leaking programs. In the case of \( S \), mutual information is considerably lower than before the change. Program \( S^{CC} \) shows a clearly mitigated mutual information compared to \( S \). Finally, \( S^{CA} \) seems to close the timing channel as \( I_{S^{CA}}(w, r) \) is nearly zero for nearly all analyses.

Hypothesis \( \text{[4.2]} \) as well as hypothesis \( \text{[4.1]} \) are approved by the results for a single input array – conditional assignment is the only transformation closing the timing channel. The results for the original measurement procedure show that analysis results can be misleading if the timing behavior is not (only) influenced by the conditional branching since it is the only part being transformed. Nev-
Figure 4.5: Average execution times $\bar{T}_{10}(r)$ in bits with 10 warm-up passes for ShareCount with multiple input arrays untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).

Figure 4.6: Average execution times $\bar{T}_{10}(r)$ in bits with 10 warm-up passes for ShareCount with a single input array untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).
ertheless, the fact that one transformation closes the timing channel completely gives evidence that the experiment result allows a meaningful comparison of transformations, up to the general limitations of the experiment, as described in chapter 3.

Looking at the results for $w = 0$, it is noticeable that, against our expectation, $I_S(0, r)$ decreases with increasing $r$ for $r$ roughly between 25 and 150. We speculate that this is due to the fact that $S$ is compiled multiple times during one analysis. If this is the case, some measurements rely on the first compilation of $S$ while some measurements rely on one or more succeeding compilations of $S$. Typically, later compilations yield a more efficient optimization and therefore the running times may change even for identical inputs. Finally, different running times for the same input affect the partitioning and reduce the mutual information.

The performance analysis results $\bar{T}_S(w, r)$ for $w = 10$ are depicted in figures 4.5 and 4.6 for the experiment with multiple input arrays and a single input array, respectively. We observe that whether all input is read from the same memory location did not influence the overall average execution time. The only non-$\gamma$-leaking program of this case study, $S^{CA}$, proved also to be the slowest program with a considerable higher average execution time than all three other programs which show only a proportional low performance reduction. Interestingly, the average execution time of $S^{TB}$ alternates between values around 22000 ns and 40000 ns even for higher values of $r$. Additionally, the performance gain during the first increasing values of $r$ is proportionally not as significant as in the modular exponentiation case study while the average execution times of the ShareCount programs is considerably lower than the average execution times of the modular exponentiation programs.

### 4.3 Case Study 3: Kruskal’s Algorithm

Our next case study consists of an application of Kruskal’s algorithm [Kru56] to calculate the minimum spanning tree of a graph. The minimum spanning tree problem is investigated since the beginning of the last century and has numerous applications [GH85]. One example is its use in graph databases, which provide a view on the stored data explicitly as a graph [AG08]. Graph database queries and responses usually can also be expressed as a graph; the query constitutes a pattern and the corresponding response, a sub graph of the database, matches the pattern. In this context, the minimum spanning tree problem can be used to compress the transitive closure of a graph either to reduce the size of a query or to create an index for a database [CGK05, TL07, LLT09, JHW+10, BMS11, Bog12].

**Example.** Let us consider a database query sent by a social networking service. The service uses a graph database and is divided into a web client sending queries and a server responding to these queries. When the user searches a sushi restaurant, the service looks up restaurants recommended by a friend of the user and being located either in the user’s residence or the city where the user works. In this manner, there are two possibilities for a query sent by the client: either it is looking up a restaurant only in one city or in two cities. Exemplary graphs for these queries are depicted in figure 4.7.
We assume that the information whether the user works and lives in the same city should be processed confidentially. Thus, an eavesdropper must not observe, directly or through a side channel, a difference between the two possible queries. Similar attacks revealing user’s (encrypted) requests to web search engines and their consequences for privacy have been reported in the literature [CWWZ10, JKPT07]. Let us further assume that Kruskal’s algorithm is used to compute the minimum spanning tree of the query. If it leaks the number of vertices contained in the query through its timing behavior, this would reveal the difference between the two queries to an adversary. We will investigate under which conditions such leakage occurs and how it can be mitigated.

**Implementation.** We implement Kruskal’s algorithm with primitive types. Graphs are represented as adjacency lists implemented as int[] arrays. The implementation is depicted in listing 4.4. Function `runKruskal` consumes a graph and stores its minimum spanning tree in the public field `mst`. The formal parameter `graph` contains as first element the number of vertices contained in the graph which is used to compute the maximum length of the resulting spanning tree `mst`. The array `parent` in conjunction with the recursive function `find` serves as a disjoint-set data structure and is used to record the connectivity of vertices currently provided by `mst`. Implicitly, the graph represented by `mst` is divided into disjoint sets where two vertices belong to the same set if and only if an edge in `mst` connects them. Array `parent` maps each vertex to another vertex (parent) which is either a representative of its set or itself mapped to another vertex contained in the same subgraph. Applying `find` to a vertex `v` returns the representative of the set containing `v`.

The `for` loop in line 14 traverses the given adjacency list. For each edge, the representative vertices for its source and target are retrieved using function `find`. Only if source and target belong to different sets, the current edge is added to `mst` and the two sets are joined (lines 18-20). Finally, the number of vertices contained in the minimum spanning tree is computed and the resulting graph is stored in the public field `mst` (lines 23 and 24).

We refrain from using path compression and union by rank in our implementation of the disjoint-set data structure since these optimization techniques do not have any effect for the graphs of the input set used during the evaluation.
of this case study. For simplicity, we additionally expect the input graph either
to be unweighted or the adjacency list to be sorted according to edge weights.

In this experiment, we analyze the timing behavior of Kruskal’s algorithm
when applied to the graphs from the example above. It will be investigated how
much information about whether the first or second graph is being processed
by the algorithm is leaked through its timing behavior.

Potential side channels. Using a static information flow analysis, one could
specify the array graph holding the input graph as confidential to detect a
potential information leak about the fact whether the actual input is the first
or second graph. This analysis would report branchings on secrets as potential
side channels at the following statements:

(a) lines 10 and 14: Loop conditions depending on a field of graph
(b) line 2: Branching condition depending on the value of an element of graph
(c) line 17: Branching condition depending on the return value of find

All mitigation techniques considered here are not applicable to loops with
high guards. Furthermore, cross-copying and conditional assignment are not
applicable to conditional branches containing recursive method calls. Hence,
the only transformable statement is [c] in line 17. In order to compare the

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

public void kruskal(int[] graph) {
    int nv = graph[0];
    int[] mst = new int[nv * 2 - 1];
    parent = new int[graph.length];
    for (int i = 0; i < parent.length; ++i)
        parent[i] = i;
    int idx = 0;
    for (int i = 1; i < graph.length; i += 2) {
        int source = find(graph[i]);
        int target = find(graph[i + 1]);
        if (source != target) {
            mst[++idx] = source;
            mst[++idx] = target;
            parent[source] = target;
        }
    }
    mst[0] = idx / 2 + 1;
    this.mst = mst;
}
```
effects of applying each program transformation, it is favorable to isolate the
timing behavior of \(\text{(c)}\). We accomplish this by restricting the input set used
during the evaluation of this case study. The input set consists of the two graphs
from the example, modified such that both graphs differ only in the number of
vertices they contain. This can be achieved by augmenting the smaller graph
with additional edges. We obtain graphs \(G_1\) and \(G_2\) which are represented by
arrays \(G_1\) and \(G_2\) defined by:

\[
G_1 = \text{new int}[\{5, 0, 1, 1, 2, 1, 2, 2, 3, 2, 3, 2, 4, 2, 4\}]; \\
G_2 = \text{new int}[\{7, 0, 1, 1, 2, 2, 3, 2, 4, 1, 5, 5, 4, 5, 6\}];
\]

Using these definitions, \(G_1\) and \(G_2\) are of equal length, hence the loops \([a]\)
in lines 10 and 14 are executed with the same frequency independent of the
input graph. Therefore, they are not considered as potential timing channels.
Additionally, with these definitions the call frequency of method \(\text{find}\) and the
frequency the condition of \([b]\) is satisfied is equal for both \(G_1\) and \(G_2\).
Thus, branching \([b]\) is not considered as a potential timing channel as well. The
only remaining expected timing channel is statement \([c]\) as desired in order to
compare program transformations applied to it.

If the timing behavior of the whole algorithm reflected the frequency this
branch \([c]\) is taken it would leak the number of edges contained in the minimum
spanning tree, which directly corresponds to the number of vertices in the input
graph. Hence, an observer would be able to distinguish between the two input
graphs. Accordingly, the channel input for the quantitative information flow
analysis is given by \(G_1\) and \(G_2\), while the channel output is denoted by the span
between different running times of the algorithm. The entropy of the channel
input is thus \(H(X) = 1\) and since \(I(X;Y) \leq H(X) \leq 1\), we can expect a
maximum mutual information of 1 bit, where \(I(X;Y) = 1\) corresponds to the
case where an observer could correctly distinguish \(G_1\) and \(G_2\).

**Experiment.** During all runs of this experiment, 500 samples were used, each
of which mapping an input to an output. As these samples do not contain more
than two distinct inputs, each input occurs 250 times and an upper bound for
the analysis result’s variance is given by \(\frac{|X|^2}{n} \approx 0.01 \cdot H(X)\). The first series
of experiments is conducted for the untransformed program \(K\) and the pro-
grams resulting from applying cross-copying \((K^{CC})\) and conditional assignment
\((K^{CA})\). Transactional branching is not applicable on the original program due
to insufficient language support. Although it supports null pointer exceptions,
it has no support for out-of-bounds exceptions as they would occur if applied
to the original program: access to the array \(\text{mst}\) is only within bounds as long
as edges of the minimum spanning tree are missing in \(\text{mst}\). Yet, after applying
transactional branching, \(\text{mst}\) is also accessed after all edges of the minimum
spanning tree are already stored in \(\text{mst}\) since such an array access is inserted
into the previously empty \textbf{else} branch of if statement \([c]\) in line 17. Note that
in general, transactional branching is termination-insensitive even if its original
language is used. However, here termination-insensitivity is introduced by the absence of support for array accesses. Later in this experiment, we will apply transactional branching on a modified implementation of Kruskal’s algorithm.

Figure 4.8 depicts the results for \( w = 10, r = 1, \ldots, 500 \). Program \( K \) is a strongly \( \gamma \)-leaking program as \( I_K(10, r) \) increases with ascending \( r \) and reaches one bit for most values of \( r \) greater than 50. Furthermore, for values of \( r \) high enough, \( K_{CC} \) leaks as much information as \( K \), while for lower values of \( r \), \( I_{K_{CC}}(10, r) \) is slightly reduced compared to \( I_K(10, r) \). Program \( K_{CA} \) shows an even higher information flow than \( K_{CC} \). As with the first results for the ShareCount case study, none of the transformations closes the timing channel. Therefore, we attempt to find the source of timing variances.

A first starting point is to withdraw our assumption that the conditional branching in line 2 does not cause a time difference. The number of times \texttt{find} is executed stays constant for both inputs, however the pattern of these calls differs. In order to compare the calling behavior of \texttt{find} for both inputs, we record the call depth for each loop iteration. It becomes apparent that the calling frequency differs during loop iterations 3, 4, 5, and 7 between the first and second input:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call freq for ( G_1 )</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>Call freq for ( G_2 )</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>19</td>
</tr>
</tbody>
</table>

We speculate that this difference causes a higher branch misprediction rate for one of the inputs and finally influences the program’s timing behavior. Accordingly, we re-conduct the experiment for a further series of program versions
where this difference is avoided. In version flat, cf. listing A.14, the recursive method \texttt{find} is substituted by a loop traversing the array \texttt{parent} in order to update connectivity of vertices. The obtained program is free of recursion and therefore can be completely transformed by cross-copying and conditional assignment. The experiment is re-conducted for version flat, showing that \( K_{CA}^{flat} \) is again a \( \gamma \)-leaking program, even though mutual information is slightly reduced compared to \( K_{CA} \).

Yet another investigation yields the instantiation of array \texttt{mst} as the source of the remaining execution time difference. The length of \texttt{mst} depends on the number of vertices and therefore differs between the first and second input. In Java, \texttt{int} arrays are initialized with the value zero for all elements. Thus, for \( G_1 \), an array of length 9 needs to be initialized while for \( G_2 \), an array of length 13 needs to be initialized. Indeed, the bytecode instruction to create an array, \texttt{newarray}, can appropriately be modeled in the functional time model [HS05] because its running time functionally depends on the length of the array.

We expect this to be the reason for different execution times. In order to eliminate this difference, we take the size of the input graph as an upper bound for the size of the minimum spanning tree and accordingly instantiate array \texttt{mst} with the length of the input array which does not differ between both inputs. In this way, elements at the end of the array might be unused and we introduce another initialization with the value \(-1\) for each element to allow a distinction between used and unused elements. Additionally, creating a longer array gives raise to applying transactional branching.

Finally, the experiment is re-conducted with both modifications (removing recursion and instantiation with constant size). The modified programs are denoted by \texttt{flat, fixed}, i.e., \( K_{flat, fixed}^{CC}, K_{flat, fixed}^{CA}, \) and \( K_{flat, fixed}^{TB} \). The results, given in figure 4.9 show \( K_{flat, fixed}^{flat} \) to be a strongly \( \gamma \)-leaking and \( K_{flat, fixed}^{CA} \) to be a non-\( \gamma \)-leaking program. We take this as evidence that the experiment allows to assess the time difference caused by the conditional branching in line 17. Programs \( K_{flat, fixed}^{CC} \) and \( K_{flat, fixed}^{TB} \) are strongly \( \gamma \)-leaking programs as well, approving hypotheses 4.2 and 4.1. Interestingly, \( I_{K_{flat, fixed}^{CA}} (10,r) \) shows three considerably high peaks but is otherwise nearly 0. We speculate that an otherwise non-effective timing channel, e.g., induced by a cache, caused a high analysis result in these rare cases. All remaining experiment results are provided in appendix A.3 on pages 74 and following.

In order to investigate whether the calling behavior of the recursive version of Kruskal’s algorithm for itself induces an observable timing channel, we re-conduct the experiment for version \texttt{rec, fixed} which is recursive but has an instantiation of array \texttt{mst} of fixed length. The experiment results show all programs of version \texttt{rec, fixed}, \( K_{rec, fixed}^{flat} \), \( K_{rec, fixed}^{CC} \), \( K_{rec, fixed}^{CA} \), and \( K_{rec, fixed}^{TB} \) to be \( \gamma \)-leaking, hence the only non-\( \gamma \)-leaking program investigated in this case study is \( K_{CA}^{flat} \). We conclude that an observable timing channel is indeed induced by the fact that the distributions of recursive method calls between loop iterations differs between both inputs (although the total number of recursive calls is constant).

Analyzing the performance results, depicted in figures 4.11 and 4.12 for versions \texttt{rec, var} and \texttt{flat, fixed}, respectively, shows the average execution times for version \texttt{rec, var} lying relatively close to each other compared to version \texttt{rec, var}. Program \( K_{CA} \) is slightly faster than \( K_{CC} \) but \( K_{CA}^{flat, fixed} \) is considerably slower.
Figure 4.9: Mutual information $I(10, r)$ in bits with 10 warm-up passes for Kruskal’s algorithm without recursion (flat) and fixed array length (fixed) untransformed ($K_{flat, fixed}$), transformed with cross-copying ($K_{CC, fixed}$), conditional assignment ($K_{CA, fixed}$), and transactional branching ($K_{TB, fixed}$).

Figure 4.10: Average execution times $\bar{T}(10, r)$ in bits with 10 warm-up passes for Kruskal’s algorithm with recursion and varying array length untransformed ($K$), transformed with cross-copying ($K_{CC}$), conditional assignment ($K_{CA}$), and transactional branching ($K_{TB}$).

Figure 4.11: Performance results for Kruskal’s algorithm with 10 warm-up passes.
Figure 4.12: Average execution times $\bar{T}(10, r)$ in bits with 10 warm-up passes for Kruskal’s algorithm without recursion (flat) and fixed array length (fixed) untransformed ($K_{flat, fixed}$), transformed with cross-copying ($K_{CC, flat, fixed}$), conditional assignment ($K_{CA, flat, fixed}$), and transactional branching ($K_{TB, flat, fixed}$) than $K_{CC, flat, fixed}$. Transactional branching causes an even higher performance reduction although not considerably mitigating the information flow.

4.4 Case Study 4: IDEA

In addition to the first three case studies, we apply our analysis to the IDEA block cipher. As described in section 2.1, a branching dependent on the secret key during a modular multiplication constitutes a known timing channel and an attack revealing 16 key bits has been performed on the Java implementation of IDEA contained in FlexiProvider. During the author’s previous work, the effects of cross-copying, conditional assignment, and transactional branching on this attack were analyzed showing that among these transformations, only conditional assignment was able to prevent a successful attack. It was suspected that branch mispredictions caused a considerable timing difference between two branches depending on the secret key and thereby caused a successful attack, cf. section 2.2. In this case study, we will re-investigate the FlexiProvider implementation of IDEA by means of our analysis framework.

The branching constituting the side channel is contained in the method mulMod16, which we provide again in listing 4.5. The formal parameter a constitutes a part of the ciphertext or plaintext, respectively, while formal parameter b constitutes a part of the secret key. Method mulMod16 calculates the product of a and b modulo $2^{16} + 1$ and is called multiple times during encryption and decryption. In case either of the operands has the value $2^{16}$, which is represented as the int value 0, the multiplication is reduced to a subtraction (lines 8 and 10). Otherwise, a and b are multiplied and the corresponding representative is calculated (lines 12 et seqq.).

Experiment. The mentioned attack assumes that each key part is nonzero, i.e., it is assumed that the second branch (line 10) is unreachable. Hence we construct two different inputs where the first branch (line 8) or third branch (line 12) will be taken, respectively. For the first input, a equals zero and b
Listing 4.5: Modular multiplication in IDEA as implemented in FlexiProvider

```java
private static final int mulModulus = 0x10001;
private static final int mulMask = 0xffff;

public 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;
}
```

takes a random non-zero value. The second consists of the same key part, i.e., b takes the same value as for the first input, while a equals a random non-zero value. The two distinct inputs are repeated 40 times leading to a total sample size of \( n = 80 \), and an upper bound for the analysis result’s variance of \( \frac{|X|^2}{n} \approx 0.05 \cdot H(X) \).

As a first step, we aim at reproducing the confirmation of a timing channel from the original IDEA experiment in our prior work. We apply our analysis to the untransformed \( \text{mulMod16} \) method which we denote by \( M \). However, the results show \( M \) not to be a \( \gamma \)-leaking program, i.e., do not confirm the presence of a timing channel. A main difference between the experiment presented here and the original experiment is the context from which \( \text{mulMod16} \) is called. While in the original experiment, a complete encryption is performed for each sample during which \( \text{mulMod16} \) is called 34 times, in the experiment on \( M \), \( \text{mulMod16} \) is called directly and only once for each sample. In the first case, for an encryption of an arbitrary block of 8 bytes, the first branch of \( \text{mulMod16} \) is executed either zero or one times out of 34 calls of \( \text{mulMod16} \). In the latter case, as the two distinct inputs are distributed evenly in our input set, the first and third branch are executed with the same frequency. As a result of our previous work, we conjectured that the timing channel in the original experiment was caused by a low execution frequency of the first branch in comparison with the execution frequency of the third branch, and a resulting branch misprediction if the first branch is taken. Assuming that the timing channel is indeed caused by branch mispredictions, the fact that the execution frequency of both branches matches explains that no timing channel is measurable in the experiment on \( M \).

In order to adapt how \( \text{mulMod16} \) is called to a situation closer to the calling behavior of an encryption in IDEA, we change the program under analysis so that it includes the context from which \( \text{mulMod16} \) is called in FlexiProvider. The method \( \text{encryptDecrypt} \) encrypts or, respectively, decrypts 8 bytes and calls
Listing 4.6: Encryption and decryption in IDEA: excerpt of method encryptDecrypt in FlexiProvider

```java
private static final int rounds = 8;

public void encryptDecrypt(int[] key, byte[] in, int in_offset, byte[] out, int out_offset) {
    int k = 0;
    int t0, t1;
    int x0 = in[in_offset++] << 8;
    x0 |= in[in_offset++] & 0xff;
    [...]
    for (int i = 0; i < rounds; ++i) {
        x0 = mulMod16(x0, key[k++]);
        x1 += key[k++];
        x2 += key[k++];
        x3 = mulMod16(x3, key[k++]);
        t0 = x2;
        x2 = mulMod16(x0 ^ x2, key[k++]);
        t1 = x1;
        x1 = mulMod16((x1 ^ x3) + x2, key[k++]);
        x2 += x1;
        x0 ^= x1;
        x3 ^= x2;
        x1 ^= t0;
        x2 ^= t1;
    }
    x0 = mulMod16(x0, key[k++]);
    t0 = x1;
    x1 = x2 + key[k++];
    x2 = t0 + key[k++];
    x3 = mulMod16(x3, key[k]);
    out[out_offset++] = (byte) (x0 >>> 8);
    out[out_offset++] = (byte) x0;
    [...]
}
```

mulMod16 34 times. By $M_{ed}$, we denote the program for a new experiment where encryptDecrypt is called once per sample. It is depicted in listing [4.6]. In lines 9 to 28, the eight rounds and output transformation of IDEA are performed. Conversion of the byte array $in$ holding the plaintext or ciphertext, respectively, to integer variables and vice versa is performed in lines 6 et seqq. and 29 et seqq. Similar to the previously used input set, we use in both distinct inputs the same random non-zero values for parameter $key$. Parameter $in$ likewise holds random non-zero values with the exception that in the first input, $in$ at indices 0 and 1 holds zero. Hence, the first branch of mulMod16 is executed with high probability exactly once for the first input and with high probability never executed for the second input. The remaining parameters $in\_offset$, $out\_offset$, and the elements of $out$ are defined as zero.

The results for the adapted experiment on program $M_{ed}$ with 10 warm-up passes are depicted in figure [4.13]. Remaining results can be found in appendix A.4 on pages 84 and following. Program $M_{ed}$ without transformation
Figure 4.13: Mutual information $I_{10}(r)$ in bits with 10 warm-up passes for mulMod16 in context of encryptDecrypt untransformed ($M_{ed}$), transformed with cross-copying ($M_{CC}^{ed}$), conditional assignment ($M_{CA}^{ed}$), and transactional branching ($M_{TB}^{ed}$).

Figure 4.14: Average execution times $\bar{T}_{10}(r)$ in bits with 10 warm-up passes for mulMod16 in context of encryptDecrypt untransformed ($M_{ed}$), transformed with cross-copying ($M_{CC}^{ed}$), conditional assignment ($M_{CA}^{ed}$), and transactional branching ($M_{TB}^{ed}$).
Program | $\gamma$-leaking | strongly $\gamma$-leaking
--- | --- | ---
$E$ | ✓ | ✓
$E^{CC}$ | – | –
$E^{CA}$ | – | –
$E^{TB}$ | ✓ | –
$S$ | ✓ | –
$S^{CC}$ | ✓ | –
$S^{CA}$ | – | –
$S^{TB}$ | ✓ | –
$K^{flat, fixed}_{CC}$ | ✓ | ✓
$K^{flat, fixed}_{CA}$ | ✓ | ✓
$K^{flat, fixed}_{TB}$ | – | –
$M^{ed}$ | ✓ | ✓
$M^{CC}$ | ✓ | ✓
$M^{CA}$ | – | –
$M^{TB}$ | ✓ | ✓

Table 4.1: Summary of experiment results for modular exponentiation ($E$), ShareCount with single input array ($S$), Kruskal’s algorithm without recursion and with fixed array length ($K^{flat, fixed}$), and IDEA in the context of method `encryptDecrypt` ($M^{ed}$).

is strongly $\gamma$-leaking, while $M^{CC}_{ed}$ and $M^{TB}_{ed}$ are $\gamma$-leaking but not strongly $\gamma$-leaking. Program $M^{CA}_{ed}$ is not $\gamma$-leaking. These results are consistent with the results from the previous case studies and show conditional assignment to be the only transformation considerably mitigating information flow.

Figure 4.14 shows the performance analysis results for `mulMod16` in context of `encryptDecrypt` and $w = 10$. As in case studies 1 and 3, for values of $r$ higher than approximately 15, the average execution times stabilize. They are clearly separated and show conditional assignment to induce the highest performance tradeoff.

### 4.5 Summary

Table 4.1 gives an overview of the obtained results. We observed that conditional assignment closes the timing channel in all case studies, i.e., hypothesis 4.2 is approved. Both cross-copying and transactional branching preserve the timing channel with the exception of the modular exponentiation case study, where only transactional branching has no desirable effect on the timing channel. Thus, hypothesis 4.1 is approved in all but one cases. In addition to table 4.1, appendix A.5 contains histograms illustrating the results of all programs analyzed during our case studies with 10 warm-up passes. These histograms depict the frequency of occurrences of mutual information results below the threshold for $\gamma$-leaking programs, $\epsilon H(X)$, above this threshold but below $H(X)$, and equal to
Following definition 4.1, a program is $\gamma$-leaking if and only if its leftmost bar in the histogram denotes a number smaller than or equal to $(1 - \gamma)\rho = 375$.

For the given arrangement of experiment and environment, we have shown that it is possible to close a timing channel at the source code level. In extension to what has been shown by Molnar et al. [MPSW05] using the program counter security model, this is true even in presence of a virtual machine with JIT compilation. The notion of PC-security seems to be a reasonable choice, as

(a) $P_{CC}$ and $P_{TB}$ are $\gamma$-leaking and not PC-secure for all $P \in \{E, S, K, M\}$.

(b) $K_{rec, var}$ and $K_{rec, fixed}$ are $\gamma$-leaking (for all transformations) and not PC-secure.

(c) $P_{CA}$ is not $\gamma$-leaking and PC-secure (with special adaption to Java) for all $P \in \{E, S, K, M\}$.

However, it is not straightforward to determine at the source code level whether a given Java program is PC-secure. It is not straightforward in the sense that removing branchings at source code level may not be sufficient as the example of $K_{flat, var}$ shows: there, the bytecode instruction `newarray` does not run in constant time which is however not reflected by a conditional branch at source code level.

Furthermore, PC-security is certainly not sufficient to protect against timing channels which are not caused by conditional branches as shown by the ShareCount case study with multiple input arrays. There, even the PC-secure program $S_{CA}$ is leaking.

Evaluating the quantitative character of our results enables to assess the security of a program with respect to specific conditions. It might be sufficient for a certain use case to mitigate a timing channel rather than completely closing it, e.g., in the ShareCount case study (with a single input array) the usage of cross-copying might be acceptable as it reduces mutual information. Accordingly, it might even considered to be sufficiently secure to leave a $\gamma$-leaking program as it is, if it is executed with a low frequency, as, e.g., $K$ does not $\gamma$-leak for $r \leq 50$. 

42
Chapter 5

Conclusion

The previous chapters introduced our approach of an empirical timing channel analysis and presented four case studies evaluating three language-based techniques for mitigating timing channels. In this chapter, we will summarize these investigations. Furthermore, we will discuss our effort to implement a program transformation inside a Java JIT compiler in the scope of objective A from the thesis proposal, cf. appendix B. Finally, we will recapitulate our most interesting findings and conclude with a justification of design decisions.

5.1 Summary of Experiment Results

In each case study, programs transformed with cross-copying, conditional assignment, and transactional branching were evaluated with respect to our notion of information leakage through timing observations, the notion of \(\gamma\)-leaking programs (cf. definition 4.1). Considering a non-\(\gamma\)-leaking program with \(\gamma = \frac{1}{4}\) as timing channel free, conditional assignment closed the timing channel in all four case studies. In contrast, cross-copying, and transactional branching did not remove any timing channel completely, with the exception of cross-copying closing the timing channel in the modular exponentiation case study. In some cases, the latter two transformations mitigate information flow via timing channels, while in other cases, e.g., the ShareCount case study, transactional branching even increases the amount of information leaked.

In addition to a quantitative assessment of information leakage in our case studies, we also implemented a simple performance analysis of the considered program transformations, yielding that all three transformations increase the average execution time of a program. In most cases, the lowest performance trade-off with respect to the average execution time is induced by cross-copying, while in some cases, conditional assignment, in some cases transactional branching show the slowest performance.

As elaborated during section 4.5, no more than those programs proved to be free of a timing channel, according to our definition of \(\gamma\)-leaking programs, which were free of control flow branchings depending on confidential data. This matched our expectation, formulated as a conclusion about the results from the author’s previous work, and is also consistent with the results of Coppens et al. \cite{CVBS09}.
Interestingly, the complete removal of control flow branchings through conditional assignment in our experiments is sensitive to its implementation. In particular, the \texttt{mask} function, generating a bitmask according to the value of a boolean expression has to be implemented such that no branchings occur, cf. section 2.3.2. Additionally, since we investigated these transformations on the source code level, compiling the source code to Java bytecode and compiling the bytecode to machine code (at runtime) must not introduce control flow branchings depending on the secret. One possible solution addressing these requirements about the compilation process is an implementation of a program transformation inside the JIT compiler of the Java virtual machine (JVM). Such an implementation can facilitate giving guarantees about the absence of control flow branchings that depend on confidential data since the program is not compiled any further after it has been transformed. As machine code instructions are available to a program transformation inside the JIT compiler, arbitrary conditions for an assignment may be used, in contrast to the limitation to comparisons of integers we discussed in section 2.3.2. The resulting program presumably also has a better timing performance: for example, the machine instruction \texttt{CMOV} on x86 platforms performs or suppresses an assignment depending on a register comparison, hence avoiding multiple instructions which are used if implementing conditional assignment on source code level. In some cases, branch predication, i.e., the replacement of conditional branchings through conditional instructions such as \texttt{CMOV}, may even decrease the execution time, as investigated by \cite{LG10}.

In the scope of objective A of the thesis proposal, we implemented a program transformation inside a JIT compiler, which eliminates a given confidential branching while preserving the effect on the memory by use of the \texttt{CMOV} instruction. However, the transformation in its current implementation does not apply to arbitrary programs, as it emerged that such a JIT compiler modification was too extensive to be realized within this work. The following section presents the concept of the aimed transformation as well as the complications arising when considering arbitrary programs.

5.2 Modifying a JVM JIT Compiler

In order to investigate the advantages and disadvantages of a timing channel mitigation inside a JIT compiler, we modified the JIT compiler of the HotSpot Server JVM \cite{Hot13}. This compiler is called C2. The HotSpot virtual machine was chosen because it seems to amplify timing channels due to its highly optimizing compiler: conducting the IDEA timing experiment described in section 2.1 on different JVMs showed that the timing channel was considerably more significant on HotSpot than on other JVMs like, e.g., Maxine \cite{Max13}. We intended to design a transformation applying \textit{branch predication} on control flow branchings indicated by Java annotations. Branch predication avoids conditional jumps by choosing conditional machine instructions which are only executed if a certain condition is met. The only conditional instruction provided by the x86 architecture is \texttt{CMOV} which we used in the implementation of the transformation.

A similar program transformation was implemented by Coppens et al. \cite{CVBS09}. However, they use the LLVM \cite{LLV13} compiler system in order
to implement a static program transformation for C. In contrast, C2 applies to
programs run in a JVM which induces a more complicated control flow struc-
ture due to exceptional behavior caught by the virtual machine. For example,
an out-of-bound array access produces a Java exception, hence, a control flow
branching before the array access is often required. Additionally, C2 compiles
at runtime and uses profiling results collected by the virtual machine’s inter-
preter in order to optimize and adapt the produced native code to the dynamic
behavior of the program.

When C2 is invoked by the JVM in order to compile a method, it first
parses the method’s byte code and constructs a program dependence graph of
the method. This high-level intermediate representation is called Ideal. The
original design of Ideal is described in [CP95], while the current implementa-
tion in C2 evolved and now differs to some extent.

After generating the graph representation, C2 applies high-level optimiza-
tions in multiple passes. Subsequently, the graph is iteratively transformed
to a low-level but still machine independent representation. Finally, the code
generation phase emits machine instructions using a bottom-up rewrite sys-

Figure 5.1 shows an extract of an exemplary Ideal graph. On the left
hand side, a control flow branching is depicted. Control flow dependencies and
data dependencies are represented by solid and dashed arrows, respectively. The
Phi node provides either Value1 or Value2, depending on whether the Region
node is reached through IfTrue or IfFalse. The value provided by the Phi
node may be used, e.g., in an assignment to a variable. The right hand side of the
figure depicts the output of our transformation. The control flow branching
is replaced by a CMoveI node providing the correct value depending on the
condition represented by the Bool node. At code generation, a CMOV assembler
instruction is generated for the CMoveI node. Control flow branchings of this
form are successfully handled by our current implementation of the program
transformation.

However, other forms of control flow branchings may occur as well due to
various factors such as loop optimizations or method inlining. Figure 5.2 shows
an extract of an Ideal graph which is not handled correctly by our current imple-
mentation. Here, the incoming control flow edges to the central Region node do
Figure 5.1: On the left hand side: An extract of an Ideal graph. On the right side: The left hand graph with the control flow branching replaced by a conditional move node. Solid and dashed arrows denote control flow and data dependence edges, respectively.

Figure 5.2: An extract of an Ideal graph. Two branches, depending on different conditions, are merged at the same Region node.
not origin from the same If node, as in the previous example. Conversely, the values represented by both Phi nodes associated with the region node depend on different If node conditions. Therefore, it is not possible to directly replace the Phi nodes with CMove nodes. In addition, no information about which of the Phi nodes corresponds to the If node we intend to eliminate is stored in the Ideal graph.

After experiencing these difficulties, it seems to be necessary to modify the compilation process at multiple phases in order to implement the program transformation. First, at parse time, information about which Phi nodes have to be replaced by CMove nodes has to be recorded. Second, generating a complicated dependency graph like in the latter example has to be avoided until the transformation is applied. This is not straightforward, as Region nodes with multiple If nodes may be generated already during parsing, while inserting CMove nodes is only available during the optimization phases.

As a result of these unforeseen difficulties, we were not able to implement the program transformation for arbitrary programs within the time constraint of this thesis. Especially, the absence of a comprehensive documentation or specification indicating the possible forms of control flow in an Ideal graph complicates and delays an implementation of the desired program transformation. Hence, the set of programs supported by our current implementation of the program transformation is limited; no more than the first of our four case studies, modular exponentiation, is supported by our program transformation. Thus, we refrained from including this transformation in the evaluation of our case studies in the interest of giving more weight to our findings summarized in the following section.

5.3 Summary of Contributions

We provide a model of timing channels suitable for an interpretation of an empirical analysis result as an information theoretic channel. This model includes a treatment of time data as a discrete channel output. We implemented the corresponding analysis tool capable of generating program input and recording timing behavior of a given Java program. This tool partitions time measurements, interpreting them as discrete channel outputs. It reports the time performance of the program and, by the help of Anonymity Engine, calculates the channel capacity and mutual information. As achieved in our case studies, our tool can be used in order to conduct experiments evaluating the effects of different mitigation techniques on a program containing a timing channel. Using this evaluation, both information leakage and time performance loss of different mitigation techniques may be compared.

We applied our analysis framework in four case studies, evaluating for each case study the effects of cross-copying, conditional assignment, and transactional branching. Each case study contains a non-trivial Java program with an exemplary setting of security concerns. Additionally, case study 3, comprising Kruskal’s algorithm, provides an extended use case in the field of graph databases and social networking services. During each case study, the three given program transformations are quantitatively analyzed with respect to information leakage and time performance loss. The experimental results derived through these case studies are interpreted with a uniform classification and con-
firm our hypotheses developed on the basis of the author’s previous work. This connection to the analysis of a timing channel in IDEA is further supported by an additional case study not required by the thesis proposal. In particular, we showed the presence of timing channels in programs transformed with cross-copying and transactional branching, which therefore prove to be unsuccessful in avoiding the vulnerability against timing attacks.

Furthermore, we reported several observations made during the design of our case studies. As described in section 4.3, we experimentally show at the example of a disjoint-set data structure that two program executions can be distinguished by alternating call frequencies of the recursive method in different loop iterations, even if the total call frequencies are equal. To the best of our knowledge, the effect of recursion on timing channels has not been mentioned in the literature before. Additionally, we confirm that the Java bytecode instruction `newarray` may cause a timing channel, cf. again section 4.3, which is also mentioned in [HS05].

Finally, during the development of case study ShareCount (see section 4.2), we noticed a timing channel depending on which of four `Object` arrays was accessed in order to read the input for the program under analysis. This timing channel even occurs in programs without input-dependent control flow branchings. Considering this kind of timing channels is important for the design of experimental timing analyses, as it may unintentionally distort the analysis result.

5.4 Design Decisions

Finally, in this section we briefly discuss design decisions in cases where they are not already mentioned before.

**Primitive Java Programs.** All programs investigated in our four case studies use primitive, i.e., non-object, types, with the exception of arrays (which are represented as objects in Java). This choice was made after observing timing behavior of programs with object types which we could not explain. For example, the analysis of a program iterating over an object type array revealed diverging execution times for different inputs of the same length although the program did not contain any branchings on the input content. It remains to investigate whether this observation is induced by the unintentional influence of the measurement procedure on the analysis result when multiple input arrays are used, cf. the modified measurement procedure described in section 4.2.

**Design of Experiments.** Chatzikokolakis et al. present two statistical hypothesis tests, a test for estimated bounds on the true value of channel capacity and a test for zero information leakage [CCG10]. Our analysis framework uses the first of these tests, giving bounds on the capacity of a timing channel. We did not design our framework for an explicit application of the latter test for zero information leakage since it requires a number of repetitions of the analysis extending the duration of the already slow analysis considerably. (The experiments comprising all four program versions of Kruskal’s algorithm discussed in section 4.3 took approximately 48 hours, on an AMD E-300 APU with 1.3GHz and 4GB main memory.) However, the test for bounds on the channel capacity

48
is only performed for single analyses and not for the whole experiments as used
in our case studies. These experiments are based on multiple single analyses
and are designed to use a large number of different values for the measurement
parameters $r$, $w$, and $c$ described in section 3.4 accepting the drawback of not
implementing a statistical hypothesis test. In particular, the measurement pa-
rameters for experiments were chosen to allow, with a reasonable experiment
duration, various different values for the number of rounds (parameter $r$) and
multiple values for the number of warm-up passes (parameter $w$). The num-
ber of different values for $w$ was reduced to three as observations during the
design of the experiments showed that the effect of increasing $w$ in steps of
one was not substantial. The number of samples (parameter $s$) was chosen
such that the bounds on channel capacity provided by the single analyses are
reasonably accurate while maintaining a reasonable overall running time of ex-
periments. Specifically, we chose $s$ such that $|X|/n < 0.1 H(X)$. By expressing
the accepted variance as a proportion of the maximum analysis result $H(X)$,
we avoid infeasible large sample sets. In order to further support the accuracy
of the experiment results, a statistical hypothesis test could be designed using
a fixed combination (or a small number of combinations) of $r$, $w$, and $s$, which
has to be found through observing the timing peculiarities of the system under
analysis.

Notion of Leakage. We introduced the property of programs of being $\gamma$-
leaking, instead of using directly a quantile or a confidence interval, since our
experiments used in the case studies are not, without further assumptions, a
realization of a single random variable, since the values for $r$ and $w$ vary during
an experiment for reasons described above.

Partitioning. Designing the channel output as partitions, rather than directly
taking a single measurement parameter as a channel output, overapproximates
the analysis result in cases of low information leakage and a large number of
executions of the program under analysis (e.g., in our experiments a high value of
$w$), cf. chapter 3. In contrast, taking single measurement parameters as channel
output overapproximates information leakage in cases of low values for $w$ and $r$.
As the undesired impact of the latter solution seemed to be considerable more
severe, we decided to partition measurement results.

The partitioning function was realized using a known distribution of chan-
nel inputs as the alternative realization we investigated as well proved to be
highly dependent on a single configuration of parameters $r$, $w$, and $s$. This
alternative realization builds a threshold for a minimum distance between two
time values corresponding to different channel outputs and would either require
manual interaction during an experiment, which was unacceptable, or a sophis-
ticated automatic calculation of an appropriate threshold. Nevertheless, our
implementation of the partitioning function using a known input distribution
could be improved to avoid an overapproximation of information leakage, e.g.,
by specially treating equal time measurements for distinct channel inputs.
Bibliography


Appendix A

Source Code and Experiment Results

A.1 Case Study 1: Modular Exponentiation

Listing A.1: Modular exponentiation ($E$)

```java
public void exp(long m, int d) {
    long r = 1;
    for (int i = 0; i < 32; i++) {
        if (d % 2 == 1)
            r = (r * m) % module;
        m = m * m % module;
        d >>= 1;
    }
    result = (int) (r % module);
}
```

Listing A.2: Modular exponentiation transformed with cross-copying ($E^{CC}$)

```java
public int resultSkip;
public void exp(long m, int d) {
    long r = 1;
    long rSkip = 1;
    for (int i = 0; i < 32; i++) {
        if (d % 2 == 1)
            r = (r * m) % module;
        else
            rSkip = (r * m) % module;
        m = m * m % module;
        d >>= 1;
    }
    resultSkip = (int) (rSkip % module);
    result = (int) (r % module);
}
```

Listing A.3: Modular exponentiation transformed with conditional assignment ($E^{CA}$)

```java
public void exp(long m, int d) {
    long r = 1;
    for (int i = 0; i < 32; i++) {
        long mask = (d << 31) >> 31;
        r = (r & ~mask) | (((r * m) % module) & mask);
        m = m * m % module;
        d >>= 1;
    }
    result = (int) (r % module);
}
```

Listing A.4: Modular exponentiation transformed with transactional branching ($E^{TB}$)

```java
private long r;
private long rSwap;
public void exp(long m, int d) {
    r = 1;
    for (int i = 0; i < 32; i++) {
        if (d % 2 == 1) {
            beginT();
            r = (r * m) % module;
            commitT();
        } else {
            abortT();
        }
    }
    r = (r * m) % module;
    result = (int) (r % module);
}
```
```java
else {
    beginT();
    commitT();
    beginT();
    r = (r * m) % module;
    abortT();
}
m = m * m % module;
d >>= 1;
}
result = (int) (r % module);

private void beginT() {
    rSwap = r;
}
private void commitT() {
}
private void abortT() {
    r = rSwap;
}
```
Figure A.1: Mutual information $I_E(0,r)$ in bits with 0 warm-up passes for modular exponentiation untransformed ($E$), transformed with cross-copying ($E^{CC}$), conditional assignment ($E^{CA}$), and transactional branching ($E^{TB}$).

Figure A.2: Mutual information $I_E(5,r)$ in bits with 5 warm-up passes for modular exponentiation untransformed ($E$), transformed with cross-copying ($E^{CC}$), conditional assignment ($E^{CA}$), and transactional branching ($E^{TB}$).
Figure A.3: Mutual information $I_{(10,r)}$ in bits with 10 warm-up passes for modular exponentiation untransformed ($E$), transformed with cross-copying ($E_{CC}$), conditional assignment ($E_{CA}$), and transactional branching ($E_{TB}$).

Figure A.4: Average execution times $\bar{T}_{(w,r)}$ in nano seconds for modular exponentiation untransformed ($E$), transformed with cross-copying ($E_{CC}$), conditional assignment ($E_{CA}$), and transactional branching ($E_{TB}$).
A.2 Case Study 2: ShareCount

Listing A.5: ShareCount (S)

```java
public void count (int[] noDB, int[] valueDB) {
    long shareValue = 0;
    int i = 0;
    while (i < noDB.length) {
        int no = noDB[i];
        if (no == SHARE) {
            int value = valueDB[i];
            shareValue = shareValue + value;
        }
        i++;
    }
    result = shareValue;
}
```

Listing A.6: ShareCount transformed with cross-copying (S\text{CC})

```java
public class ShareCountPrimitiveCrossCopying {
    public long shareFieldDummy;
    public void count (int[] noDB, int[] valueDB) {
        long shareValue = 0;
        long shareValueDummy = 0;
        int i = 0;
        while (i < noDB.length) {
            int no = noDB[i];
            if (no == Share.MYSHARE) {
                int value = valueDB[i];
                shareValue = shareValue + value;
            } else {
                int value = valueDB[i];
                shareValueDummy = shareValue + value;
            }
            i++;
        }
        shareField = shareValue;
        shareFieldDummy = shareValueDummy;
    }
}
```

Listing A.7: ShareCount transformed with conditional assignment (S\text{CA})

```java
public class ShareCountPrimitiveCondAssign {
    public void count (int[] noDB, int[] valueDB) {
        long shareValue = 0;
        int i = 0;
        while (i < noDB.length) {
            int no = noDB[i];
            int m = ((Share.MYSHARE - no) >> 31) | ((no - Share.MYSHARE) >> 31);
            int value = valueDB[i];
            shareValue = (shareValue & ~m) | ((shareValue + value) & m);
            i++;
        }
        shareField = shareValue;
    }
}
```

Listing A.8: ShareCount transformed with transactional branching (S\text{TB})

```java
public class ShareCountPrimitiveTransBranch {
    private void beginT() {
        shareValueSwap = shareValue;
    }
    private void abortT() {
        shareValue = shareValueSwap;
    }
    private void commitT() {
    }
    private long shareValue = 0;
    private long shareValueSwap = 0;
    public void count (int[] noDB, int[] valueDB) {
        int i = 0;
        while (i < noDB.length) {
            int no = noDB[i];
            if (no == Share.MYSHARE) {
                beginT();
                int value = valueDB[i];
                shareValue = shareValue + value;
                commitT();
                beginT();
            }
        }
    }
}
```
Listing A.9: Measurement procedure

repeat \((w + 1)\) times
  initialize inputs
  for \(j = 1, \ldots, n\)
    for \(i = 1, \ldots, r\)
      \(\text{measures}[j] := \text{time}(\text{program}(\text{inputs}[i]))\)
  partitioning(\text{measures}, |\mathcal{Y}|)
output observations

Listing A.10: Measurement procedure modified for a single input array

repeat \((w + 1)\) times
  initialize inputs
  for \(j = 1, \ldots, n\)
    copy \text{inputs}[i] to \text{singleInput}
      for \(i = 1, \ldots, r\)
        \(\text{measures}[j] := \text{time}(\text{program}(\text{singleInput}))\)
  partitioning(\text{measures}, |\mathcal{Y}|)
output observations
Figure A.5: Mutual information $I_s(0, r)$ in bits with 0 warm-up passes for ShareCount with multiple input arrays untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).

Figure A.6: Mutual information $I_s(5, r)$ in bits with 5 warm-up passes for ShareCount with multiple input arrays untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).
Figure A.7: Mutual information $I_{S}(10, r)$ in bits with 10 warm-up passes for ShareCount with multiple input arrays untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).

Figure A.8: Average execution times $T_{w, r}$ in nano seconds for ShareCount with multiple input arrays untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).
Figure A.9: Mutual information $I_{S}(0, r)$ in bits with 0 warm-up passes for ShareCount with a single input array untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).

Figure A.10: Mutual information $I_{S}(5, r)$ in bits with 5 warm-up passes for ShareCount with a single input array untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).
Figure A.11: Mutual information $I_{S}(10, r)$ in bits with 10 warm-up passes for ShareCount with a single input array untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).

Figure A.12: Average execution times $\bar{T}(w, r)$ in nano seconds for ShareCount with a single input array untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).
A.3 Case Study 3: Kruskal’s Algorithm

Listing A.11: Kruskal’s algorithm recursive with varying array length ($K_{\text{rec, var}}$)

```java
public class KruskalPrimitiveRecVar {
    public int[] mst;
    private int[] parent;
    private int find(int x) {
        if (parent[x] != x)
            return find(parent[x]);
        return x;
    }
    public void runKruskal(int[] graph) {
        int nv = graph[0];
        int[] mst = new int[(nv - 1) * 2 + 1];
        parent = new int[graph.length];
        for (int i = 0; i < parent.length; ++i) {
            parent[i] = i;
        }
        int idx = 0;
        for (int i = 1; i < graph.length; i += 2) {
            int source = find(graph[i]);
            int target = find(graph[i + 1]);
            if (source != target) {
                mst[++idx] = source;
                mst[++idx] = target;
                parent[source] = target;
            }
        }
        mst[0] = idx / 2 + 1;
        this.mst = mst;
    }
}
```

Listing A.12: Kruskal’s algorithm transformed with cross-copying – recursive with varying array length ($K_{\text{CC, rec, var}}$)

```java
public class KruskalPrimitiveRecVarCrossCopying {
    public int[] mst;
    private int[] parent;
    private int[] parentSwap;
    private int find(int x) {
        if (parent[x] != x)
            return find(parent[x]);
        return x;
    }
    public void runKruskal(int[] graph) {
        int nv = graph[0];
        int[] mst = new int[(nv - 1) * 2 + 1];
        int[] mstSwap = new int[graph.length];
        parent = new int[graph.length];
        parentSwap = new int[graph.length];
        for (int i = 0; i < parent.length; ++i) {
            parent[i] = i;
        }
        int idx = 0;
        int idxSwap = 0;
        for (int i = 1; i < graph.length; i += 2) {
            int source = find(graph[i]);
            int target = find(graph[i + 1]);
            if (source != target) {
                mst[++idx] = source;
                mst[++idx] = target;
                parent[source] = target;
            } else {
                mstSwap[++idxSwap] = source;
                mstSwap[++idxSwap] = target;
                parentSwap[source] = target;
            }
        }
        mst[0] = idx / 2 + 1;
        this.mst = mst;
    }
}
```

Listing A.13: Kruskal’s algorithm transformed with conditional assignment – recursive with varying array length ($K_{\text{CA, rec, var}}$)

```java
public class KruskalPrimitiveRecVarCondAssign {
    public int[] mst;
    private int[] parent;
    private int find(int x) {
        if (parent[x] != x)
            return find(parent[x]);
        return x;
    }
    public void runKruskal(int[] graph) {
        int nv = graph[0];
        int[] mst = new int[(nv - 1) * 2 + 1];
        int[] mstSwap = new int[graph.length];
        parent = new int[graph.length];
        parentSwap = new int[graph.length];
        for (int i = 0; i < parent.length; ++i) {
            parent[i] = i;
        }
        int idx = 0;
        int idxSwap = 0;
        for (int i = 1; i < graph.length; i += 2) {
            int source = find(graph[i]);
            int target = find(graph[i + 1]);
            if (source != target) {
                mst[++idx] = source;
                mst[++idx] = target;
                parent[source] = target;
            } else {
                mstSwap[++idxSwap] = source;
                mstSwap[++idxSwap] = target;
                parentSwap[source] = target;
            }
        }
        mst[0] = idx / 2 + 1;
        this.mst = mst;
    }
}
```
return x;
}
public void runKruskal(int[] graph) {
    int nv = graph[0];
    int[] mat = new int[(nv - 1) * 2 + 1];
    parent = new int[graph.length];
    for (int i = 0; i < parent.length; ++i) {
        parent[i] = i;
    }
    int idx = 0;
    for (int i = 1; i < graph.length; i += 2) {
        /* Directly look up parent in array instead of calling
        * recursive method 'find' */
        int source = find(graph[i]);
        int target = find(graph[i + 1]);
        int m = ((source - target) >> 31) | ((target - source) >> 31);
        idx = (m & (idx + 1)) | ('m' & idx);
        mst[idx] = (m & source) | ('m' & mst[idx]);
        parent[source] = (m & target) | ('m' & source);
    }
    mst[0] = idx / 2 + 1;
    this.mst = mst;
}

Listing A.14: Kruskal’s algorithm non-recursive with varying array length (K_{flat.var})

public class KruskalPrimitiveFlatVar {
    public int[] mst;
    public void runKruskal(int[] graph) {
        int nv = graph[0];
        int[] mat = new int[(nv - 1) * 2 + 1];
        parent = new int[graph.length];
        for (int i = 0; i < parent.length; ++i) {
            parent[i] = i;
        }
        int idx = 0;
        for (int i = 1; i < graph.length; i += 2) {
/* Directly look up parent in array instead of calling
 * recursive method 'find' */
            int source = find(graph[i]);
            int target = find(graph[i + 1]);
            if (source == target) {
                mst[+idx] = source;
                mst[+idx] = target;
                /* union */
                for (int j = 0; j < parent.length; ++j) {
                    if (parent[j] == source)
                        parent[j] = target;
                }
            }
        }
        mst[0] = idx / 2 + 1;
        this.mst = mst;
    }
}

Listing A.15: Kruskal’s algorithm transformed with cross-copying – non-recursive with varying array length (K_{CC.flat.var})

public class KruskalPrimitiveFlatVarCrossCopying {
    public int[] mst;
    public void runKruskal(int[] graph) {
        int nv = graph[0];
        int[] mat = new int[(nv - 1) * 2 + 1];
        matSwap = new int[graph.length];
        parent = new int[graph.length];
        parentSwap = new int[graph.length];
        for (int i = 0; i < parent.length; ++i) {
            parent[i] = i;
        }
        int idx = 0;
        int idxSwap = 0;
        for (int i = 1; i < graph.length; i += 2) {
            int source = find(graph[i]);
            int target = find(graph[i + 1]);
            if (source == target) {
                mst[+idx] = source;
                mst[+idx] = target;
                for (int j = 0; j < parent.length; ++j) {
                    if (parent[j] == source)
                        parent[j] = target;
                }
            } else {
                matSwap[+idxSwap] = source;
                matSwap[+idxSwap] = target;
                for (int j = 0; j < parentSwap.length; ++j) {
                    mstSwap[+idxSwap] = source;
                    mstSwap[+idxSwap] = target;
                }
            }
        }
    }
}
Listing A.16: Kruskal’s algorithm transformed with conditional assignment – non-recursive with varying array length (K\textsubscript{CA}flat,\textsubscript{var})

```java
public class KruskalPrimitiveFlatVarCondAssign {
    private int[] parent;
    public int[] mst = new int[graph.length];
    public void runKruskal (int[] graph) {
        int nv = graph[0];
        int[] mst = new int[(nv - 1) * 2 + 1];
        parent = new int[graph.length];
        for (int i = 0; i < parent.length; ++i) {
            parent[i] = i;
        }
        int idx = 0;
        for (int i = 1; i < graph.length; i += 2) {
            int source = parent[graph[i]];  
            int target = parent[graph[i + 1]];    
            int m = ((source - target) >> 31) | ((target - source) >> 31);    
            idx = (m & (idx + 1)) | ('m & mst[idx]);
            mst[idx] = (m & source) | ('˜m & mst[idx]);
            for (int j = 0; j < parent.length; ++j) {
                m = ˜((parent[j] - source) >> 31) | ((source - parent[j]) >> 31);    
                parent[j] = (m & target) | ('˜m & parent[j]);
            }
        }
        mst[0] = idx / 2 + 1;
        this.mst = mst;
    }
}
```

Listing A.17: Kruskal’s algorithm recursive with fixed array length (K\textsubscript{rec,\textsubscript{fixed}})

```java
public class KruskalPrimitiveRecFixed {
    public int[] mst;
    private int[] parent;
    public int[] mst = new int[graph.length];
    public void runKruskal (int[] graph) {
        int[] mst = new int[graph.length];
        parent = new int[graph.length];
        for (int i = 0; i < parent.length; ++i) {
            mst[i] = −1;
            parent[i] = i;
        }
        int idx = 0;
        for (int i = 1; i < graph.length; i += 2) {
            int source = find(graph[i]);
            int target = find(graph[i + 1]);
            if (source != target) {
                mst[++idx] = source;
                mst[++idx] = target;
                // union
                parent[source] = target;
            }
        }
        mst[0] = idx / 2 + 1;
        this.mst = mst;
    }
}
```

Listing A.18: Kruskal’s algorithm transformed with cross-copying – recursive with fixed array length (K\textsubscript{CC}rec,\textsubscript{fixed})

```java
public class KruskalPrimitiveRecFixedCrossCopying {
    public int[] mst;
    private int[] parent;
    public int[] parentSwap = new int[graph.length];
    public void runKruskal (int[] graph) {
        int[] mst = new int[graph.length];
        int[] parent = new int[graph.length];
        for (int i = 0; i < parent.length; ++i) {
            parent[i] = i;
        }
        int idx = 0;
        for (int i = 1; i < graph.length; i += 2) {
            int source = find(graph[i]);
            int target = find(graph[i + 1]);
            if (source != target) {
                mst[++idx] = source;
                mst[++idx] = target;
                // union
                parent[source] = target;
            }
        }
        mst[0] = idx / 2 + 1;
        this.mst = mst;
    }
}
```
```java
public void runKruskal(int[] graph) {
    int[] mst = new int[graph.length];
    int[] mstSwap = new int[graph.length];
    parent = new int[graph.length];
    parentSwap = new int[graph.length];
    for (int i = 0; i < parent.length; ++i) {
        mst[i] = -1;
        parent[i] = i;
    }
    int idx = 0;
    int idxSwap = 0;
    for (int i = 1; i < graph.length; i += 2) {
        int source = find(graph[i]);
        int target = find(graph[i + 1]);
        if (source != target) {
            mst[++idx] = source;
            mst[++idx] = target;
            union(source, target);
        } else {
            mstSwap[++idxSwap] = source;
            mstSwap[++idxSwap] = target;
            parentSwap[source] = target;
        }
    }
    mst[0] = idx / 2 + 1;
    this.mst = mst;
}
```

Listing A.19: Kruskal’s algorithm transformed with transactional branching – recursive with fixed array length ($K_{CB_{rec,fixed}}$)

```java
public class KruskalPrimitiveRecFixedTransBranch {
    public int[] mst;
    private int[] parent;
    private int[] mstSwap;
    private int[] parentSwap;
    private int idx;
    private int idxSwap;
    private int find(int x) {
        if (parent[x] != x)
            return find(parent[x]);
        return x;
    }
    public void runKruskal(int[] graph) {
        this.mst = mst;
    }
}
```

Listing A.20: Kruskal’s algorithm transformed with transactional branching – recursive with fixed array length ($K_{CB_{rec,fixed}}$)
mst[i] = -1;
parent[i] = i;
}
idx = 0;
for (int i = 1; i < graph.length; i += 2) {
    int source = find(graph[i]);
    int target = find(graph[i + 1]);
    if (source != target) {
        beginT();
        mst[++idx] = source;
        mst[++idx] = target;
        parent[source] = target;
        commitT();
        beginT();
        abortT();
    } else {
        beginT();
        mst[++idx] = source;
        mst[++idx] = target;
        parent[source] = target;
        abortT();
        beginT();
        commitT();
    }
}
}
}

Listing A.21: Kruskal’s algorithm non-recursive with fixed array length ($K_{flat,fixed}$)

Listing A.22: Kruskal’s algorithm transformed with cross-copying – non-recursive with fixed array length ($K_{CC_{flat},fixed}$)
```java
public class KruskalPrimitiveFlatFixedCondAssign {
    private int[] parent;
    public int[] mst;
    public void runKruskal(int[] graph) {
        int[] mst = new int[graph.length];
        parent = new int[graph.length];
        for (int i = 0; i < parent.length; ++i) {
            mst[i] = -1;
            parent[i] = i;
        }
        int idx = 0;
        for (int i = 1; i < graph.length; i += 2) {
            int source = parent[graph[i]];
            int target = parent[graph[i + 1]];
            int m = (source - target) >>> 31 | (target - source) >>> 31);
            idx = (m & (idx + 1)) | (~m & idx);
            mst[idx] = (m & source) | (~m & mst[idx]);
            idx = (m & (idx + 1)) | (~m & idx);
            mst[idx] = (m & target) | (~m & mst[idx]);
            for (int j = 0; j < parent.length; ++j) {
                m = ~((parent[j] - source) >>> 31) | ((source - parent[j]) >>> 31));
                parent[j] = (m & target) | (~m & parent[j]);
            }
        }
        mst[0] = idx / 2 + 1;
        this.mst = mst;
    }
}
```

Listing A.23: Kruskal’s algorithm transformed with conditional assignment – non-recursive with fixed array length ($KCA_{flat, fixed}$)

```java
public class KruskalPrimitiveFlatFixedTransBranch {
    private int[] mstSwap;  
    private int[] parent;  
    private int[] parentSwap;  
    private int idx;  
    private int idxSwap;  
    public void runKruskal(int[] graph) {
        int[] mst = new int[graph.length];
        parent = new int[graph.length];
        for (int i = 0; i < parent.length; ++i) {
            mst[i] = -1;
            parent[i] = i;
        }
        idx = 0;
        for (int i = 1; i < graph.length; i += 2) {
            int source = parent[graph[i]];
            int target = parent[graph[i + 1]];
            int m = (source - target) >>> 31 | (target - source) >>> 31);
            idx = (m & (idx + 1)) | (~m & idx);
            mst[idx] = (m & source) | (~m & mst[idx]);
            idx = (m & (idx + 1)) | (~m & idx);
            mst[idx] = (m & target) | (~m & mst[idx]);
            for (int j = 0; j < parent.length; ++j) {
                m = ~((parent[j] - source) >>> 31) | ((source - parent[j]) >>> 31));
                parent[j] = (m & target) | (~m & parent[j]);
            }
        }
        mst[0] = idx / 2 + 1;
        this.mst = mst;
    }
}
```

Listing A.24: Kruskal’s algorithm transformed with transactional branching – non-recursive with fixed array length ($KTB_{flat, fixed}$)
commitT();
beginT();
abortT();
} else {
beginT();
mst[++idx] = source;
mst[++idx] = target;
for (int j = 0; j < parent.length; ++j) {
    if (parent[j] == source)
        parent[j] = target;
}
abortT();
beginT();
commitT();
}
}
}
mst[0] = idx / 2 + 1;
private void abortT() {
    idx = idxSwap;
    mst = mstSwap;
}
private void commitT() {
    private void beginT() {
    idxSwap = idx;
    mstSwap = new int[mst.length];
    System.arraycopy(mst, 0, mstSwap, 0, mst.length);
}
Figure A.13: Mutual information $I(0, r)$ in bits with 0 warm-up passes for Kruskal’s algorithm with recursion and varying array length untransformed ($K_{rec,var}$), transformed with cross-copying ($K_{CC, rec,var}$), conditional assignment ($K_{CA, rec,var}$), and transactional branching ($K_{TB, rec,var}$).

Figure A.14: Mutual information $I(5, r)$ in bits with 5 warm-up passes for Kruskal’s algorithm with recursion and varying array length untransformed ($K_{rec,var}$), transformed with cross-copying ($K_{CC, rec,var}$), conditional assignment ($K_{CA, rec,var}$), and transactional branching ($K_{TB, rec,var}$).
Figure A.15: Mutual information $I_r(10, r)$ in bits with 10 warm-up passes for Kruskal’s algorithm with recursion and varying array length untransformed ($K_{rec,var}$), transformed with cross-copying ($K_{CC,rec,var}$), conditional assignment ($K_{CA,rec,var}$), and transactional branching ($K_{TB,rec,var}$).

Figure A.16: Mutual information $I_r(0, r)$ in bits with 0 warm-up passes for Kruskal’s algorithm without recursion and varying array length untransformed ($K_{flat,var}$), transformed with cross-copying ($K_{CC,flat,var}$), conditional assignment ($K_{CA,flat,var}$), and transactional branching ($K_{TB,flat,var}$).
Figure A.17: Mutual information $I(5, r)$ in bits with 5 warm-up passes for Kruskal’s algorithm without recursion and varying array length untransformed ($K_{\text{flat, var}}$), transformed with cross-copying ($K_{\text{CC flat, var}}$), conditional assignment ($K_{\text{CA flat, var}}$), and transactional branching ($K_{\text{TB flat, var}}$).

Figure A.18: Mutual information $I(10, r)$ in bits with 10 warm-up passes for Kruskal’s algorithm without recursion and varying array length untransformed ($K_{\text{flat, var}}$), transformed with cross-copying ($K_{\text{CC flat, var}}$), conditional assignment ($K_{\text{CA flat, var}}$), and transactional branching ($K_{\text{TB flat, var}}$).
Figure A.19: Mutual information $I_i(0, r)$ in bits with 0 warm-up passes for Kruskal’s algorithm with recursion and fixed array length untransformed ($K_{rec, fixed}$), transformed with cross-copying ($K_{CC, fixed}$), conditional assignment ($K_{CA, fixed}$), and transactional branching ($K_{TB, fixed}$).

Figure A.20: Mutual information $I_i(5, r)$ in bits with 5 warm-up passes for Kruskal’s algorithm with recursion and fixed array length untransformed ($K_{rec, fixed}$), transformed with cross-copying ($K_{CC, fixed}$), conditional assignment ($K_{CA, fixed}$), and transactional branching ($K_{TB, fixed}$).
Figure A.21: Mutual information $I_r(10, r)$ in bits with 10 warm-up passes for Kruskal’s algorithm with recursion and fixed array length untransformed ($K_{rec, fixed}$), transformed with cross-copying ($K_{CC, fixed}$), conditional assignment ($K_{CA, fixed}$), and transactional branching ($K_{TB, fixed}$).

Figure A.22: Mutual information $I_r(0, r)$ in bits with 0 warm-up passes for Kruskal’s algorithm without recursion and fixed array length untransformed ($K_{flat, fixed}$), transformed with cross-copying ($K_{CC, flat, fixed}$), conditional assignment ($K_{CA, flat, fixed}$), and transactional branching ($K_{TB, flat, fixed}$).
Figure A.23: Mutual information $I(5, r)$ in bits with 5 warm-up passes for Kruskal’s algorithm without recursion and fixed array length untransformed ($K_{\text{flat, fixed}}$), transformed with cross-copying ($K'_{\text{CC, flat, fixed}}$), conditional assignment ($K'_{\text{CA, flat, fixed}}$), and transactional branching ($K'_{\text{TB, flat, fixed}}$)

Figure A.24: Mutual information $I(10, r)$ in bits with 10 warm-up passes for Kruskal’s algorithm without recursion and fixed array length untransformed ($K_{\text{flat, fixed}}$), transformed with cross-copying ($K'_{\text{CC, flat, fixed}}$), conditional assignment ($K'_{\text{CA, flat, fixed}}$), and transactional branching ($K'_{\text{TB, flat, fixed}}$)
Figure A.25: Average execution times $\bar{T}_r(w, r)$ in nano seconds for Kruskal’s algorithm with recursion and varying array length untransformed ($K_{rec, var}$), transformed with cross-copying ($K_{CC, rec, var}$), conditional assignment ($K_{CA, rec, var}$), and transactional branching ($K_{TB, rec, var}$).

Figure A.26: Average execution times $\bar{T}_r(w, r)$ in nano seconds for Kruskal’s algorithm with recursion and varying array length untransformed ($K_{flat, var}$), transformed with cross-copying ($K_{CC, flat, var}$), conditional assignment ($K_{CA, flat, var}$), and transactional branching ($K_{TB, flat, var}$).
Figure A.27: Average execution times $\bar{T}_r(w, r)$ in nano seconds for Kruskal’s algorithm with recursion and varying array length untransformed ($K_{rec, fixed}$), transformed with cross-copying ($K_{rec, fixed}$), conditional assignment ($K_{CA, fixed}$), and transactional branching ($K_{TB, fixed}$).

Figure A.28: Average execution times $\bar{T}_r(w, r)$ in nano seconds for Kruskal’s algorithm with recursion and varying array length untransformed ($K_{flat, fixed}$), transformed with cross-copying ($K_{flat, fixed}$), conditional assignment ($K_{CA, fixed}$), and transactional branching ($K_{TB, fixed}$).
A.4 Case Study 4: IDEA

Listing A.25: Modular multiplication (without context) in IDEA as implemented in FlexiProvider ($M$)

```java
private static final int mulModulus = 0x10001;
private static final int mulMask = 0xffff;
public 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;
}
```

Listing A.26: Modular multiplication in IDEA as implemented in FlexiProvider transformed with cross-copying ($M_{CC}$)

```java
private static final int mulModulus = 0x10001;
private static final int mulMask = 0xffff;
public int result skip;
public int mulMod16(int a, int b) {
    int p;
    a &= mulMask;
    b &= mulMask;
    int a skip = a;
    if (a == 0) {
        a = mulModulus - b;
        if (b == 0) {
            a skip = mulModulus - a skip;
        } else {
            p = a skip * b;
            b = p & mulMask;
            a skip = p >>> 16;
            a skip = b - a skip + (b < a skip ? 1 : 0);
        }
    } else {
        a skip = mulModulus - b;
        if (b == 0) {
            a = mulModulus - a;
        } else {
            p = a * b;
            b = p & mulMask;
            a = p >>> 16;
            a = b - a + (b < a ? 1 : 0);
        }
    }
    result skip = a skip & mulMask;
    return a & mulMask;
}
```

Listing A.27: Modular multiplication in IDEA as implemented in FlexiProvider transformed with conditional assignment ($M_{CA}$)

```java
private static final int mulModulus = 0x10001;
private static final int mulMask = 0xffff;
private static int cond(int E m, int E t, int E f) {
    return (E t & E m) | (E f & ~E m);
}
public int mulMod16(int a, int b) {
    int p = 0;
    /* (a == 0 ? -t : 0) */
    int m1 = "((a - 0) >> 31) | ((0 - a) >> 31)";
    a = cond(m1, mulModulus - b, a);
    m1 = "m1;";
    /* (b == 0 ? -t : 0) */
    int m2 = cond(m1, "((b - 0) >> 32) | ((0 - b) >> 31)", 0);
    a = cond(m2, mulModulus - a, a);
    int m3 = cond(m1, "m2", 0);
    p = cond(m3, a + b, p);
    b = cond(m3, p & mulMask, b);
    a = cond(m3, p >>> 16, a);
    a = cond(m3, b - a, a);
    int m4 = (b - a) >> 31; /* (b < a ? -t : 0) */
    a = cond(m4 & m4, a + 1, a);
}
```
private static final int mulModulus = 0x10001;
private static final int mulMask = 0xffffffff;

private int at;
private int aswp1;
private int aswp2;

private void beginT1() {
    a swp1 = at;
}
private void commitT() {}
private void abortT() {
    at = aswp1;
}
private void beginT2() {
    a swp2 = at;
}
private void commitT2() {}
private void abortT2() {
    at = aswp2;
}

private int mulMod16(int a, int b) {
    int p;
    a t = a;
    a t &= mulMask;
    b &= mulMask;
    if (a t == 0) {
        beginT1();
        at = mulModulus - b;
        commitT();
        beginT1();
        if (b == 0) {
            beginT2();
            at = mulModulus - a t;
            commitT2();
            beginT2();
            p = a t = b;
            b = p & mulMask;
            a t = p >>> 16;
            a t = b - a t + (b < a t ? 1 : 0);
            abortT2();
        } else {
            beginT2();
            at = mulModulus - a t;
            abortT2();
            beginT2();
            p = a t = b;
            b = p & mulMask;
            a t = p >>> 16;
            a t = b - a t + (b < a t ? 1 : 0);
            commitT2();
        }
    } else {
        beginT1();
        a t = mulModulus - b;
        abortT();
        beginT1();
        if (b == 0) {
            beginT2();
            at = mulModulus - a t;
            commitT2();
            beginT2();
            p = a t = b;
            b = p & mulMask;
            a t = p >>> 16;
            a t = b - a t + (b < a t ? 1 : 0);
            abortT2();
        } else {
            beginT2();
            at = mulModulus - a t;
            abortT2();
            beginT2();
            p = a t = b;
            b = p & mulMask;
            a t = p >>> 16;
            a t = b - a t + (b < a t ? 1 : 0);
            commitT2();
            commitT();
        }
    }
    return a t & mulMask;
}
Figure A.29: Mutual information $I_{M_{ed}}(0, r)$ in bits with 0 warm-up passes for IDEA in the context of method `encryptDecrypt` untransformed ($M_{ed}$), transformed with cross-copying ($M_{CC}$), conditional assignment ($M_{CA}$), and transactional branching ($M_{TB}$).

Figure A.30: Mutual information $I_{M_{ed}}(5, r)$ in bits with 5 warm-up passes for IDEA in the context of method `encryptDecrypt` untransformed ($M_{ed}$), transformed with cross-copying ($M_{CC}$), conditional assignment ($M_{CA}$), and transactional branching ($M_{TB}$).
Figure A.31: Mutual information $I(10, r)$ in bits with 10 warm-up passes for IDEA in the context of method encryptDecrypt untransformed ($M_{ed}$), transformed with cross-copying ($M_{cc}^{ed}$), conditional assignment ($M_{ca}^{ed}$), and transactional branching ($M_{tb}^{ed}$).
A.5 Summary of Case Study Results
Figure A.32: Histogram depicting the number of occurrences of mutual information results $I(10, r)$ with values in the intervals $[0, eH(X)]$ and $[eH(X), H(X)]$ or exactly equal to $H(X)$, with 10 warm-up passes for modular exponentiation untransformed ($E$), transformed with cross-copying ($E^{CC}$), conditional assignment ($E^{CA}$), and trans- actional branching ($E^{TB}$).

Figure A.33: Histogram depicting the number of occurrences of mutual information results $I(10, r)$ with values in the intervals $[0, eH(X)]$ and $[eH(X), H(X)]$ or exactly equal to $H(X)$, with 10 warm-up passes for ShareCount with multiple input arrays untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).

Figure A.34: Histogram depicting the number of occurrences of mutual information results $I(10, r)$ with values in the intervals $[0, eH(X)]$ and $[eH(X), H(X)]$ or exactly equal to $H(X)$, with 10 warm-up passes for ShareCount with a single input array untransformed ($S$), transformed with cross-copying ($S^{CC}$), conditional assignment ($S^{CA}$), and transactional branching ($S^{TB}$).
Figure A.35: Histogram depicting the number of occurrences of mutual information results $I(10, r)$ with values in the intervals $[0, \epsilon H(X)]$ and $[\epsilon H(X), H(X)]$ or exactly equal to $H(X)$, with 10 warm-up passes for Kruskal’s algorithm with recursion and varying array length untransformed ($K_{rec, var}$), transformed with cross-copying ($K_{CC, var}$), conditional assignment ($K_{CA, var}$), and transactional branching ($K_{TB, var}$).

Figure A.36: Histogram depicting the number of occurrences of mutual information results $I(10, r)$ with values in the intervals $[0, \epsilon H(X)]$ and $[\epsilon H(X), H(X)]$ or exactly equal to $H(X)$, with 10 warm-up passes for Kruskal’s algorithm without recursion and varying array length untransformed ($K_{flat, var}$), transformed with cross-copying ($K_{CC, var}$), conditional assignment ($K_{CA, var}$), and transactional branching ($K_{TB, var}$).

Figure A.37: Histogram depicting the number of occurrences of mutual information results $I(10, r)$ with values in the intervals $[0, \epsilon H(X)]$ and $[\epsilon H(X), H(X)]$ or exactly equal to $H(X)$, with 10 warm-up passes for Kruskal’s algorithm with recursion and fixed array length untransformed ($K_{rec, fixed}$), transformed with cross-copying ($K_{CC, fixed}$), conditional assignment ($K_{CA, fixed}$), and transactional branching ($K_{TB, fixed}$).
Figure A.38: Histogram depicting the number of occurrences of mutual information results $I_{(10,r)}$ with values in the intervals $[0, \epsilon H(X)]$ and $[\epsilon H(X), H(X)]$ or exactly equal to $H(X)$, with 10 warm-up passes for Kruskal’s algorithm without recursion and fixed array length untransformed ($K_{flat, fixed}$), transformed with cross-copying ($K_{CC, fixed}$), conditional assignment ($K_{CA, fixed}$), and transactional branching ($K_{TB, fixed}$).

Figure A.39: Histogram depicting the number of occurrences of mutual information results $I_{(10,r)}$ with values in the intervals $[0, \epsilon H(X)]$ and $[\epsilon H(X), H(X)]$ or exactly equal to $H(X)$, with 10 warm-up passes for IDEA in the context of method encryptDecrypt untransformed ($M_{ed}$), transformed with cross-copying ($M_{CC, ed}$), conditional assignment ($M_{CA, ed}$), and transactional branching ($M_{TB, ed}$).
Appendix B

Thesis Proposal
Timing channels—differences in running time of a computer system depending on secrets that it operates on—is an infamous class of security vulnerabilities. By observing such differences, an attacker can reconstruct the secret, either partially, or completely. This is known as a timing attack, and such attacks have been practically demonstrated on a wide range of systems, including soft- and hardware (e.g., [Koc96, FS00, BB05, BT11, GBK11]).

Side Channel Finder [LMPS10, LS11] is a prototypical tool for detection of timing channels in Java programs. It is based on static program analysis for information flow control, and, in a nutshell, searches for branchings of the program control flow that depend on (derived) secrets. Such branchings can be due to conditionals, loops, or polymorphic method calls. Side Channel Finder has been demonstrated to successfully identify a timing channel in a Java implementation of the IDEA cipher from the FlexiProvider cryptographic library [Fle12]. Subsequently a timing attack has been engineered to obtain a part of the cipher’s secret key through the detected timing channel [LS11].

Having identified timing channels like this, the next natural question that one poses is how to mitigate them. Since the early 2000s, program transformations have been suggested as a promising approach to close timing channels [Aga00]. The main advantage of program transformations compared to other techniques (e.g., blinding) is that they are defined as generic syntactic refinements of program’s source code that are independent of a particular program’s meaning. This opens up for an appealing perspective to remove timing channels from a wide range of software systems in an automated and platform-independent manner.

Several such program transformations, namely cross-copying [Aga00], transactional branching [BRW06], and conditional assignment [MPSW05], have been considered as candidates for integration in Side Channel Finder, and have been experimentally evaluated in an attempt to close the detected timing channel in IDEA. This has revealed the following fact: None of the considered transformation was capable to close the timing channel completely, although causing the corresponding attack to require more iterations.

This fact has been subsequently investigated in greater detail and it has been conjectured that such an outcome was caused by low-level platform-specific behavior of the compiled transformed code. Indeed, since none of our implementations of the considered program transformations has technically removed the branchings from the compiled code that depend
on secrets\(^1\), low-level details of such branchings could have introduced additional timing differences that have not been captured by high-level definitions of program transformations that we have considered. One example of such low-level details that we have identified is a branch predictor, a processor unit that attempts to guess branches that will be taken in the future and speculatively execute their code. In case of a mistake in the guess, the speculatively executed code has to be drained from the processor’s pipeline, which causes a delay. As delays like this are not visible in the semantics of high-level languages, there is little hope in their elimination by high-level platform-independent program transformations. At this place, a platform-specific transformation can be considered. For instance, on the Intel x86 platform branchings can be completely replaced with the help of the conditional move instruction. This, in its turn, will allow to get rid of all unwanted timing differences induced by branch predictors (as conditional moves do not use them).

All in all, this poses a trade-off dilemma for a security engineer: Accept the degree of security achieved by platform-independent transformations while benefiting from the portability and clearness of the solution, or strive for a more secure platform-specific mitigation while losing the advantages of the platform-independent solution. Further decision will needed to be taken if several transformations show comparable degree of security: Which transformation to apply?

The goal of this thesis is to design and implement a tool that will support security engineers in solving the dilemma and taking such decisions. The degree of security shall be measured based on quantitative theories of information flow.

More specifically, the solutions provided by the thesis shall support the following step-wise process for analysis and mitigation of timing channels in Java (cf. Figure 1.)

- Firstly, the security engineer uses Side Channel Finder to spot timing channels in a

\(^1\)For two transformations, cross-copying and transactional branching, this was expected: These mitigations aimed to achieve equal running time of two branchings where the branching decision depends on a secret. For the third transformation, conditional assignment, such behavior was not really expected: This mitigation aims at encoding branchings inside all expressions that were under the transformed branch in the original program. It turned out that our choice of such encoding based on Java’s ternary operator was compiled into code containing branching instructions.
• Secondly, the security engineer identifies with the help of the analysis tool whether the possible leakage through detected timing channels meets his security requirements (expressed as a quantitative measure). In case it does not, he passes the code together with the spotted places of timing channels to the mitigation tool that applies, in first place, a platform-independent transformation of his choice. In case several timing channels have been detected by Side Channel Finder, at this place he has freedom to choose whether he wants to mitigate all detected timing channels with the selected transformation at once, or in a step-wise manner. The transformed code is then passed back to the analysis tool that computes the leakage and outputs it to the security engineer. In addition to that, the analysis framework measures performance degradation induced by the transformation. In case the security engineer is not satisfied by the resulting security and performance, he can repeat the step with some other platform-independent transformation and/or other timing channels (in case there were multiple and he had chosen to tackle them in a step-wise manner).

• Thirdly, in case all platform-independent mitigations failed to achieve desired security criteria, a platform-specific transformation is applied and the analysis is repeated. In case several platform-specific mitigations are available, they can be assessed with the help of the tool support similarly to platform-independent transformations.

1 Project Objectives

1.1 Core

A: Modified JVM for Mitigation of Timing Channels This objective is to implement a platform-specific program transformation for mitigation of timing channels based on the conditional move instruction available on the Intel x86 architecture and embed this transformation into a Java Virtual Machine (JVM). This shall result in a modified JVM that takes Java bytecode obtained by compiling the Java source code with timing channels spotted by Side Channel Finder and produces native code where the spotted timing channels are mitigated.

B: Tool Support for Timing and Quantitative Analysis This objective is to design and implement tools supporting timing and quantitative analysis of timing channels detected by Side Channel Finder. As an input these tools shall take program code (source or native, original or transformed) with spotted places of timing channels detected by Side Channel Finder and the original security policy that has been supplied to Side Channel Finder to mark the secrets. On the output the tool shall provide quantitatively estimated information leakage via timing behavior and performance degradation of the transformed code.

C: Case Studies and Evaluation This objective is to design and implement at least three case studies of meaningful Java programs with timing channels and use them to evaluate the solutions of the thesis. Timing channels in these case studies shall be detected by Side Channel Finder and they shall be passed through the complete step-wise analysis and mitigation process depicted in Figure 1. Altogether, four transformations...
for mitigation of timing channels shall be considered: three platform-independent and one platform-specific. Platform-independent transformations, namely, cross-copying, transactional branching, and conditional assignment, shall be manually applied to the code of the case studies. Platform-specific transformations shall be obtained by the modified JVM from objective A.

Restrictions

R1 Only timing channels that are caused by conditionals shall be considered
R2 Code fragments under such conditionals may not have loops and method calls
R3 As a platform for the thesis Intel x86 shall be considered

1.2 Extensions

E1 Restrictions R1 and R2 can be weakened
E2 Additional case studies can be conducted to further illustrate the results of the thesis

2 Main Activities

A: Modified JVM for Mitigation of Timing Channels

A1: Annotations for Side Channel Finder Side Channel Finder shall be extended with a feature to mark the spotted timing channels with annotations. A format of these annotations shall be suggested. Such annotations shall subsequently guide the modified JVM with the information about places of potential timing channels in the analyzed programs. As the analysis process will require the security policy from Side Channel Finder in order to identify the secrets, there is an option to integrate this policy into the code in form of annotations as well.

A2: Platform-specific conditional assignment Java Virtual Machine shall be modified to protect against timing channels identified by Side Channel Finder. A modified JVM shall take annotated Java programs as an input (see A1) and transform the branches that may cause timing channels with a platform-specific implementation of conditional assignment. This implementation shall be based on the conditional assignment instruction available on the Intel x86 platform. It shall be possible to obtain the native code produced by the modified JVM in order to analyze it by the tools from objective B.

B: Tool Support for Timing and Quantitative Analysis

B1: Timing analysis Tool support for timing measurements shall be designed and implemented. As an input this tool shall receive Java source or native code with respective annotations (see A1) as well as the security policy that was supplied to Side Channel Finder. Using this, the tool shall implement functionality for generation of sample secret inputs. Further, the tool shall allow to run the received code against these generated inputs and accurately measure the running time of the code. Finally, a methodology shall be suggested for how the implemented
timing analysis can be used to compare performance of two implementations. This shall allow to analyze performance degradation induced by transformations.

**B2: Quantitative analysis** Tool support for quantitative analysis of information leakage via timing channels shall be designed and implemented. This shall be founded on the concept of mutual information. As an input this tool shall receive vectors of sample secret inputs and the corresponding measured running time as output by the timing analysis tool (see B1). The tool shall compute the mutual information between the supplied inputs and compare it against the supplied security criteria. For computation of mutual information a recently proposed approach of Chatzikokolakis, Chothia, and Guha [CCG10] that is capable of computing the mutual information from trial runs of a systems shall be used. This method is implemented in a tool Anonymity Engine [Ano12] that can calculate the mutual information and capacity of a channel from a number of observations of the inputs and outputs of a system. This tool can be deployed as a component of the tool support created in the thesis.

**C: Case Studies and Evaluation**

**C1: Design of case studies** Three meaningful case studies shall be designed and implemented as Java programs that contain timing channels. Security concerns of these case studies shall be clearly described. It is desirable that the case studies are inspired by real-world scenarios. Timing channels in the case studies shall be detected by Side Channel Finder. These timing channels shall be mitigated with platform-independent and platform-specific transformations (as stated in the objective C description before).

**C2: Performance evaluation of transformations** Performance of transformations in three case studies shall be evaluated using the methodology suggested in B1.

**C3: Quantitative analysis of case studies** Original and transformed versions of case studies shall be evaluated by the quantitative analysis tool support from B2.

**3 Deliverables**

The thesis shall include precise description of the results achieved at all three aforementioned phases of the core objectives. For activities of objective A algorithmic descriptions of transformations shall be presented. For activities of part B theory and an algorithm for quantitative analysis shall be presented. Based on results of objective C, a conclusion shall be made about characteristics of considered transformations. Detailed explanation of the made decisions (description of alternatives, discussion of advantages and disadvantages) has to be provided. Elaboration on the gained insights, identified problems, and possible extensions for the future would be a plus. A talk has to be given which will present the final results of the thesis. All developed software must be accompanied with comprehensive documentation and must be submitted together with the thesis.
4 Prerequisites

- Basic knowledge in computer security
- Basic knowledge in information theory
- Java, C, and x86 assembly programming skills

5 Language

- English or German.

6 Supervision

Prof. Dr.-Ing. Heiko Mantel (mantel@cs.tu-darmstadt.de)
Dr.-Ing. Artem Starostin (starostin@mais.informatik.tu-darmstadt.de)

References


