UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Quantification of information leakages in binaries through side-channel vulnerable runtime behaviour

Quantifizierung der Informationspreisgabe durch Ausführungscharakteristika von Binärdateien im Kontext von Seitenkanalangriffen

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Julia Gawlik

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
M. Sc. Jan Wichelmann

Lübeck, den 05. Juni 2020

Abstract

With the ongoing increase of digital data storage and processing, the confidentiality of private information has gained a high significance within the society, economy and science. One of the recent threats towards privacy is the development of side-channel attacks, exploiting the microarchitecture of the computing system to leak sensitive information to an adversary. In order to make side-channel vulnerability of software detectable, the *MicroWalk* framework offers a toolset to analyse binaries for leakages due to input-dependent runtime characteristics that might be exposed through a side-channel towards the attacker. There are different information theoretical techniques on the basis of Shannon entropy, min-entropy and guessing entropy that can provide a profound quantity measurement for identified leakages. These are explored in this thesis and evaluated for the given use case. Finally, the *MicroWalk* analysis stage is extended by an implementation of the evaluated techniques.

Je weiter die gegenwärtige Entwicklung in Richtung digitaler Speicherung und Verarbeitung von Daten voranschreitet, desto wichtiger wird auch der Schutz sensitiver Informationen. Dieser wird unter anderem auch durch die jüngere Entwicklung verschiedener Seitenkanalangriffe gefährdet, die Eigenschaften der Mikroarchitektur ausnutzen, um Informationen zu erlangen. Daher ist es für die effektive Entwicklung von Gegenmaßnahmen notwendig, Verwundbarkeiten gegen derartige Seitenkanalangriffe erkennbar zu machen. Das *MicroWalk* Framework ist eine Softwarelösung, die genau diese Aufgabe der Analyse von ausführbaren Binärdateien auf Informationspreisgabe übernimmt und den Instruktionsablauf auf eingabeabhängige Charakteristika prüft, die durch einen Seitenkanal von einem Angreifer ausgenutzt werden können. Diese Arbeit betrachtet verschiedene Quantifizierungen basierend auf den Maßen der Shannon-Entropie, Min-Entropie und Guessing-Entropie, die genutzt werden können, um eine fundierte Aussage über die Größenordnung einer Datenpreisgabe zu treffen. Diese werden anhand verschiedener Szenarien für den gegebenen Anwendungsfall evaluiert und schließlich als Teil der Analyse-Phase von *MicroWalk* implementiert.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, 05.06.2020

Acknowledgements

An erster Stelle möchte ich Professor Eisenbarth und Jan Wichelmann dafür danken, dass sie diese Bachelorarbeit erst möglich gemacht haben. Jan hat mit seiner Arbeit an dem *MicroWalk* Framework nicht nur einen wichtigen Grundstein gelegt, sondern auch geduldig den gesamten Bearbeitungszeitraum begleitet, auch wenn er sich manchmal mit sehr kurzfristigen Bitten um Feedback konfrontiert sah.
Weiterhin möchte ich meinen Korrekturlesern aus meinem Freundeskreis für ihre Zeit und ihren Einsatz danken, Korrekturen und Verbesserungsvorschläge zu erarbeiten und dabei auch noch zahlreiche motivierende Worte zu finden.
Zu guter Letzt möchte ich meiner Familie und besonders meinem Verlobten Torge Tönnies danken, der mir in jeder Hinsicht zur Seite steht, nicht nur während der Bachelorarbeit und dem Studium. Mehr denn je habe ich gespürt, wie entscheidend seine Unterstützung während dieser Arbeit für mich war, um meine Ziele im Blick zu behalten.
Der größere Teil dieser Bachelorarbeit ist während einer besonderen, durch COVID-19 geprägten Zeit entstanden, sodass ich zum Schluss anmerken möchte, was für eine gute Kommunikation und Zusammenarbeit ich dennoch mit allen genannten Menschen erfahren durfte. Auch das ist nicht selbstverständlich und daher möchte ich meinen besonderen Dank dafür ausdrücken.

Contents

1 Introduction

During an age of digital information, data privacy and confidentiality have gained a high significance within the society, economy and science. A large field of research in computer science focusses on the development, applicability and provability of means required for a secure communication and information storage.
By now, there are widely used and well proven cryptographic methods that provide confidentiality in information transfer and storage. They are proven to be secure against attackers with limited resources of computational time and space. However in practice, the algorithm itself is not the only potentially vulnerable element that has to be considered in terms of security.
Side-channel attacks, different to other attack techniques like cryptanalysis, do not target weaknesses of the algorithm, but exploit characteristics of the implementation and computational environment to gain secret information. There are different hardware features known to be exploitable for side-channel attacks and caches are one of them. There are various attacks on caches introduced in research literature targeting cryptographic implementations [Ber05, OST06]. Additionally, successful recoveries of secret keys exploiting leakages of key-dependent system activities have been shown based on cache-attacks [YB14, PGBY16].
Hence, the prevention of sensitive data being leaked via side-channels is increasingly considered in computer security research. Among others, especially the aforementioned implementations of cryptographic algorithms are highly relevant for the confidentiality of the processed information. Given an attacker that is able to observe memory accesses or time behaviour of a cryptographic process, she must not be able to learn secret information from her observation.
In detail, this implicates that there may not be any parts of the control flow of the binary that are directly or indirectly dependent from a secret value. This includes, but is not limited to, the frequency of function calls, the number of iterations within a loop as well as the count of memory accesses and the location of information within the cache.
Recent approaches in the research literature show how these requirements can be met. There are concepts of implementation techniques and technical mitigations to prevent information from leaking towards an unauthorised eavesdropper using a side-channel to spy on her victim. The approach that will be mostly focussed on in this thesis is the concept of constant time execution: It requires each run of a program to take the exact

same amount of time. Combined with invariant memory allocations and accesses, all runs of the cryptographic binary are indistinguishable and therefore to be considered secure against this kind of side-channel analysis [ABB+16].
Besides the conceptual work regarding this kind of attacker model, there is also a practical need for reliable tools to verify implementations and to detect leakages, so developers in charge of the examined implementation can efficiently identify and fix existing vulnerabilities. For this purpose, researchers from the University of Lübeck, Germany, and the Worcester Polytechnic Institute, USA, developed the *MicroWalk* framework, which is an essential basis of this thesis.

1.1 The underlying framework

Fundamental for this work is the *MicroWalk* framework, a software framework relying on binary instrumentation to detect side-channels in binaries [WMES18]. Published in 2018, *MicroWalk* searches for dependencies between a secret value and the computational states of the binary regarding memory access and control flow specifics.
With regard to the findings during the instrumented execution, the possible leakage is quantified using a Shannon entropy based mutual information analysis. This resembles the average amount of leaked data in the case that an adversary can combine measurement values, she observed, with a secret information, she does not know, and eventually learn parts of the secret by doing so.

1.2 Focus of this thesis

The *MicroWalk* framework is a valuable framework for software examination focussing on vulnerability towards side-channel attacks for researchers and developers. But to gain a significant result on the actual severity of a leakage, the mutual information analysis might not be suitable for any possible constellation of leakages. Assuming a scenario where for every possible secret exactly the same runtime behaviour occurs except for one, then an adversary will gain full knowledge of this secret, as soon as she observes the deviating execution. In this case, the average leakage would be quantified as rather small, although the attacker might learn the victims secret completely.
This thesis focusses on describing and evaluating different quantification methods existing in research literature for the specific usage within the *MicroWalk* framework. Especially the appropriate quantification of worst case leakages is discussed.
Part of the compared quantification techniques are the mutual information analysis based on the Shannon entropy, which is implemented by the *MicroWalk* framework, and another mutual information approach using min-entropy, as suggested by Wichelmann et

al. [WMES18] as a possible enhancement of the *MicroWalk* analysis for certain use cases. The min-entropy, as a part of the Rényi-family of entropies, is related to the Shannon entropy, but does not focus on expressing the average amount of information. Instead, min-entropy takes a more conservative approach and therefore emphasises the presence of secret values that are more vulnerable than others.
Additionally, this thesis considers the concept of guessing entropy to further determine the attackers probability to learn the secret from the exploited leakage using a well-defined quantification technique.
Finally, the mathematical considerations of the different quantifications are implemented and integrated in the analysis section of the *MicroWalk* framework to also evaluate the practical usage besides the theoretic discussion.
The thesis is structured accordingly. At first, Chapter 2 introduces the referred mathematical concepts and formulas used for the considered entropies and frames the assumed attacker model. The next Chapter 3 introduces the underlying framework *MicroWalk*, followed by the utilisation of the introduced formulas for leakage quantification within the framework in Chapter 4. The acquired results are discussed in Chapter 5 on a theoretic basis, using numerous examples to emphasise the advantages and disadvantages of the given approaches. Finally, Chapter 6 takes the practicability of the quantifications as a part of the implementation of the *MicroWalk* framework into account.

2 Definitions and delimitations

The classification of vulnerability of any kind essentially depends on the scope of the attacker's actions. Therefore, the first step to approach the problem that was outlined in Chapter 1 is to shape the abilities of the assumed adversary.

2.1 Attacker model

Assuming that an adversary might learn secret data from observing a program during the runtime requires a suitable attacker model to discuss abilities and boundaries of the opponent.
In order to learn information that should remain secret, solely known by the victim, by analysing the runtime behaviour and specifics of a binary, the attacker must have access to any information regarding the program execution. This includes, but is not limited to, the memory accesses performed by the program, register values at any time of the execution, timing behaviour and accessed shared resources like system libraries. All these characteristics of a single program execution the attacker can observe with a specific input value are summarised by the term *trace*.
The system on which the target binary is executed can therefore be considered a white-box [WMES18] where the attacker can legally observe every internal state of the program. Also, when it comes to the analysis of the binary, the attacker is in full control. She can choose arbitrary input values to observe the runtime behaviour for, as many as she wants before running the actual attack, and thereby build herself a knowledge base of known execution traces, whose specific characteristics she might recognise in the future if they happen to be unique.
Compared to other attacker models used in side-channel attack considerations, this is the profile of an attacker with a perfect side-channel. As long as the attacker is on the same machine and CPU, she can access all the beforehand described information about the binary execution. But at the same time she can not read the victim's input or directly access memory contents. This makes her different from highly privileged attackers on the same system as the victim, who in fact might be able to spy on the victim's keyboard inputs and files as well as RAM contents of user space processes, as conceivable in various OS-level attacks [Gen19].

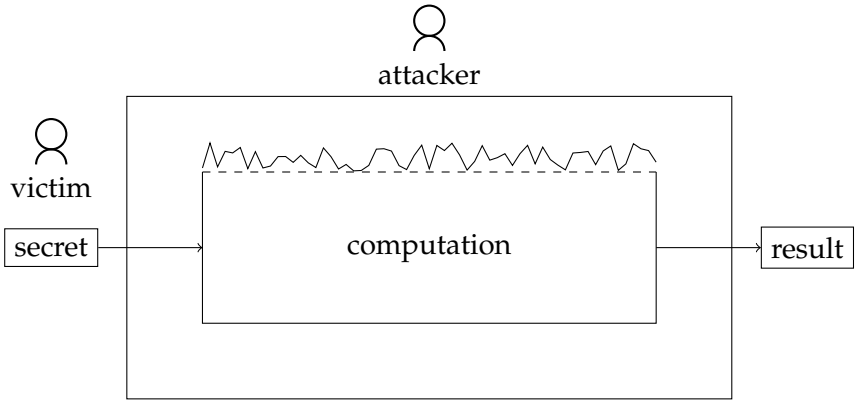

Figure 2.1: Visualisation of the attacker's action during the assumed observation of the target.

The attacker has to launch her attack in different stages, which are not uncommon for side-channel attackers exploiting runtime behaviour such as observable timing for their attack:

Stage 0 - Implementation Profiling: Before an actual attack can be executed, the attacker would profile the target implementation regarding trace characteristics of specific inputs. She would collect a preferably high number of tuples (x, t_x), where x is an input and t_x the corresponding trace observable during the program runtime. The attacker is able to do this for as many inputs as she wants, therefore her knowledge about these characteristics can be considered complete after this stage.

Stage 1 - Observation of the target: As visualised in Figure 2.1, the attacker uses her side-channel to eavesdrop on the execution behaviour of the target implementation on the secret input of the user. She gains knowledge of a trace t_{x_i} without directly learning the corresponding x_i.

Stage 2 - Recovery of the secret: Using t_{x_i} and her knowledge base from the preparation stage 0, the attacker might be able to recover information about the value x_i [KB07].

The latter two stages of the actual attack are depicted in Figure 2.2. The success of the attack in Stage 2 depends solely on the amount of information about x_i that is revealed by t_{x_i}. The more the value x_i affects the characteristics of the program execution, the higher is the attacker's probability to successfully recover x_i in Stage 2.

2.1.1 Information leakage

Therefore, the attacker should not be able to directly learn the victim's secret input (or parts of it) by just observing execution characteristics of the binary running on the input.

Provided that for any input the victim might choose it is impossible to distinguish any two executions of the program, then there is no information leakage.
But if in fact the attacker can gain information on the secret input by using her given perfect side-channel, this is called a leakage. Depending on the amount of information received by the attacker, she might be able to identify inputs that she chose in the past and that led to a similar program trace. In the worst case she can reduce the set of possible inputs from the cardinality 2^n, where n is the length of the secret, to very few or even a single value.
Sabelfeld and Myers identified two "sensitivity levels" of information processed during calculation, which they entitle *high* and *low*. Any confidential data that is entered by the user or computed during the calculation is regarded high information and thus should be protected from unauthorised access. On the other hand, there might be internal states, results and even inputs that are known by the public due to computational reasons or because they were designed to be. These are called the low information of the program and require lesser or even no protection in terms of confidentiality by definition [SM06].
As a direct conclusion, the low data should not leak parts of the high information due to lack of protection. An attacker should not be able to derive high information from observing low states or outputs of the program.
Considering the attacker model defined in this section, the concerned data can as well be divided into low and high information for the later mathematical consideration.

Data Entity	Description	Classification
X	User input	high
M	Memory contents	high
c	Computational instructions and constants	low[1]
t_x	Detailed trace for specific input $x \in X$	low
Y_i	Occurring traces at a time point i	low
T_i	Affiliation of traces to corresponding inputs $x \in X$	low

Table 2.1: Classification of the sensitivity levels of the concerned data in the given attacker model.

Let X be a set of secret input values, M a set of memory contents accessed during the calculation, and t_x a trace invoked when executing a program c on input $x \in X$. Furthermore, let $h(t_x) = y$ be the hash value of the trace (required for efficient storage of multiple traces) and T_i a set of tuples (x, y), in which the inputs x along with their corresponding trace hashes y are stored for traces up to a certain time point of execution i. Finally Y_i is

[1] This is not necessarily required by the attacker model, but due to the contemporary conception of Kerckhoffs' principle [Ker83] the security of high information may not depend on the attacker's unknowingness of neither the used algorithm nor the implementation.

the set of all occurring values y within tuples of T_i. Then these mathematical entities are partitioned in high and low information as shown in Table 2.1 due to the knowledge of the attacker in this model.
Different to many OS-level attacker models, in this case the attacker is required to have performed a successful side-channel attack beforehand to read the traces on the victim's machine. She is actually allowed or at least assumed to know about all runtime characteristics of the program execution, which are therefore classified as low information. However, the program should aim for protecting the instances of high data even facing this comparatively strong attacker. Outputs of the program that are deliberately publicly shared are not considered during the leakage analysis in this scenario.

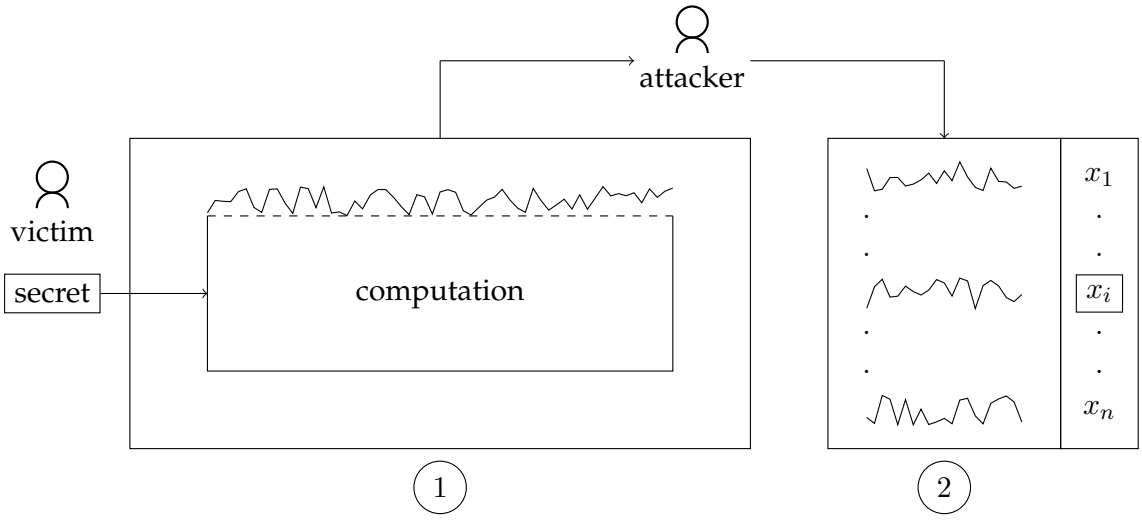

Figure 2.2: Stages of a successful side-channel attack on a system in order to learn secret data according to Köpf and Basin [KB07] applied to the considered attack scenario
(1) extract information (related to the secret key or data) from the target system
(2) effectively recover the key from the extracted information

2.2 Quantification of information

In order to describe the mathematical properties of an information leakage and to derive the severity based on the amount of knowledge the attacker can gain, it is required to use an adequate measurement for the exposed information.
Information can usually be measured by calculating corresponding entropy values. Entropy is the average rate of information that is produced by a stochastic source and is usually measured in bit. It also gives a lower bound for the average number of digits necessary to encode the outcome in a bit string over the binary alphabet.

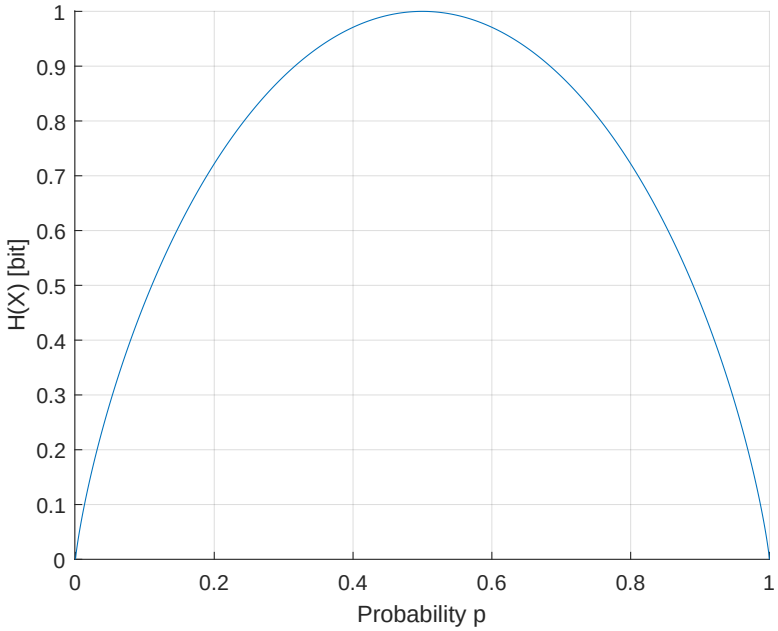

Figure 2.3: Visualised entropy $H(X)$ of a coin flipping in dependence of the probability distribution $(p, 1-p)$. The entropy in this example is calculated as $H(X) = -(p \log_2(p) + (1-p) \log_2(1-p))$.

Looking at the stochastic event of a coin flipping, the entropy depends on the probability of each side of the coin as depicted in Figure 2.3. Naturally, the entropy is the largest for uniform distributions, as there is the least a priori knowledge about the upcoming result of the flip. Since there are two possible outcomes, the flipping of a coin has the exact entropy of 1 bit (e.g. by representing heads by 1 and tails by 0) for a fair coin.

Definition 2.1. The entropy value $H(X)$ for a given random source X with a probability distribution P is calculated by the following formula summing up the weighted probabilities

$$H(X) = \sum_{x \in X} p(x) \log_2 \left(\frac{1}{p(x)} \right) \tag{2.1}$$

in bit.

This is what is called the Shannon entropy due to its introduction by Claude Shannon in a 1948 paper on information theory [Sha48].
Let X denote such a random variable as described above, then $H(X)$ is the entropy of this random variable.
The Shannon entropy is applicable for arbitrary random experiments on the basis of random variables to measure the uncertainty of the outcome and especially useful when measuring the average strength of a cryptographic secret towards an attacker.

Example 2.2.1 Usual passwords consist of lower and upper case letters with a count of 26 each in the ASCII, 10 digits and a subset of allowed special characters e.g. "!@#$%&*?". This makes $52 + 10 + 8 = 70$ characters in total to choose a password from. Considering passwords of length 6 without additional restrictions regarding the choice of characters, there are $70^6 \approx 11.7649 \times 10^{10}$ different passwords.
Assuming that these passwords are equally likely to be chosen by users, and X being the random variable to describe this choice, the Shannon entropy is calculated as follows:

$$H(X) = \sum_{i=1}^{70^6} \frac{1}{70^6} \log_2(70^6) = \log_2(70^6) \approx 36.7757$$

Further assume that the passwords have mandatory rules they need to fulfil: Each password must contain at least one special character. Then there are

$$n = \sum_{i=1}^{6} \binom{6}{i} \cdot 8^i \cdot 62^{6-i} \approx 6.0849 \times 10^{10}$$

passwords and a Shannon entropy of

$$H(X) = \sum_{i=1}^{n} \frac{1}{n} \log_2(n) = \log_2(n) \approx 35.8245$$

which is approximately 1 bit less than for the first scenario. This is conform to the observation that without the additional special character rule there were roughly twice as many valid passwords.

2.2.1 Mutual information

In information theory, the mutual information describes dependencies between two random variables. Precisely, it quantifies the amount of information that can be derived about one random variable using knowledge of the other. The following formulas were initially defined by Shannon [Sha48] and are given in numerous textbooks and research literature, of which Robert Gray's "Entropy and Information Theory" [Gra90] is referenced here besides Shannon's paper.
First, the entropy of a distribution characterised by two random variables X and Y is represented by the joint entropy $H(X, Y)$.

Definition 2.2. The joint entropy of random variables X and Y is defined as

$$H(X, Y) = -\sum_{x \in X, y \in Y} p(x, y) \log_2(p(x, y)) \tag{2.2}$$

Second, consider the entropy of a conditional probability of random variables X and Y:

Definition 2.3. Let X and Y be random variables. Then the conditional entropy $H(X|Y)$ of X conditioned on Y is defined as

$$H(X|Y) = -\sum_{x \in X, y \in Y} p(y) \cdot p(x|y) \log_2(p(x|y)) \tag{2.3}$$

From this definition, the mutual information can be deduced as the difference between the entropy of a random variable X itself and the amount of information that is gained about X when knowing an outcome of another specific random variable Y, which is given by $H(X|Y)$.

Definition 2.4. Let X and Y be random variables. Then the mutual information $I(X,Y)$ is defined as follows:

$$I(X,Y) = H(X) - H(X|Y) = \sum_{x \in X, y \in Y} p(x,y) \log_2 \left(\frac{p(x,y)}{p(x)p(y)} \right) \tag{2.4}$$

It is obvious that the mutual information of stochastically independent random variables is exactly 0 bit of information, due to the fact that the entropy of X is not lowered by the knowledge of an independent outcome of Y.
Furthermore, the mutual information of two random variables is symmetric:

Theorem 2.5.

$$H(X) - H(X|Y) = I(X,Y) = I(Y,X) = H(Y) - H(Y|X) \tag{2.5}$$

2.2.2 Rényi entropy

The Shannon entropy as introduced in the sections before is probably the most widely used information quantification in Computer Science. For example, the Shannon entropy is perfectly suitable for calculating the average number of bits required to encode the outcome of a dice roll or similar random experiment. In terms of security, the Shannon entropy can be used to judge the average strength of passwords of a given length over a certain alphabet as shown in Example 2.2.1.
However, there are use cases for which an average case consideration is not the best fit. Looking at the following example, there is a a clear gap between the actually requested result and the Shannon entropy:

Example 2.2.2 Consider again passwords of length 6 that may contain lower case and upper case letters, digits and a choice of 8 special characters. In Example 2.2.1 it was shown

that there are approximately 11.8×10^{10} different passwords with an entropy for a uniform distribution of $H(X) = 36.8$.
Given a uniform input distribution, the maximum entropy is reached as this provides the highest uncertainty about the (equally likely) values. But in reality, a uniform distribution can not always be presupposed, especially when it comes to user chosen passwords. There are other factors that influence the choice of a password than its strength: What also matters is that it can be easily memorised and exactly this circumstance tends to draw users to choose more simple and thereby "weak" passwords.
For this example, assume the probability distribution of the random variable X, describing the user's choice of the password, to be as follows: Let half of the users choose "123456" as their password (which was the most commonly used password in 2019 according to a breach analysis performed by the National Cyber Security Centre [Cen19]), and the other half pick theirs randomly and uniformly among the $70^6 - 1$ other valid passwords.
The Shannon entropy in this case computes as

$$H(X) = \frac{1}{2} \cdot \log_2(2) + \frac{1}{2} \log_2(2 \cdot (70^6 - 1)) \approx 19.3878$$

which is still a rather high value considering that an attacker can guess the password correctly in 50% of the cases. This still accurately represents the average number of bits required to encode the outcome, but it does not highlight the vast advantage an attacker gains from the given probability distribution all that well.

This example imposes the question if there is a related but more moderate quantification measure of the information, which in this case directly implies an overall security estimation on X. The answer is given by a generalisation approach on the Shannon entropy, which allows different scopes for the quantification. It gives a parametrised formula which by choice of the parameter α can produce more optimistic or more conservative results respectively on the given source X than the Shannon entropy.
Namely, this generalisation is called Rényi entropy and subsumes the Shannon entropy and other approaches to uncertainty quantifications known as the min-entropy, the collision entropy and the Hartley entropy in research literature [ABH18].
These different entropies all depict a varying confidence in the actual uncertainty of a random source. Whereas the Shannon entropy measures the average amount of information gained when learning a value from the random source, there are entropies representing a more optimistic or moderate point of view on the considered input. Considering the use case of data protection from an attacker and a leakage quantification that gives security assertions, average values might not exactly be the measurement of choice.

Therefore, the Rényi entropy in general and especially the min-entropy as a representative of this entropy family are adduced in the following [Ré61]:

Definition 2.6. The Rényi entropy of order α of a discrete random variable X, where $\alpha \in \mathbb{N}_0 \setminus \{1\}$, is defined as

$$H_\alpha(X) = \frac{1}{1-\alpha} \log_2 \left(\sum_{x \in X} p(x)^\alpha \right) \tag{2.6}$$

For different values of α and a fix probability distribution $P = (p_1, p_2 ... p_n)$ the various Rényi entropies measure the amount of uncertainty of a random variable X in different ways, except for a uniform distribution. Just like the Shannon entropy, any Rényi entropy reaches the same maximum value $H_\alpha(X) = \log n$ for the uniform distribution $P_{\text{uni}} = (\frac{1}{n}, ..., \frac{1}{n})$, independent from the choice of α.

The limiting value of H_α with $\alpha \to 1$ equals the Shannon entropy [BTBT10], whereas for $\alpha \to \infty$ it converges to the so called min-entropy H_∞, which is defined to be the most conservative measurement of uncertainty.

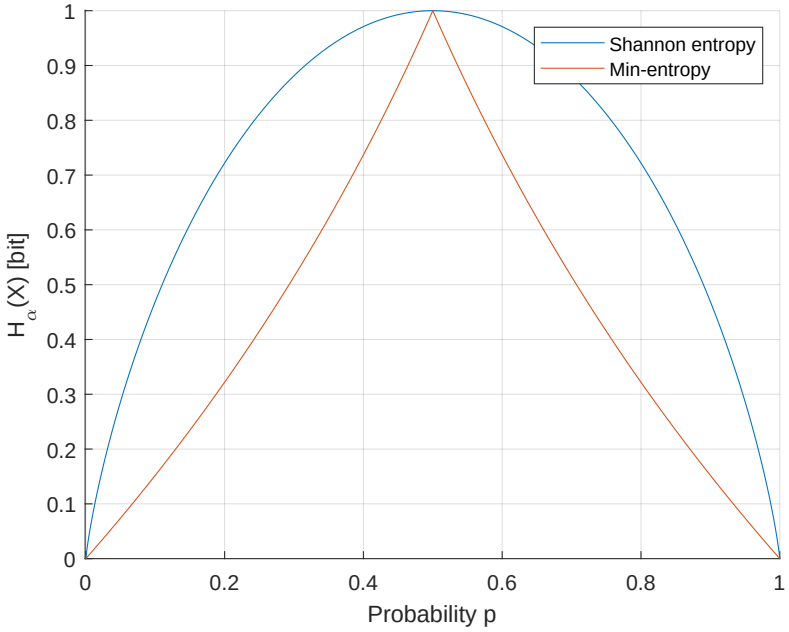

Figure 2.4: Visualised Shannon entropy $H_1(X)$ and min-entropy $H_\infty(X)$ of a coin flipping in dependence of the probability distribution $(p, 1-p)$.

2.2.3 Min-entropy

Definition 2.7. In the limit $\alpha \to \infty$ the min-entropy of X denoted with $H_\infty(X)$ is obtained from the general Rényi entropy $H_\alpha(X)$ [Smi09]:

$$H_\infty(X) = -\log_2 \left(\max_{x \in X} Pr[X = x] \right) \tag{2.7}$$

As one can derive from the definition of the min-entropy, this represents the most conservative way of measuring the unpredictability of a random source. For any distribution, it holds that the Shannon entropy, corresponding to the average uncertainty in bits, is always larger than or equal to the min-entropy.

The definition given in the following is based on the concept of vulnerability given by Smith as a part of an alternative foundation representing quantitative information flow with min-entropy instead of the well known Shannon entropy. The vulnerability $V(X)$ thereby denotes the probability that an adversary's guess on the value of X is correct on the first try in a worst case consideration [Smi09].

Definition 2.8. The vulnerability $V(X)$ of a random variable X is defined as

$$V(X) = \max_{x \in X} Pr[X = x] \tag{2.8}$$

It turns out that for this definition there is an immediate correlation to the previously given min-entropy, as

$$H_\infty(X) = \log_2 \left(\frac{1}{V(X)} \right) \tag{2.9}$$

holds.

The min-entropy is as well measured in bits just like the Shannon entropy. The amount of bits min-entropy equals the greatest lower bound for the uncertainty of a random variable X. So the min-entropy $H_\infty(X) = m$ guarantees that with each observation of X at least m bits are provided and that m is the maximum value with this property. Therefore, it is often used as a worst case measurement for unpredictability of e.g. random number generators [BK15].

However, there seems to be no consensus about the definition of a conditional min-entropy, as different approaches are found in research literature [Cac97, DRS04].

Sticking to the publication of Smith, the conditional min-entropy can be defined as follows using an underlying concept of conditional vulnerability [Smi09].

Definition 2.9. Let X and Y be random variables. Then the conditional min-entropy is given by

$$H_\infty(X|Y) = \log_2\left(\frac{1}{V(X|Y)}\right) \tag{2.10}$$

$V(X|Y)$ is the conditional vulnerability of X and Y defined as

$$V(X|Y) = \sum_{y\in Y} \max_{x\in X} Pr[Y = y|X = x]Pr[X = x] \tag{2.11}$$

All of the given formulas hold for arbitrary distributions on X and deterministic as well as probabilistic calculations. In case that the program's behaviour is deterministic for each input and X is uniformly distributed, then the calculations can be notably simplified:

$$V(X) = \frac{1}{|X|} \tag{2.12}$$

$$H_\infty(X) = \log_2(|X|) \tag{2.13}$$

$$V(X|Y) = \frac{|X|}{|Y|} \tag{2.14}$$

Example 2.2.3 Returning to the Example 2.2.2, for the given non-uniform probability distribution of the random variable X, representing the choice of password of length 6, the corresponding min-entropy calculates as

$$H_\infty(X) = -\log_2\left(\max_{x\in X} Pr[X = x]\right) = -\log_2\left(\frac{1}{2}\right) = 1$$

since there is one password, "123456", which has a probability of $\frac{1}{2}$, whereas all other passwords have a probability of $\frac{1}{2\cdot(70^6-1)}$. This depicts a much smaller quantification of the information content that is in the secret choice of an $x \in X$ in this example.
The corresponding vulnerability as introduced in this section is consequently:

$$V(X) = \frac{1}{2}$$

This shows that the vulnerability is especially high due to the existence of the given weakest value of X: The attacker is guaranteed to guess the value of X correctly in one try with a likelihood of 50%. On the other hand, it guarantees that no outcome has a probability greater than this.
However, if the attacker happens to try on the password of a user who did not choose a vulnerable value, then her success chances are low due to the large number of remaining

possible passwords. Up to now, there is no quantity used for the leakage analysis to give the expected number of guesses she would need in such a case. In order to cover this, the guessing entropy is covered next in Section 2.2.4.

2.2.4 Guessing entropy

A third approach to the characterisation of the strength and the knowledge gain of an attacker is the guessing entropy, which as well provides another point of view that should be taken into account additionally to the Shannon entropy and the min-entropy described earlier.
The guessing entropy is not exactly a mathematical quantification of the information within some kind of binary data. Different to the Shannon entropy and the min-entropy it does not provide directly a quantification of uncertainty in the average or respectively worst case. Instead, it gives the expected number of guesses an attacker $\mathcal{A}$ needs in order to find out which value x is attained by a random variable X [Mas94].
Especially of interest and not that well covered by the already introduced quantifications is the crucial case that a specific input can be indistinguishably identified when observing a corresponding trace. In an average case consideration, the impact of such a case might likely be vanishingly low, but especially for applications guarding and guaranteeing the confidentiality of user data, one extraordinary vulnerable input value might be striking enough to consider the whole program as faulty in terms of security.
Also, it is possible that the attacker can not unambiguously identify the value of X, but narrow the uncertainty down to a reasonable small amount of possible candidates. Assuming the attacker has a possibility to check her assumptions for correctness, she probably could simply brute force the correct value if there are only a few left from the initial $|X|$ values after her observation.
In order to consider this characteristic leakage type appropriately, guessing entropy might prove itself as a valuable measurement technique. The definition is given by Köpf and Basin [KB07] as follows.

Definition 2.10. The guessing entropy of a random variable X equals the average number of questions

$$\text{Is } X = x_i \text{ true?}$$

an attacker has to ask to identify the correct value of X. More formally speaking, if X has an a priori probability distribution $P = (p_1, ...p_n)$, the attacker might benefit from values that are more likely than others. Without loss of generality, X is assumed to be indexed such that the probabilities are arranged in a nonincreasing order from highest at position

p_1 to lowest at p_n. Then the guessing entropy $G(X)$ dependent of this a priori distribution can be described as:

$$G(X) = \sum_{i=1}^{n} i \cdot p_i \tag{2.15}$$

The guessing entropy for a coin flip is depicted in Figure 2.5 along with the Shannon entropy, showing that both function reach their maxima for a uniform distribution.

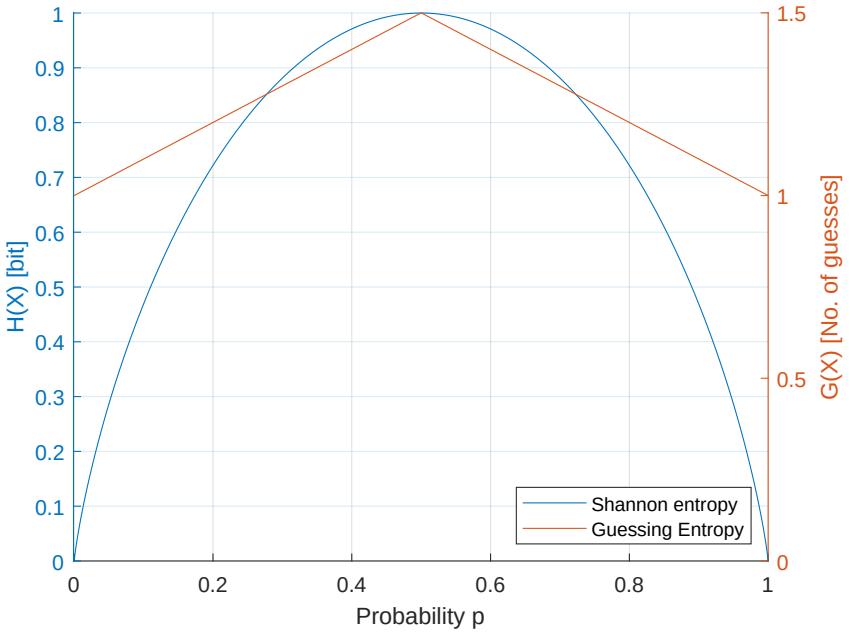

Figure 2.5: Visualised Shannon entropy $H(X)$ and corresponding guessing entropy $G(X)$ of a coin flipping in dependence of the probability distribution $(p, 1-p)$. Note that the measurement for the Shannon entropy depicted on the left-hand y-axis denotes the number of bits of information, whereas the guessing entropy on the right-hand y-axis quantifies the expected number of guesses. The two axis are unevenly aligned in order to visualise the correspondence of the functions extreme values.

In an analogous way as the guessing entropy, the conditional guessing entropy for random variables X and Y is defined:

Definition 2.11. The conditional guessing entropy $G(X|Y)$ represents the expected number of questions of the form mentioned above to determine the value of X given that the value of Y is known at this time.

$$G(X|Y) = \sum_{y \in Y} p(y) G(X|Y = y) \tag{2.16}$$

To especially cover the weakest $x \in X$, if there is such, in a worst case analysis, the minimal guessing entropy is additionally given by the following definition.

Definition 2.12. The minimal guessing entropy $\hat{G}(X|Y)$ quantifies the number of guesses an attacker needs to learn the value x of X that is the easiest to guess with knowledge of the value of Y.

$$\hat{G}(X|Y) = \min_{y \in Y} G(X|Y = y) \tag{2.17}$$

Assuming, by any coincidence, the user chooses this weakest value x as the secret input, then the attacker is expected to derive the value within $\hat{G}(X|Y)$ guesses on the basis of the performed observation.

Example 2.2.4 The concept of guessing entropy concludes the Examples 2.2.2 and 2.2.3. Continuing with the recent consideration of X as the random variable of the chosen password with one especially vulnerable value "123456" with the probability of $\frac{1}{2}$, there is still the question, how many guesses an adversary would need to attack such a password.
The guessing entropy in this scenario is given as

$$G(X) = \sum_{i=1}^{|X|} i \cdot p_i = \frac{1}{2} + \sum_{i=2}^{70^6} i \cdot \frac{1}{2 \cdot (70^6 - 1)} \approx 2.9412 \times 10^{10}$$

because there is still such a large number of possible values in half of the cases.
However, if there were no values more vulnerable than the others, as in the uniformly distributed case from Example 2.2.1, the expected number of guesses would be twice as high:

$$G(X_{\text{uni}}) = \sum_{i=1}^{70^6} i \cdot \frac{1}{70^6} = 5.8825 \times 10^{10}$$

3 Binary instrumentation

Prior to discussing leakage quantification, the presence of a certain leakage must be detected within a given binary with some method of binary instrumentation. This resembles closely the knowledge the attacker has, as presented in Section 2.1, about the exact properties of the execution of the binary on the secret input.
A framework that is able to perform a full instrumentation of the binary during runtime produces the same trace an attacker with a perfect side-channel would observe and therefore is a tool to detect possible leakages. The *MicroWalk* framework, as presented in the paper *MicroWalk: A Framework for Finding Side Channels in Binaries* by Wichelmann et al., is based on the Dynamic Binary Instrumentation (DBI) framework *Pin* [WMES18].

3.1 DBI and DBI frameworks

In research literature, typically two types of binary instrumentation are distinguished: static binary instrumentation (SBI) and dynamic binary instrumentation (DBI). In general, both techniques rely on inserting code into an existing binary. This instrumentation code can then be used to analyse the binaries' behaviour in terms of control flow and execution characteristics.
While SBI modifies the compiled binary on disk permanently using binary rewriting techniques, DBI on the other hand places the instrumentation code during the runtime of the binary. To be able to do so, DBI monitors the execution and injects the code directly into the instruction stream. Therefore, using DBI, the code section of the binary is not directly modified and the instrumentation proceeds transparently. However, due to this on the fly operating, DBI tends to slow down the execution significantly, increasing the total execution time by four times or more [And19].
Naturally, DBI supports dynamically generated instrumentation code, such as just-in-time (JIT) compiled code, which would not be possible for SBI. Furthermore, DBI does not require a correct disassembly of the binary but uses a DBI framework which performs the instrumentation during a contained execution of the binary.
A DBI framework includes an API for user-defined DBI tools specifying the instrumentation. Starting the analysis, the framework successively fetches code from the binary, adds the instrumentation according to the DBI tool and then utilises a JIT compiler to write the instrumented instructions to a code cache. Different to most other JIT compilers, there is

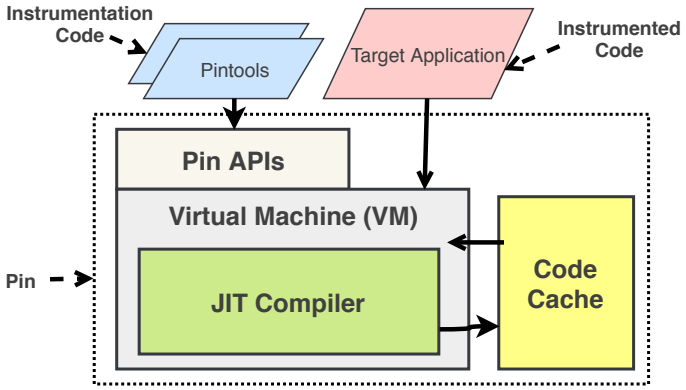

Figure 3.1: Visualisation of the main components of the DBI framework *Pin*, showing the connection between the *Pintools*, the target application that is to be instrumented, the *Pin* API and the JIT compiler, executing the combined application and instrumentation code.
Figure from the paper *MicroWalk* by Wichelmann et al. [WMES18]

no translation between languages but from machine code to machine code, ensuring that the instrumentation parts are included correctly and that there is no control transfer to the application process during the execution.

From the code cache, the instrumented code is executed until it reaches a code part that is not already present in the cache, but instead has to be prepared by the DBI engine. In this case, the control is handed over to the DBI engine in order to have it fetch and instrument the next code block like before so that the execution can be continued afterwards [And19].

3.1.1 *Pin* and *Pintools*

One widely used DBI framework is *Intel Pin*, developed and maintained as proprietary software by Intel, supporting Intel x86 and x64 CPUs [Int12]. For non-commercial purposes, the software is free to use, available for Windows, Linux and macOS[2].

The definition of *trace* as used in this thesis originates from the *Pin* JIT process, where instructions are fetched on trace granularity. Hereby, trace denotes a sequence of instructions ending on an unconditional control transfer or after a specified number of instructions. The JIT compilation of *Pin* takes place at trace granularity. However, the actual instrumentation of the code can be performed on different granularity levels ranging from the complete binary to instruction level.

[2]Published at `https://software.intel.com/content/www/us/en/develop/articles/pin-a-binary-instrumentation-tool-downloads.html` [Accessed: 23-May-2020]

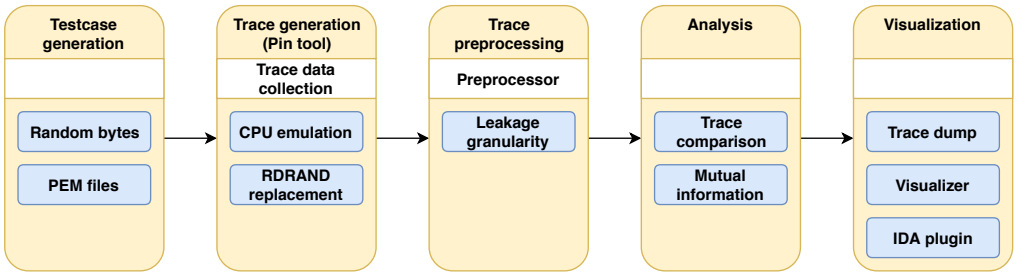

Figure 3.2: The pipeline of the *MicroWalk* framework including steps for the preparation of test cases, the instrumentation of the binary and the preprocessing of the traces that are then provided for analysis.
Figure from the paper *MicroWalk* by Wichelmann et al. [WMES18]

Pintools are DBI tools written for *Pin*, comprising the instrumentation routines and analysis routines. The first-mentioned hold the information regarding the instrumentation process, i.e. where instrumentation instructions shall be inserted. Analysis routines on the other hand are the ones containing the actual instrumentation instructions. The instrumentation routines are only executed the first time a specific code part is encountered, installing callbacks to the analysis routines that are called on every execution [And19].

3.2 *MicroWalk*: A framework for finding side channels in binaries

MicroWalk is a framework to perform an automated leakage detection and quantification for binaries, using up to this time Shannon entropy mutual information (MI) analysis as described in Section 4.1 and DBI. It aims to detect leakages in binaries occurring due to input dependent memory accesses and control flow variations.

The used DBI backend is *Pin* as described in the previous Section 3.1.1. *MicroWalk* is a tool to support the security analysis for specific binaries of interest, e.g. cryptographic libraries, by performing a white-box threat analysis, revealing if and where leakages occur in the binary. Also, the framework uses the mentioned MI analysis to give a mathematical quantification serving as a guidance level to assess the severity of an occurred leakage.

The *MicroWalk* framework operates as a pipeline, executing separate stages as depicted in Figure 3.2. In the first stage, test inputs are generated. They can either be random with a specified length or based on required input formats like for example the PEM format for cryptographic keys. Using these inputs, a custom *Pintool* generates the traces using the inputs, the instrumentation specification and the code that is to be instrumented. In this step, the DBI framework performs the DBI, runs the emulations and stores the resulting traces to disk in a custom binary format.

On those traces, the preprocessing stage adds allocation data and performs a calculation of the relative offsets of memory addresses, to e.g. further identify instruction offsets that are used as branch targets within traces. As a result, the preprocessed trace file is smaller than the raw data generated using *Pin*. Directly before passing the data to the analysis stage, the desired leakage granularity is applied, which defines the number of least significant address bits that should not be considered in the analysis.
The analysis, taking place afterwards in the analysis stage, comprises different methods: The full trace comparison, and the Shannon entropy MI analysis for either the whole traces or each single instruction. Further information on the analysis stage is given in Section 4, where also additional quantification techniques are suggested.
The results of the analysis are stored as human readable files. Furthermore, there is an option to convert the binary traces from the preprocessing stage into a human readable format. Other visualisation components are a plug-in for *IDA Pro*[3], an interactive disassembler and debugger, which is widely used in the area of reverse engineering and binary analysis [WMES18].
The *MicroWalk* framework has been published on GitHub[4]. For portability and modularity reasons, the implementation recently underwent major changes. Up to `commit #8c5fc26`[5] the version as presented in the paper *MicroWalk: A framework for finding side-channels in binaries* by Wichelmann et al. [WMES18] is present. The reimplemented version, which as well will be referenced in Chapter 6, starts from `commit #ea0b2d8`[6] on and provides an improved maintainability and extensibility of the separate pipeline stages.

[3] IDA is a product by Hex-Rays. They provide additional information on their website `https://www.hex-rays.com/products/ida/`. [Accessed 24-May-2020]
[4] `https://github.com/UzL-ITS/Microwalk` [Accessed 24-May-2020]
[5] `https://github.com/UzL-ITS/Microwalk/commit/8c5fc2666d3eb3827596c5ab57c3ec83445243e3`
[6] `https://github.com/UzL-ITS/Microwalk/commit/ea0b2d898c862519cddac716ae217081ab2c4516`

4 Quantification of binary leakages

The accurate quantification of a binary leakage detected by *MicroWalk* is an important concern to be able to classify the severity of a leakage. The quantification of information in general makes use of entropy as a quantity in information theory associated to random variables representing the data. As depicted in Section 2.2, there are different entropies that vary in specific mathematical properties and in their resulting assertions regarding the data. Thereby, different approaches lead to varying results dependent on the particular entropy used for the calculation.
Unless various measures take different views on the data, there are some relevant key values for each of the considerations [Smi09]:

- The *initial uncertainty* on the secret input.
- The amount of *information leaked* to an adversary $\mathcal{A}$.
- The *remaining uncertainty* which is not affected by the leakage.

Intuitively, the quantities considered in the following should satisfy the following [Smi09, KB07]:

$$\text{initial uncertainty} = \text{information leaked} + \text{remaining uncertainty}$$

Using the *MicroWalk* framework, the scope of the analysis can be used to add another viewing angle to the calculation of the results. It distinguishes three different analysis scopes, of which two use an entropy based leakage quantification [WMES18].

1. Trace Comparison

   This analysis technique compares two processed traces gradually. Thereby, it is checked, looking at the respective entries of both traces, whether they differ at all. If the trace is indifferent between varying inputs, then they can not be differentiated by an attacker $\mathcal{A}$.

2. Whole-trace Analysis

   Using this analysis technique, again two traces are considered as a whole. It internally processes observations on matchings between input values $x \in X$, the set of valid inputs, and hash values $y \in \{0, ..., 2^{64} - 1\}$ of a hash function h, representing

corresponding traces. A tuple (x, y) is stored if the executions of the examined program or algorithm c on input x results in a trace t_x so that the hash value $h(t_x) = y$. As a consequence, the analysis relies on judging the amount of leakage by means of equivalence classes like $\{(x', y')|y' = y\}$ for occurring y and not by observations on the traces t_y themselves.

Especially, if there are multiple leakages caused by different instructions in the examined program, all of these are assumed have their individual impact on the trace. Thereby the number of distinctive hash values for varying inputs $x \in X$ increases, the more leaking instructions are present. However, this technique is clearly limited to an aggregated view on the leakages, where each leaking instruction has its impact, but afterwards one can not tell from the stored hash value how many instructions actually affected the result.

3. Single-instruction Analysis

   This analysis technique is similar to the whole-trace analysis, but only for traces that cover exactly one specific instruction. It provides the most precise result for the single instruction, but at the same time naturally covers only the smallest part of an examined program.

4.1 Mutual information analysis with Shannon entropy

The quantity originally used in the *MicroWalk* framework for leakage quantification is a mutual information (MI) analysis. For simplification only deterministic calculations are considered and the set of inputs X is assumed to be uniformly distributed.
Given these prerequisites, then Y is a set of internal states that are attained during the program executions on the inputs, represented as hash values, and T_i is defined as the execution state at a time i from $X \times Y$. Naturally, for each input $x \in X$ there is exactly one execution state (x, y) in each set T_i, because the execution on x eventually attains state y at time point i. Let Y_i denote the set of all y occurring in tuples of T_i for an arbitrary but fixed i.
Geoffrey Smith adduces numerous research sources to conclude the consensus of mathematical entities in this consideration as follows [Smi09]:

- initial uncertainty: $H(X)$
- information leaked: $I(X, Y_i)$
- remaining uncertainty: $H(X|Y_i)$

with random variables X and Y_i representing the secret input and the attacker's observation as defined above.
Leakages detected by the *MicroWalk* framework are now quantified as follows [WMES18]:

$$I_i(X, Y_i) = \sum_{(x,y)\in T_i} \frac{1}{|X|} \log_2 \left(\frac{|T_i|}{|\{(x', y') \in T_i | y = y'\}|} \right) \tag{4.1}$$

4.2 Min-Entropy based mutual information analysis

As stated in Section 2.2.2, the widely used Shannon entropy is one part of the Rényi-family of entropies. Another is the min-entropy, which is characteristical for having the smallest values on input variables among the Rényi entropies, and thus gives a rather conservative assessment of the actual unpredictability of a source.
Just as for the MI analysis using the Shannon entropy, let X be a set of uniformly distributed input values, for which a deterministic calculation is performed. Again, Y represents the internal states of the program that are hashed and associated to the corresponding $x \in X$ by storing tuples (x, y) in sets T_i for an arbitrary but fixed i. Let Y_i denote the set of all y occurring in tuples of T_i.
Then, the leakage using min-entropy is characterised as [Smi09]

$$H_\infty(X) - H_\infty(X|Y_i) \tag{4.2}$$

which is, for deterministic programs and uniformly distributed X:

$$H_\infty(X) - H_\infty(X|Y_i) = \log_2(|Y_i|) \tag{4.3}$$

This considers the relevant entities described above according to Smith [Smi09] as

- initial uncertainty: $H_\infty(X) = \log_2(|X|)$
- information leaked: $H_\infty(X) - H_\infty(X|Y_i) = \log_2(|Y_i|)$
- remaining uncertainty: $H_\infty(X|Y_i) = \log_2\left(\frac{|X|}{|Y_i|}\right)$

Thus, only the number of equivalence classes regarding y present in T_i, which is just $|Y_i|$, do have an impact on the leakage quantification. But this consideration does not pay regard to the fact of what cardinality these equivalence classes might be. Especially, when an equivalence class contains exactly one tuple (x', y'), then x' can be unambiguously identified when observing y', which is not exactly stressed by neither of both MI analyses using the Shannon entropy or the min-entropy.

4.3 Analysis of guessing vulnerability

The guessing entropy introduced in Section 2.2.4 does not directly quantify the amount of information gained by the attacker, but the number of guesses she has to make to find the secret value X attained out of all possible and uniformly distributed input values.
In the previous sections it was described how the attacker is able to build a set Y_i from the traces containing all occurring y in tuples of T_i. Y_i hereby indicates that by every value y there is a corresponding equivalence class of values of X causing the execution to show a trace with hash value y.
Before observing the trace, the attacker has the maximum uncertainty on a uniformly distributed X. However, this uncertainty might be reduced by the knowledge of the correspondencies of T_i and the resulting equivalence classes. This consideration can be adapted analogously to the other considered entropies as follows, where the guessing difficulty, expressed by the guessing entropy, replaces the uncertainty suitably:

- initial average guessing difficulty: $G(X) = \sum_{k=1}^{|X|} k \cdot \frac{1}{|X|} = \frac{|X|+1}{2}$
- number of guesses spared by additional knowledge: $G(X) - G(X|Y_i)$
- remaining average guessing difficulty:
  $G(X|Y_i) = \sum_{y \in Y_i} p(y) \cdot G(X|Y_i = y) = \sum_{y \in Y_i} p(y) \cdot \sum_{k=1}^{|X|} k \cdot Pr[X = x_k | Y_i = y]$

This holds for the average case. For a worst case consideration it is expected that the attacker needs only one guess if at least one weakest value exists in X which is completely revealed when observing the (unique) trace t_x. So an analogous definition of relevant entities can take place using the *minimum guessing entropy*, as introduced in Section 2.2.4 as well, instead of the guessing entropy. Then the following is obtained:

- initial worst case guessing difficulty: $\hat{G}(X) = G(X)$
- number of guesses spared by additional knowledge: $\hat{G}(X) - \hat{G}(X|Y_i)$
- remaining average guessing difficulty: $\hat{G}(X|Y_i) = \min_{y \in Y_i} G(X|Y_i = y)$

Since all values of X have an equal probability due to the uniform distribution, the initial guessing difficulty is not different between worst and average case consideration.

4.4 On the uniformity of the input distribution

When taking any of the various Rényi entropies as a measurement for uncertainty, a uniform distribution always maximises the value $H_\alpha(X)$ independently from the other pa-

rameters. Furthermore, as stated before, this maximum is the very same for every Rényi entropy [Cac97].
Returning to the concept of an attacker spying on a victim, the aim must be to present a maximum amount of uncertainty about secret user inputs to the attacker. This can obviously be reached by maintaining a uniform distribution across all possible user input values. Assume that the user chooses her secret input value for a program perfectly random from the set of possible values. This might not exactly depict the reality of passphrases chosen by average users [Cen19], but it is clearly the worst case assumption from the attacker's point of view, as she has the least knowledge about the target secret.
In this situation, the attacker has a chance of guessing the secret input correctly of $\frac{1}{|X|}$, when $|X|$ is the count of possible input values. But according to the attacker model assumed for this consideration, she still can observe an execution trace of the program running on the user's secret input value, after observing as many executions on any input values she chooses.
Then the maximum uncertainty of the user's secret input might be reduced if the attacker is able to retrieve additional information from her observation. This is especially the case, when she observes a distinct part of the input values to cause a deviating timing behaviour of the binary during run time. To avoid such a inadvertent knowledge gain of the attacker, the concept of *constant time execution*, which will be covered in Section 5.1.1, is crucial.

5 Evaluation of leakage quantification measurements

Having defined the mathematical tool set for leakage quantification in Chapter 4, the different techniques can be evaluated on examples in order to understand individual advantages and drawbacks for each as well as rate the particular significance of the results.
There are a number of leakage constellations, on which the precision and reliability of the varying approaches might likely differ when giving certain security guarantees for these inputs.

5.1 Leakage free calculations

The best case to be considered is a program c that does not have exploitable leakages at all. Assuming an optimal input distribution as described in Section 4.4, then the original uncertainty of the attacker about the value of X is not lowered in any way by observing the execution and learning the traces of c on the desired value.
This can be achieved, if the trace t_x does not provide additional knowledge about x. Neither may contents of t_x be used to directly compute the value x (e.g. an accessed address is directly determined by x), nor might t_x split X in distinguishable equivalence classes in dependence of y. Especially the latter is not as easy to achieve, but there are countermeasures to prohibit this kind of information leakage. There are different actions that can be taken in order to mitigate or completely prevent high information from leaking within the execution [GYCH16], of which especially the first will be further considered in this thesis:

- Constant execution: If for input values $x_i \in X$ and any $x_j \in X \setminus \{x_i\}$ and for the corresponding traces $t_{x_i} = t_{x_j} = t_x$ holds, then the attacker gains zero knowledge from observing t_x, because it is equally likely to have been caused by x_i as well as any other input value x_j. As a consequence, c has to be invariant for all inputs regarding the information covered by the traces. This concept, especially when applied to the timing behaviour of c, is summarised in the term *constant time execution* [ABB+16].

- Randomisation / injecting noise: As for instance in the approach of *fuzzy time* [Hu91], the visible events within the computation that might be observed and evaluated by an attacker are mixed with noise to make measurements hard and thereby mitigate information leakage. Other approaches in this category suggest

adding randomisation to the cache permutation [WL07] or even making internal time sources inaccurate by fuzzing the counter [MDS12].

- Partitioning time: This approach targets the isolation of different security domains and virtual machines (VMs) by flushing the cache on context or VM switches and likewise suggestions for other components involved in the information-flow on the system.

- Partitioning hardware: By prohibiting certain hardware resources to be shared among processes or VMs, cross-party information-flow can be mitigated, as most side-channel attacks rely to a certain amount on different components shared with their victim.

From these countermeasures, constant code is the only one that can fully be controlled by the developers of an application with sensitive inputs. The others mentioned mostly rely on mitigating the attack on the underlying platform like the OS, the hypervisor or even special hardware components. So the usage of them requires the application to trust the foreign security measures and the developers can not directly control presence or absence of these measures.

5.1.1 Constant time code

One method to mitigate the impact of side-channel attacks is presenting an invariant code execution to the attacker. The information leakage depends on the existence of secret dependent control flow or timing characteristics of a program the attacker is able to observe. These characteristics include especially the execution time of the calculation as well as conditional branches and memory accesses. In general, a leakage usually emerges with multiple of these characteristics at once, as for instance in Example 5.1.1.

A leaking computation

In order to visualise the previously mentioned aspects to regard when constructing constant time computations, the first example to be considered now is a counterexample to constant time code: A computation that indeed leaks the input through the mentioned execution characteristics.
It serves as a basis to deduce a leakage free version in Example 5.1.2, which performs the same calculation but with an input independent control flow.

Example 5.1.1 Consider this function as a part of a program that uses a four digit PIN (personal identification number) as the secret value `x` in the denoted computation as follows:

Listing 5.1: Example of a computation that is not constant time.

```
1  func check_pin(String x in '0000'..'9999')
2    String x_correct = 1290
3    Integer success = 0
4
5    foreach i in 0..3 do
6      if(x[i] == x_correct[i]) then
7        success++
8      else
9        break
10     fi
11   od
12   return success
```

The loop is executed more or less often in dependence of the secret as also visualised in Figure 5.1. As soon as there is the first mismatch of a digit of the input `x` and the reference value `x_correct`, there is a break from the loop causing it to have less iterations. Thereby it shows both timing behaviour, because there are more or less loop iterations in total and a conditional branch within the loop controlling the number of iterations by checking the current digit. Additionally, the variable `success` is accessed within one branch, so there are as well differences in the memory accesses related to the secret.

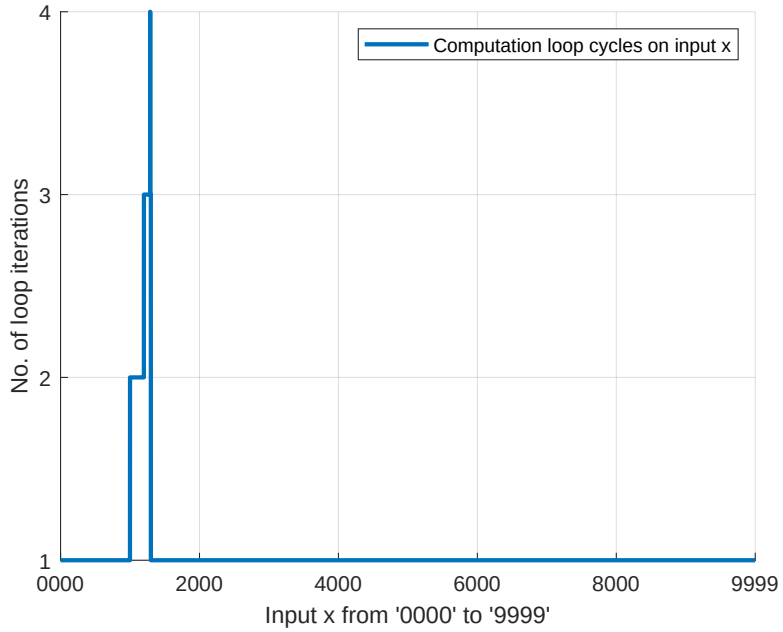

Figure 5.1: Plot of the number of loop cycles for each input of Example 5.1.1.

All these leakage characteristics are observable by the side-channel attacker defined in Section 2.1 and she can directly derive information about `x` from her observation. Whether she observed 1, 2, 3 or 4 loop iterations, she is able to narrow down the set of inputs matching the observation and, in the worst case, to learn the value of `x` directly if `x == x_correct`. This is due to the direct impact, the secret has on the control flow of the computation.

Definitions of constant time code

Any code that is written in ways to avoid all these pitfalls of the privacy of the secret is said to be constant in terms of execution. In most research literature, it is simply referred to as *constant time code*, because timing behaviour is one of the most fundamental characteristics of a leakage that comes along with almost any other. In the following parts of the thesis these terms both refer to a leakage free program with invariant executions, i.e. the observable traces are indifferent for any input value $x \in X$.
Constant time code, by definition, is reliably resistant to certain types of side-channels, primarily timing attacks, as the uncertainty of the input data is not lowered just by observing timing characteristics during the run time, since these are not distinguishable for any two inputs [Ber05].

Definition 5.1. Almeida et al. give a profound and generalized definition of constant time security as follows:

> A program is secure when:
>
> 1. Initially i-equivalent and finally o-equivalent executions are indistinguishable.
> 2. Initially i-equivalent infinite executions are indistinguishable.
>
> Otherwise, P is insecure. [ABB+16]

The so called *i-equivalence* and *o-equivalence* express the equality of low inputs and outputs, which are issued with a public accessibility by design, as a requirement for the program to be secure. In other words, there may no high inputs determine the execution so that it becomes distinguishable for an attacker observing it during the runtime, what is exactly part of the assumptions in the attacker model in Section 2.1.
Although infinite executions and public inputs / outputs are not part of the consideration in this thesis, they might as well be treatable with analogous approaches, as they share important similarities in their conceptual foundation regarding constant time requirements as well as dangers invoked by inconstancies in the execution. Almeida et al. furthermore

stated that the definition holds completely for programs that are free of public outputs, as they are by definition always finally o-equivalent [ABB+16]. A similar statement can trivially be applied to the initial i-equivalence, as non-existent public inputs are the same for each execution.

However, writing constant time code is challenging [Ber05] as not only the code itself, but also compiler optimisations [McL] and even microarchitecture instructions [LM99], naming the floating point operations in x86 processors [AKM+15], can have input depended effects on the execution time.

Performing leakage analyses on security critical programs might help detecting inadvertently leaking parts of the calculation that probably actually should have been constant time code or were assumed to be. The *MicroWalk* framework could serve as a white-box test to developers. Most helpful to this, as motivated at the beginning of this thesis, would a reliable quantification of the detected leakage and optimally a localization of the leakage within the control flow of the execution be in addition to the already existing tool set of the framework.

A similar contribution to test cryptographic applications for leakages with a completely different analysis approach has been achieved by Reparaz et al., introducing the tool `dudect` that performs leakage detection by statistically analysing execution time measurements of the target program [RBV17].

5.1.2 Leakage free computations

Example 5.1.2 Consider the following pseudo-code example of a constant time calculation and assume it to have been implemented in a way that it runs in constant time:

Listing 5.2: Example without leakage.

```
1 func check_pin(String x in '0000'..'9999')
2   String x_correct = 1290
3   Integer success = 0
4
5   foreach i in 0..3 do
6     success += (x[i] ^ x_correct[i])
7   od
8   return (success == 0)
```

In this example, similar to the first Example 5.1.1, the user provides a four digit PIN as a string that is checked against a system internal hard-coded value `x_correct` that represents the correct value. The success of the user is calculated in a way that avoids conditional jumps by counting the number of bits different between digits of the input and the target value (^ denotes the bit-wise XOR).

From an algorithmic point of view, there are apparent areas of improvements in terms of efficiency in this code: The program could simply return `false` as soon as the equality check `(x[i] == x_correct[i])` fails for the first time. Then there would be no necessity of the additional variable `success` and the correlated return statement. That would save a little memory and especially time in case that the user entered a wrong PIN.

But those algorithmic improvements would also impair the constant execution time property of the program. For the security it is important that an attacker is not able to derive the position of mismatch or to see differences in the access of the variable `success` during the calculation, because if she could, then she would gain information on the provided value x, which is what the program shall prevent.

Instead, the programs behaviour is exactly the same for any correct or incorrect `x`. The need of evaluating `x` is covered by an arithmetic evaluation, which has the same computational characteristics for each input. The attacker, being unable to directly read the value of `x` as well as any intermediate or the final value of `success` just as defined in the attacker model in Section 2.1, can thereby not draw any conclusions about certain properties of `x` and gains no information.

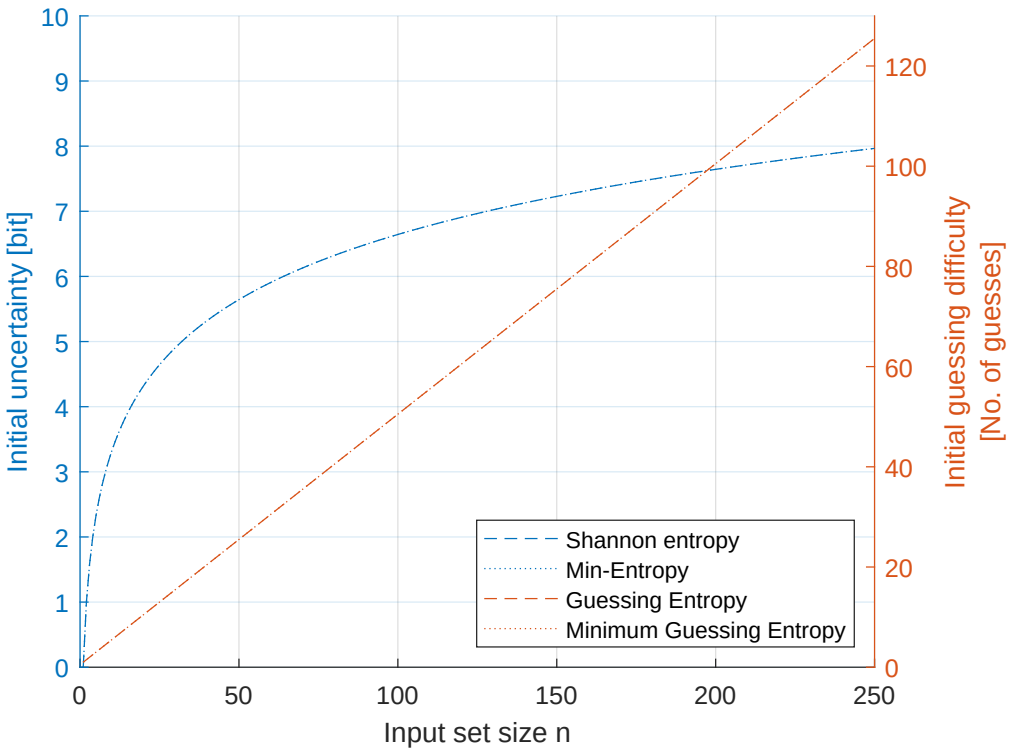

Figure 5.2: Initial uncertainty and guessing difficulty depending on $|X| = n$ for a uniformly distributed input.

Formally, considering the previously in Chapter 4 elaborated quantifications, the following observations can be made, which support the aforesaid conclusion drawn from the attacker model:
The initial uncertainty is $H(X) = \log_2(10^4)$, as there are four digits with exactly ten possible values. For the min-entropy in this case the special equality to the Shannon entropy occurs as described in Section 4.4, as they reach the same amount on uniformly distributed inputs: $H_\infty(X) = -\log_2\left(\max_{x \in X} Pr[x = X]\right) = -\log_2 \frac{1}{10^4} = \log_2(10^4)$. The attacker is expected to succeed on guessing the user PIN within $G(X) = \frac{10^4+1}{2}$ guesses.
As for the traces, the attacker can observe the following program behaviour during the runtime:

- A for-loop is executed four times in a row.
- Within the loop, one variable as well as the i-th bit of the input and of the correct value are accessed. Two operations + and ^ are performed.
- The content of one variable is returned.

Therefore, all traces t_x look exactly the same for any x for the attacker, as there are no memory accesses, conditional branches or execution time changes dependent on the specific PIN.
Concluding, none of the initial uncertainties nor the expected number of guesses required are lowered by knowing characteristics of t_x. The calculated values for the different considered mathematical quantities are listed in Table 5.1.

Average case consideration using Shannon entropy and guessing entropy			
Initial uncertainty	$H(X) = \log_2(n)$	$G(X) = \frac{n+1}{2}$	Initial number of guesses
Information leaked	$I(X, Y_i) = \log_2(1) = 0$	$G(X) - G(X\|Y_i) = 0$	Number of guesses spared
Remaining uncertainty	$H(X\|Y_i) = \log_2(n)$	$G(X\|Y_i) = \frac{n+1}{2}$	Remaining guesses
Worst case consideration using min-entropy and minimum guessing entropy			
Initial uncertainty	$H_\infty(X) = \log_2(n)$	$\hat{G}(X) = \frac{n+1}{2}$	Initial number of guesses
Information leaked	$H_\infty(X) - H_\infty(X\|Y_i) = \log_2(1) = 0$	$\hat{G}(X) - \hat{G}(X\|Y_i) = 0$	Number of guesses spared
Remaining uncertainty	$H_\infty(X\|Y_i) = \log_2\left(\frac{n}{\|Y_i\|}\right) = \log_2(n)$	$\hat{G}(X\|Y_i) = \frac{n+1}{2}$	Remaining guesses

Table 5.1: Summary of the results of leakage quantification in a leakage free example with $|X| = n$ and a uniform input distribution. Note that $|Y_i| = 1$ in this example as all $x \in X$ cause the very same trace hash y.

As visualised by the calculated values, the initial uncertainty, as also depicted in Figure 5.2 for different $|X| = n$, stays the same in this case, no matter if the attacker learns Y_i, and even in a worst case consideration constant time code with a uniformly distributed secret input delivers the same security guarantees as for the average case.

5.2 Average leakage in calculations

Although there are several recent approaches in research literature to create [Por, MKW18, ZBPB17] and to verify [RBV17, ABB+16] constant time implementations of cryptographic methods and protocols, this is still by far not the case for all programs dealing with sensitive user inputs for e.g. cryptographic operations. Thus, this section will look at examples for code leaking *some* information through their execution traces, as this is most likely the case for many applications used, contrary to the extremes of zero leakage and complete leakage covered in Section 5.1 and 5.3.

Constant leakage per input

First, consider the following example leaking a fix number of bits of the secret input.

Example 5.2.1 Let $x \in X$ be a four digit PIN just as in Example 5.1.2.

Listing 5.3: Example leaking one bit.

```
1 func lookup(String x in '0000'..'9999')
2   Integer[] lookup = {0,1}
3   Integer i = lookup[x[0] % 2]
4
5   return i
```

In this example, an array lookup is performed dependent on the value of the first digit of `x`. An attacker can thereby derive knowledge about the value of this digit by analysing the memory accesses during the program execution.
Intuitively, this means a reduction of the initial uncertainty by 1 bit of `x[0]` (odd and even values are distinguishable), which is just the result of the mutual information analysis for the leakage for both the Shannon entropy and the min-entropy based calculation:

$$I(X, Y_i) = \log_2\left(\frac{10^4}{10^3 \cdot 5}\right) = \log_2(2) = \log_2(|Y_i|) = H_\infty(X) - H_\infty(X|Y_i)$$

This is due to the fact that the input distribution is uniformly (which only affects the mutual information using the Shannon entropy and not the min-entropy) and the resulting equivalence classes by $y \in Y_i$ are of the same size. The same holds for the guessing entropy

and minimum guessing entropy respectively, where in both cases the remaining guesses required for the attacker are $\frac{(10^3 \cdot 5)+1}{2}$.

For this example, the difference between the varying quantification approaches are not apparent, as it partitions the input in same sized equivalence classes by Y_i. To work this out further, consider a slightly modified version of the same example where the inputs are not evenly partitioned by the observation of the traces.

Leakages causing an uneven partitioning of the inputs

Example 5.2.2 Let $x \in X$ again be a four digit PIN.

Listing 5.4: Example leaking with an uneven partitioning by Y_i.

```
1 func cases(String x in '0000'..'9999')
2   Integer i
3   if(x < '4000') then
4     i = 0
5   else
6     i = 1
7   fi
8   return i
```

In this example, an if-else statement is executed dependent on the value of `x`. As in Example 5.2.1, the input is partitioned into 2 equivalence classes by Y_i, but different to the first example, they are not even regarding their size. To be precise, the first class contains 4000 values (`'0000'` to `'3999'`) and the second 6000 values (`'4000'` to `'9999'`).
In this case, the mutual information calculates as

$$\begin{aligned} I(X, Y_i) &= 4000 \cdot \frac{1}{10^4} \log_2 \left(\frac{10^4}{4000} \right) + 6000 \cdot \frac{1}{10^4} \log_2 \left(\frac{10^4}{6000} \right) \approx 0.9710 \\ &< 1 = \log_2(2) = \log_2(|Y_i|) = H_\infty(X) - H_\infty(X|Y_i) \end{aligned}$$

The guessing entropy is

$$\begin{aligned} G(X|Y_i) &= \frac{2}{5} \cdot \frac{4000+1}{2} + \frac{3}{5} \cdot \frac{6000+1}{2} = \frac{5200+1}{2} \\ &> \frac{4000+1}{2} = \min_{y \in Y_i} G(X|Y_i = y) = \hat{G}(X|Y_i) \end{aligned}$$

In this example, there is a noticeable difference between the average case consideration and the worst case. Assuming the average case, the attacker gains less knowledge about the secret input and accordingly needs more guesses, to find the correct value. On the

other hand, assuming x to be a weak value increases the advantage of the attacker, raising the leaked information and lowering the expected number of guesses.

According to this, the relation of average and worst case quantification in the previous Example 5.2.1 becomes apparent. For some leaking programs, the mutual information analysis might give the same results for Shannon entropy and min-entropy under certain circumstances.
For a uniformly distributed X the following equation holds

$$\begin{aligned} I(X, Y_i) &= \sum_{(x,y)\in T_i} \frac{1}{|X|} \log_2 \left(\frac{|T_i|}{|\{(x', y') \in T_i | y = y'\}|} \right) \\ &= \log_2 \left(\frac{|X|}{|\{(x', y') \in T_i | y = y'\}|} \right) \\ &= \log_2 \left(\frac{|X|}{\frac{|X|}{|Y_i|}} \right) \\ &= \log_2(|Y_i|) \\ &= H_\infty(X) - H_\infty(X|Y_i) \end{aligned}$$

if and only if for all $y \in Y_i$ the size of $\{(x', y') \in T_i | y = y'\}$ is exactly the same and every $x \in X$ is contained in one tuple $(x, y) \in T_i$.
With a similar calculation the same constraints on the partitioning of X by Y_i lead to an equality of the guesses spared for the attacker using guessing entropy and minimum guessing entropy:

$$\begin{aligned} G(X) - G(X|Y_i) &= \frac{|X|+1}{2} - \sum_{y\in Y_i} p(y) \sum_{k=1}^{|X|} k \cdot Pr[X = x_k | Y_i = y] \\ &= \frac{|X|+1}{2} - \min_{y\in Y_i} \sum_{k=1}^{|\{(x',y')\in T_i | y=y'\}|} k \cdot \frac{1}{|\{(x', y') \in T_i | y = y'\}|} \\ &= \frac{|X|+1}{2} - \min_{y\in Y_i} \sum_{k=1}^{|X|} k \cdot Pr(X = x_k | Y_i = y) \\ &= \frac{|X|+1}{2} - \min_{y\in Y_i} G(X|Y_i = y) \end{aligned}$$

In this particular case, every $y \in Y_i$ is associated with $\{(x', y') \in T_i | y = y'\}$ values $x \in X$, and all these sets have the same size. Therefore, any y fulfils the minimum and they are

have the same impact on their respective x

i.e. $Pr[X = x_k | Y_i = y] = \begin{cases} \frac{1}{|\{(x',y') \in T_i\}|} & \text{if } y' = y \\ 0 & \text{else} \end{cases}$

However, generally speaking the average and the worst case consideration do not result in the same quantities, as they naturally depict different circumstances. A visualisation is given in Example 5.2.2.

Indirect leakage of input parts

The two previous examples dealt with cases where there was a directly leaking of input bits. As the attacker from the attacker model is capable to analyse the whole trace, there are also accumulated leakages that reduce the uncertainty about the secret input by an indirect leakage.

Consider the following example to highlight code parts that are not being constant in execution, but do not leak input bits directly. Instead, the attacker can derive properties of the secret and thereby narrow down the number of possible value.

Example 5.2.3 Let $x \in X$ be a secret user PIN just as in the examples before.

Listing 5.5: Example leaking the input partly calculating an internal sum.

```
1 func conditional(String x in '0000'..'9999')
2   Integer i = 0, a = 1
3   while (i < (x[0] + x[1])) do
4     a = a*i++
5   od
6   return a
```

In this example, a value `a` is computed based on the loop counter `i`. The number of loop iterations is determined by the sum of the first two digits of `x`.

Different to previous examples in this section, there are not directly bits from the secret leaked, but instead a property that the number l of observed loop iterations is exactly `x[0]+x[1]` because of the loop condition. This makes the leakage quantification clearly more complicated, as there are more cases that impact the number and size of equivalence classes of X determined by the attackers knowledge.

With regard to Figure 5.3 the leakage can be quantified as follows. The probability for each sum s that can occur for `x[0]+x[1]` is calculated as $p(s) = c_s \cdot 10^{-2}$ for the respective number of configurations (`x[0]`, `x[1]`) with sum s. The size of the respective equivalence class $|\{(x, y) \in T_i | x[0] + x[1] = s\}|$ is $c_s \cdot 10^2$.

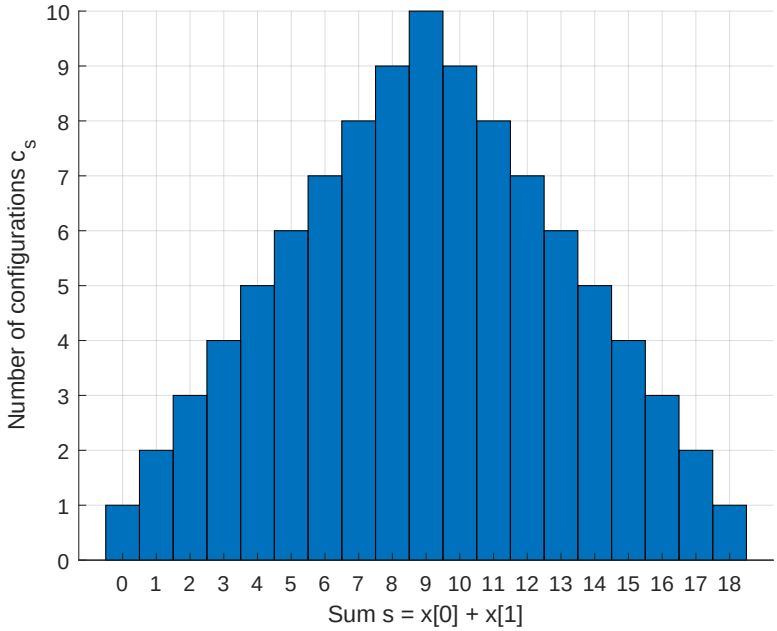

Figure 5.3: Number of configurations (`x[0], x[1]`) for each occurring sum s in Example 5.2.3. There are $10^2 = 100$ configurations of `x[0], x[1]` in total and 19 different s covering all 10^4 possibilities for $x \in X$.

The conditional Shannon entropy is then calculated as

$$\begin{aligned} H(X|Y_i) &= -\sum_{x\in X, y\in Y_i} p(y)\cdot p(x|y)\log_2(p(x|y)) \\ &= -\sum_s p(s)\log_2\left(\frac{1}{|\{(x,y)\in T_i | x[0]+x[1]=s\}|}\right) \approx 9.2572 \end{aligned}$$

where s denotes the sum of `x[0]` and `x[1]`. Accordingly the occurring leakage given by the mutual information of X and Y_i quantifies as

$$\begin{aligned} I(X,Y_i) &= \sum_{(x,y)\in T_i} \frac{1}{|X|}\log_2\left(\frac{|T_i|}{|\{(x',y')\in T_i | y=y'\}|}\right) \\ &= \sum_s \frac{c_s}{10^4}\cdot\log_2\left(\frac{10^4}{|\{(x,y)\in T_i | x[0]+x[1]=s\}|}\right) \approx 4.0306 \end{aligned}$$

bit, which is more than one quarter of the initial uncertainty $H(X) = \log_2(10^4) \approx 13.2877$.

In comparison, the min-entropy quantifies the remaining uncertainty as

$$H_\infty(X|Y_i) = \log_2\left(\frac{|X|}{|Y|}\right) = \log_2\left(\frac{10^4}{19}\right) \approx 9.0398$$

and the according leakage as

$$H_\infty(X) - H_\infty(X|Y_i) = \log_2(|Y_i|) = 4.2479$$

which is slightly more than the equivalent average case consideration using the Shannon entropy. But still, the value of the min-entropy indicates a higher leakage looking at the vulnerable input values.
In this example the attacker requires

$$\begin{aligned} G(X|Y_i) &= \sum_{y \in Y_i} p(y) \cdot \sum_{k=1}^{|X|} k \cdot Pr[X = x_k | Y_i = y] \\ &= \sum_{s} p(s) \cdot \sum_{k=1}^{|\{(x,y) \in T_i | x[0]+x[1]=s\}|} k \cdot \frac{1}{|\{(x,y) \in T_i | x[0]+x[1]=s\}|} = 192.775 \end{aligned}$$

guesses in the average case to guess the value of x correctly. Assuming the worst case with the smallest set of possible values, then the attacker is expected to need

$$\begin{aligned} \hat{G}(X|Y_i) &= \min_{y \in Y_i} \sum_{k=1}^{|X|} k \cdot Pr[X = x_k | Y_i = y] \\ &= \frac{10^2 + 1}{2} = 50.5 \end{aligned}$$

guesses to learn a "weak" value of x. In this example, especially the guessing entropy shows crucial differences between the average and the worst case assumption. The minimum guessing entropy turns out to be approximately a quarter of the guessing entropy, as there are only 10^2 different inputs in the smallest equivalence class of $x \in X$ determined by the leakage.

The results of the examples given in this chapter are accumulated in table 5.2 for direct comparability.

5.3 Worst case leakage in calculations

While the average case might be the mostly expected when examining binaries, there can also be the case that the program leaks the input or parts of the input completely.
The latter might still give relatively high security guarantees for a huge number of inputs, except for some inputs that are more vulnerable. However, if these weak inputs are nevertheless available for the choice of the user, they might be chosen by chance or because of other characteristics like memorability (as also mentioned in Example 2.2.2).

Leakage of some inputs

Example 5.3.1 As a first example the complete leakage of some inputs is exactly the case:

Listing 5.6: Example leaking some inputs completely.

```
1 func unique(String x in '0000'..'9999')
2   Integer i
3   if(x == '0000') then
4     i = 0
5   else
6     i = 1
7   fi
8   return i
```

Like before, `x` is a secret user input. This function differentiates the value of a variable `i` depending on whether `x` is exactly the special value `'0000'` or not. In this case, all traces generated by observing the program execution are invariant except for this one weakest input of all, which causes the trace to be a singleton among the others and as such uniquely identifiable.
So whenever a user chooses `'0000'` as their value for `x`, this reveals the full information to the attacker, on the other hand, the information gained for any other value is particularly small.
Therefore, the gap between average and worst case quantification means becomes significantly greater than for the consideration of average case leakages. The information leaked quantifies as follows

$$\begin{aligned} I(X, Y_i) &= \sum_{(x,y)\in T_i} \frac{1}{|X|} \log_2 \left(\frac{|T_i|}{|\{(x', y') \in T_i | y = y'\}|} \right) \\ &= \frac{1}{10^4} \log_2 \left(\frac{10^4}{1} \right) + \frac{10^4 - 1}{10^4} \log_2 \left(\frac{10^4}{10^4 - 1} \right) \approx 0.0015 \\ &< 1 = \log_2(2) = \log_2(|Y_i|) = H_\infty(X) - H_\infty(X|Y_i) \end{aligned}$$

In this example, the quantified leakage using min-entropy exceeds the Shannon entropy mutual information by more than a six hundredfold, emphasising the leakage to be highly critical for the most vulnerable high value `'0000'`.
In fact, the according gap between average and worst case guessing entropy is even more striking, as is does not just give a numerical approximation of the magnitude of a leakage, but directly corresponds to the number of guesses an adversary needs, to find out the secret value.
The number of guesses required for the attacker knowing Y_i is

$$\begin{aligned} G(X|Y_i) &= \sum_{y \in Y_i} p(y) \cdot \sum_{k=1}^{|X|} k \cdot Pr[X = x_k | Y_i = y] \\ &= \left(\frac{1}{10^4} \cdot 1 + \frac{10^4 - 1}{10^4} \cdot \frac{10^4}{2} \right) \approx 4999.5 \\ &> 1 = \min \left\{ 1, \frac{10^4 - 1}{2} \right\} \\ &= \min_{y \in Y_i} \sum_{k=1}^{|X|} k \cdot Pr[X = x_k | Y_i = y] = \hat{G}(X|Y_i) \end{aligned}$$

While in the average case the guessing entropy is nearly not reduced compared to the initial guessing entropy of $G(X) = \frac{10^1+1}{2} = 5000.5$, the minimal guessing shows that, assuming `x = '0000'`, the attacker needs only one guess to have the value of `x` correct; the value of `x` in the worst case is completely determinable from the observed trace.

Leakage of all inputs

The previous Example 5.3.1 shows, how different the security guarantees emerging from an average case and a worst case consideration can turn out. In contrary, in the case, where all inputs are similarly leaked by the program, the average and worst case should become aligned again.

Example 5.3.2 Consider the following program taking again a four digit PIN as it's input:

Listing 5.7: Example leaking the whole input.

```
1 func retrieve_input(String x in '0000'..'9999')
2   Integer[] lookup = {0,1,2,3,4,5,6,7,8,9}
3   foreach i in 0..3 do
4     lookup[x[i]]
5   od
6   return 0
```

For each digit of x there is a specific memory access done that leaks the value of x completely to an attacker observing the traces.
In this example, the amount of leakage is quantified as

$$I(X, Y_i) = \log_2\left(\frac{10^4}{1}\right) = \log_2(10^4) = H_\infty(X) - H_\infty(X|Y_i)$$

because there are 10^4 traces corresponding to exactly one value of x each.
The attacker is guaranteed to learn the secret PIN correctly within one guess:

$$G(X|Y_i) = 10^4 \cdot \frac{1}{10^4} \cdot 1 = 1 = \min_{y \in Y_i} G(X|Y = y) = \hat{G}(X|Y_i)$$

The relation between the Shannon entropy and the min-entropy mutual information depending on the number of equivalence classes of cardinality 1 is further visualised by Figure 5.4.

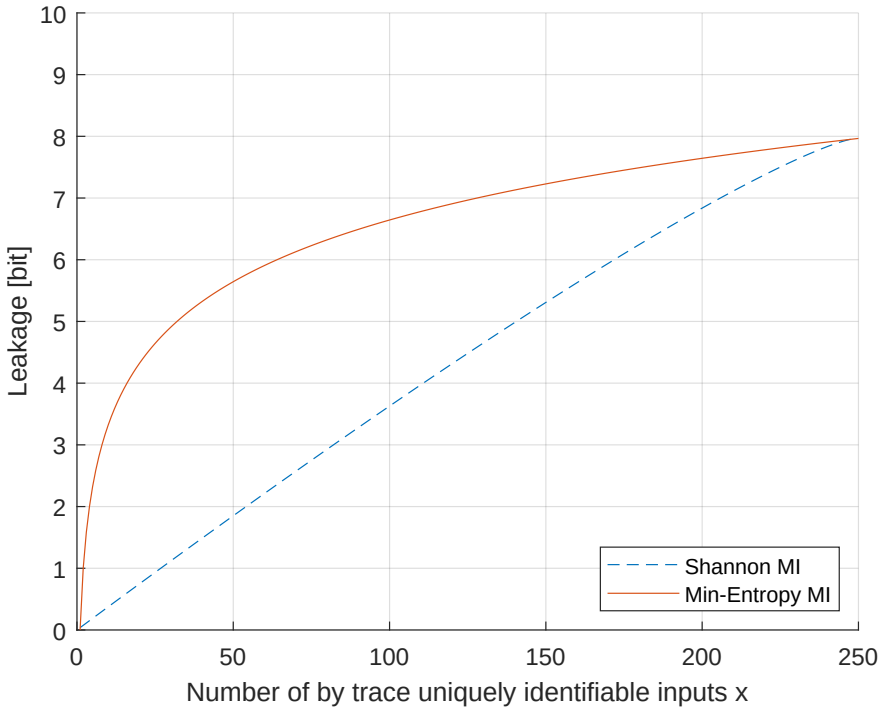

Figure 5.4: Relation of the Shannon entropy and min-entropy MI for a fixed input set size $|X| = 250$ and a variable number of inputs $x \in X$ that are completely revealed when observing the corresponding trace.

5.4 Comparison

As for the examination of a leakage free computation, Section 5.1 has shown by example and for the general case listed in Table 5.1 that there is no difference between worst case and average case consideration. This is exactly the result expected, because for a uniformly distributed input with a total leakage of 0 bit, any input value is from the attacker's perspective indifferent from all the other.

For the average case considerations, the Shannon entropy based mutual information performs quite well; the min-entropy MI is very close to Shannon entropy MI in the considered examples as listed in Table 5.2. In fact, the min-entropy emphasises the impact of the count of leakage classes emerging from the observation of the program traces on the quantity of leakage, regardless how many input values x are within each of these equivalence classes.

The behaviour of the guessing entropy compared to the minimum guessing entropy shows similar characteristics in the average case: The number of guesses spared for the attacker of guessing entropy and minimum guessing entropy are more alike, the smaller the differences in size of the leakage classes are. The minimal guessing entropy does not take into account how many leakage classes are present, but instead emphasises the size of the smallest.

If the smallest equivalence class turns out to have a size of 1, then the attacker can derive this x-value from the observation of the trace. This case is covered in the worst case consideration, where the examples show how much more impact the presence of such a small equivalence class has on the min-entropy MI and the minimal guessing entropy.

The min-entropy MI shows a much higher leakage in this case and the minimal guessing entropy judges such a program as bad as one that leaks the whole input completely.

5.4.1 Utilisation in the context of *MicroWalk*

The evaluation of the quantification techniques gives also clues about a possible utilisation for the analysis of *MicroWalk*. The aim is still to improve the preciseness and significance of the analysis results. This can support developers to reasonably rate the severity of a found leakage and estimate the possible impact on the confidentiality of the input.

First, the result of the quantification highly depends on the input characteristics. Looking at Example 5.1.2 and 5.3.1, they show in Table 5.2 nearly the same value for the Shannon entropy MI. However, the min-entropy based MI of the latter is significantly higher, as there is one input leaked completely. This has nearly zero impact on the average quantification, so the Shannon entropy MI remains close to the leakage free computation.

The benefit of this observation depends on the security requirements of the analysed binary: There might be use cases, where just one weak high value might be crucial. This is for instance the case, when an attacker attacks multiple targets and just needs to compromise one of them. Then the likelihood that one of the targets chooses the weak secret is increased with the overall number of targets and so is the success probability of the attacker. On the other hand, there might be applications where a small number of weak input values are not that critical, so the severity of the leakage complies with the average quantification again.
The second observation is that for leakages that evenly affect all input values the Shannon entropy MI and the min-entropy based MI perform equally good as in Example 5.2.1. Therefore, if the analysis stage of *MicroWalk* conducts both analyses and they turn out to be equal or close to equal, this induces the conclusion that there is a static leakage within the application that is the same for any high value. To be able to distinguish leakage types by comparing the analysis results, might be valuable for finding the root of the leakage.
Comparing the values $I(X, Y_i)$ and $H_\infty(X) - H_\infty(X|Y_i)$ that both quantify the leakage, a developer in charge of remedying the application can distinguish between leakages that occur for all / most high values (Example 5.2.1) or for rather few (Example 5.3.1) by evaluating the difference of the leakage results.
However, in the case that the applications leaks a small number high inputs completely even the min-entropy approach does not emphasise this circumstance. This is due to the fact that the min-entropy leakage solely depends on the total number of equivalence classes the attacker can create using the traces. The size of the equivalence classes is not taken into account, as visualised by Example 5.2.2 and 5.3.1 which both have a min-entropy leakage quantification of 1 bit.
Therefore to be also able to determine, if there are values, the attacker can learn completely observing the corresponding trace, the minimum guessing entropy must be considered as well. But even apart from this strong worst case scenario, the guessing entropy can provide useful information on the knowledge of the adversary after exploiting the leakage. The quantification, how many request the attacker is expected to put, might help determining mitigations such as limiting the number of failed attempts to insert the correct secret application-sided.

	Example 5.1.2	Example 5.2.1	Example 5.2.2	Example 5.2.3	Example 5.3.1	Example 5.3.2
$H(X)$	13.2877	13.2877	13.2877	13.2877	13.2877	13.2877
$I(X, Y_i)$	0	1	0.9710	4.0306	0.0015	13.2877
$H(X\|Y_i)$	13.2877	12.2877	12.3167	9.2572	13.2862	0
$H_\infty(X)$	13.2877	13.2877	13.2877	13.2877	13.2877	13.2877
$H_\infty(X) - H_\infty(X\|Y_i)$	0	1	1	4.2479	1	13.2877
$H_\infty(X\|Y_i)$	13.2877	12.2877	12.2877	9.0398	12.2877	0
$G(X)$	5000.5	5000.5	5000.5	5000.5	5000.5	5000.5
$G(X) - G(X\|Y_i)$	0	2500	2400	4807.725	1	4999.5
$G(X\|Y_i)$	5000.5	2500.5	2600.5	192.775	4999.5	1
$\hat{G}(X)$	5000.5	5000.5	5000.5	5000.5	5000.5	5000.5
$\hat{G}(X) - \hat{G}(X\|Y_i)$	0	2500	3000	4950	4999.5	4999.5
$\hat{G}(X\|Y_i)$	5000.5	2500.5	2000.5	50.5	1	1
Number of leakage classes	1	2	2	19	2	$10,000$
Size of leakage classes	$10,000$	5000	Varying 4000 / 6000	Varying 100 to 1000	Varying 1 / 9999	1
Focus	best case: leakage free	average case: same size leakage classes	average case: different size leakage classes	average case: different probability of leakage classes	worst case: one input leaked	worst case: all inputs leaked

Table 5.2: Overview of the results for all given examples in Section 5.1 to 5.3 approximated to 10^{-4}.

6 Implementation

Having the quantification techniques evaluated for different scenarios of occurring leakage, the next step is implementing them within the *MicroWalk* analysis stage and executing them on binaries.
The stages of the *MicroWalk* framework are implemented in C#, the *PinTracer* and *PinTracerWrapper* used for the DBI are external C++ components. The configuration is provided by a `.yaml` file that defines the execution parameters of the framework in several configuration blocks[7]:

Dictionary	**Contents**
`general`	Logging configuration
`testcase`	Testcase generation parameters
`trace`	Pin and image configuration
`preprocess`	Preprocessing options
`analysis`	Analysis stage configuration

Table 6.1: Top-level configuration blocks for *MicroWalk* in the `config.yaml`.

Each of the configuration blocks listed in Table 6.1 except for `general`, which only contains the logger, has the following subordinate elements: `module`, `module-options` and `options`. Most of the stages contain exactly one of each. The `analysis` stage, different to the other stages, allows the user to store a list of multiple modules within the block `modules`, each with own `module-options`. This structure is depicted in Figure 6.1 for two different analysis-modules exemplarily.

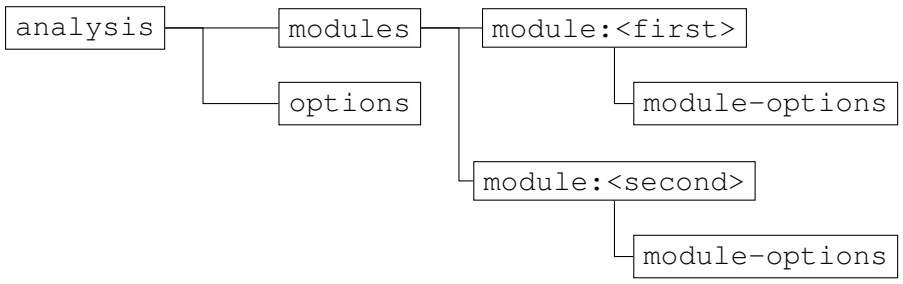

Figure 6.1: Visualisation of the configuration structure of the analysis stage.

[7]Further information on the YAML syntax can be found the YAML specification `https://yaml.org/spec/1.2/spec.html` [Accessed: 26-May-2020]

Listing 6.1: Configuration of the analysis stage with additional modules.

```
analysis:
  modules:
    - module: dump
      module-options:
        output-directory: <Path>\traces
        include-prefix: true
        map-files:
          - /tmp/a.map
          - /tmp/b.map
    - module: memory-access-mi
      module-options:
        output-directory: <Path>\results
    - module: memory-access-minentropy-mi
      module-options:
        output-directory: <Path>\results
    - module: memory-access-guessingentropy
      module-options:
        output-directory: <Path>\results
    - module: memory-access-minguessingentropy
      module-options:
        output-directory: <Path>\results

  options:
    keep-traces: true
    max-parallel-threads: 2
```

6.1 Configuration

The described configuration has to be customised for the implementation of new analysis modules. First, any paths used in the configuration need to be adjusted for the execution environment. The installation path of Intel Pin and of the builds of the *MicroWalk PinTracer* and *PinTracerWrapper* have to be provided accordingly to the locations of the respective executables. The output directories for the logs, testcases, traces and analysis results require customisation as well.

The next step is to register the new analysis modules in the `modules` list of the `analysis` stage as depicted in Figure 6.1. Analogous to the existing `memory-access-mi` module, each of the new modules has subordinate `module-options` specifying the `output-directory`.

The emerging configuration of the analysis stage for the three additional modules looks as shown in the Listing 6.1.

6.2 Module implementation

Additional modules are implemented in the same way as the existing `MemoryAccessMi.cs`. The class `Analysis.cs` provides a generalised abstract base class.

Each analysis module has a reference that serves as a handle for the configuration file, consisting of the name and a description.

Listing 6.2: The framework module declaration.

```
1 [FrameworkModule("<Name>", "<Description>")]
```

The input consists of a map of test case IDs and maps of instructions and observed hashes. So the mapping is twofold: For each test case k there is a map of instructions i_j that were encountered during the execution of the instrumented application for k. For each i_j in the map, the corresponding t_{i_j} is stored along with it. Then t_{i_j} is the hash of the trace observed for instruction i_j during the execution of the application on test case k.

This input is then reshaped to fit an instruction level analysis. For each occurring instruction a list is generated that stores the `instructionData` for this instruction. Each entry of this complex data type consists of a count of test cases, in which the instruction occurred, and a dictionary `HashCounts` storing any encountered trace hash for this instruction along with the number, for how many test cases this trace was observed.

6.2.1 Implementation of the min-entropy leakage

Recall that the min-entropy leakage for uniformly distributed inputs, which is the case for the random generated test cases of *MicroWalk*, is calculated as

$$H_\infty - H_\infty(X|Y_i) = \log_2(|Y_i|)$$

The value $|Y_i|$ is provided as the number of hash values in the `HashCounts`-dictionary. Then the leakage can be computed as follows in a newly implemented analysis module:

Listing 6.3: Implementation of the min-entropy MI analysis.

```
1 // Calculate min-entropy mutual information of each instruction
2 foreach (var instruction in instructions)
3 {
4   // Since the keys are uniquely generated, we have a uniform distribution
5   // Calculate min-entropy mutual information
6   double minEntropyMutualInformation =
7     Math.Log(instruction.Value.HashCounts.Count,2);
8 }
```

This value is then stored in a map of instructions for the currently evaluated instruction. The result of the analysis is printed to a human readable text file.

6.2.2 Implementation of the guessing entropy

For the Shannon entropy and the min-entropy approaches, the value of concern to be calculated for each instruction is clearly the amount of leakage occurring for the instruction. The result is very comprehensible and can easily be used for the analysis of the leakage.
However, for the guessing entropy approach, the amount of remaining guesses, after the attacker successfully observed and exploited a present leakage, is more valuable than considering the number of guesses the attacker can skip by exploiting the leakage. The number of remaining guesses is quantified by

$$G(X|Y_i) = \sum_{y \in Y_i} p(y) \cdot \sum_{k=1}^{|X|} k \cdot Pr[X = x_k | Y_i = y]$$

This value can be computed using the respective hash counts in the `HashCounts`-dictionary and the number of test case. In the original guessing entropy formula the probabilities $Pr[X = x_k|Y = y]$ had to be ordered nonincreasingly beforehand, as this is a requirement for the guessing entropy formula.
In this case, since the input values are uniformly distributed, there are only two possible probabilities $Pr[X = x_k|Y_i = y]$:

$$Pr[X = x_k|Y_i = y] = \begin{cases} \frac{1}{|\{(x',y') \in T_i\}|} & \text{if } y' = y \\ 0 & \text{else} \end{cases}$$

Traces that never occurred for an instruction are not stored, therefore it is sufficient to only consider the nonzero probabilities, which are also all equal for the test cases. Then, the formula can be simplified as follows in Listing 6.4.

Listing 6.4: Implementation of the conditional guessing entropy analysis.

```
1  // Calculate the conditional guessing entropy of each instruction
2  foreach (var instruction in instructions)
3  {
4    double conditionalGuessingEntropy = 0.0;
5    // Since the keys are uniquely generated, we have a uniform distribution
6    foreach (var hash in instruction.Value.HashCounts)
7    {
8      double pY = (double)hash.Value / instruction.Value.TestcaseCount;
9      conditionalGuessingEntropy += pY * (hash.Value + 1.0) / 2;
10   }
11 }
```

Again, the computed value `conditionalGuessingEntropy` is stored in a map with the instruction it has been computed for and printed line-wise to a text file.

6.2.3 Implementation of the minimum guessing entropy

The implementation of the minimum guessing entropy is very similar to the guessing entropy implementation, as the required calculation is

$$\hat{G}(X|Y_i) = \min_{y \in Y_i} \sum_{k=1}^{|X|} k \cdot Pr[X = x_k | Y = y]$$

The implementation computes the `conditionalGuessingEntropy`, which represents $G(X|Y_i = y)$, for each hash value and keeps the minimum in the end.

Listing 6.5: Implementation of the conditional minimum guessing entropy analysis.

```
1  // Calculate the conditional guessing entropy of each instruction
2  foreach (var instruction in instructions)
3  {
4    double minimumConditionalGuessingEntropy = double.MaxValue;
5    double conditionalGuessingEntropy = 0.0;
6    // Since the keys are uniquely generated, we have a uniform distribution
7    foreach (var hash in instruction.Value.HashCounts)
8    {
9      conditionalGuessingEntropy = (hash.Value + 1.0)/2;
10     if (conditionalGuessingEntropy < minimumConditionalGuessingEntropy)
11     {
12       minimumConditionalGuessingEntropy = conditionalGuessingEntropy;
13     }
14   }
15 }
```

The results are stored for each instruction and printed to a text file.

6.3 Testing the implementation

To assure the functionality of the newly implemented task, there were several tests executed using at first an own `.dll` that was purposefully implemented to show a certain leakage. The sample takes 1 byte as the input value. To reach a full coverage of input samples, the test case generation has been modified for these examples to not use a random function, but instead provide each possible input value exactly once. As a result, 256 test cases were run on each example.

6.3.1 A leakage free sample library

Example 6.3.1 At first, just as in Section 5, the `.dll` implements a leakage free calculation, avoiding input dependencies completely. The function called is the `pin_checker` for the aforementioned input of one byte.

Listing 6.6: Test code of a leakage free computation.

```
1 volatile int a[255];
2 int pin_checker(uint8_t pin) {
3   return a[0];
4 }
```

Since there are no input dependent operations and instead a constant memory access, the function is not supposed to leak at all. The results exactly matches this expectation:

Memory Access Mutual Information	
Instruction 0x0116B7: `a[0]`	0.000 bits
Other Instructions	0.000 bits
Min-Entropy Memory Access Mutual Information	
Instruction 0x0116B7: `a[0]`	0.000 bits
Other Instructions	0.000 bits
Guessing Entropy	
Instruction 0x0116B7: `a[0]`	128.500 guesses
Other Instructions	128.500 guesses
Minimum Guessing Entropy	
Instruction 0x0116B7: `a[0]`	128.500 guesses
Other Instructions	128.500 guesses

Table 6.2: Summary of the results for the leakage free test.

The results of the tests are summarised in Table 6.2. Both mutual information techniques show no signs of leakage and the guessing entropies stay at their highest possible value (for test case size 256).

6.3.2 Average leakage sample libraries

In Chapter 5 there were two primary cases of average leakage considered. First, a small leakage affecting all input values, and second, a leakage affecting a small number of input values. In this order, suitable examples are also evaluated for the implementation.

Example 6.3.2 In the first example, there is a calculation dependent on the PIN being either even or odd. The expected leakage is 1 bit for both the Shannon entropy MI analysis and the min-entropy MI, as there are exactly 2 equivalence classes of PINs distinguishable by their traces and each PIN leaks 1 bit within the leaking instruction. There should just be one leaking instruction, as there is only one leakage dependent operation. The guessing entropy is expected to be approximately half of the leakage free guessing entropy.

Listing 6.7: Test code of a calculation leaking one bit for each input.

```
1 volatile int a[255];
2 int pin_checker(uint8_t pin) {
3   return a[pin % 2];
4 }
```

The results are listed in Table 6.3. As assumed, both MI analyses return the same quantities of leakage, which is 1 bit for the leaking memory access instruction, and 0 for the others.

Memory Access Mutual Information	
Instruction 0x0116DD: `a[pin%2]`	1.000 bits
Other Instructions	0.000 bits
Min-Entropy Memory Access Mutual Information	
Instruction 0x0116DD: `a[pin%2]`	1.000 bits
Other Instructions	0.000 bits
Guessing Entropy	
Instruction 0x0116DD: `a[pin%2]`	64.500 guesses
Other Instructions	128.500 guesses
Minimum Guessing Entropy	
Instruction 0x0116DD: `a[pin%2]`	64.500 guesses
Other Instructions	128.500 guesses

Table 6.3: Summary of the results for the test for an average leakage among all input values.

Example 6.3.3 The second average leakage test case, where a leakage only applies to some of the test cases, uses a variable `i` to make the access dependent from small inputs. In this case, there are actually two input dependent instructions, the access itself and the conditional branch to set `i` in case the current PIN is, regarding its numerical value, smaller than 10.

Listing 6.8: Test code for a computation leaking only on few inputs.

```
1 volatile int a[255];
2 int pin_checker(uint8_t pin) {
3   int i = 0;
4   if (pin < 10) {
5     i = 200;
6   return a[i];
7 }
```

Of $2^8 = 256$ input values in total, there are 9 with a numerical value smaller than 10. So the assumed leakage should be 1 for the min-entropy, as $|Y_i| = 2$, splitting the input into the very small and the larger values.

Hence, the expected value for the Shannon entropy MI should be approximately

$$I_i(X|Y_i) = \frac{246}{256}\log_2\left(\frac{256}{246}\right) + \frac{10}{256}\log_2\left(\frac{256}{10}\right) \approx 0.2379$$

bit.

The results as returned by the *MicroWalk* analysis is listed in Table 6.4

Memory Access Mutual Information	
Instruction 0x0116EA: `a[i]`	0.238 bits
Instruction 0x0116D8: `i = 200`	0.000 bits
Other Instructions	0.000 bits
Min-Entropy Memory Access Mutual Information	
Instruction 0x0116EA: `a[i]`	1.000 bits
Instruction 0x0116D8: `i = 200`	0.000 bits
Other Instructions	0.000 bits
Guessing Entropy	
Instruction 0x0116D8: `i = 200`	5.500 guesses
Instruction 0x0116EA: `a[i]`	118.891 guesses
Other Instructions	128.500 guesses
Minimum Guessing Entropy	
Instruction 0x0116D8: `i = 200`	5.500 guesses
Instruction 0x0116EA: `a[i]`	5.500 guesses
Other Instructions	128.500 guesses

Table 6.4: Summary of the results for the test for an average leakage among few input values.

This example shows, besides the expected outcome for the leaking memory access, another interesting circumstance: There is a second leaking instruction 0x0116D8 that does not appear in the mutual information analyses, but as the lowest guessing entropy result indicating a comparable high vulnerability.

When stepping through the instructions with a debugger, the reason for this behaviour of the analyses becomes clear: The instruction that has a leaking memory access is the assignment within the conditional block. The structure can also be seen in Figure 6.2.

When this instruction is executed, the set of possible input values the secret shrinks to the set of test cases that pass the `< 10` constraint. These are much less than the test case count as a whole, because there are 10 out of 256 matching the constraint. Therefore, the number of guesses required to learn the correct input value is much lower that for guessing the value without observing the telltale instruction execution.

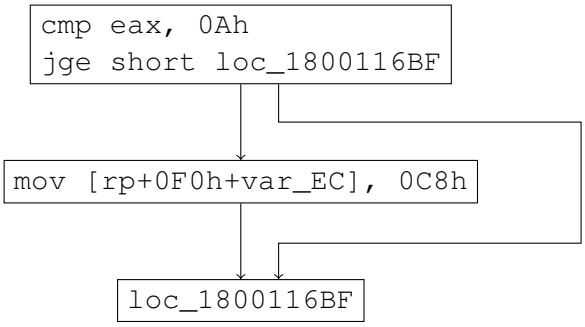

Figure 6.2: The assignment `i = 200;` within the conditional branch dependent on the input.

On the other hand, the mutual information analysis does not detect this leakage source at all. The instruction is listed with a value of 0.000 for both the Shannon based and the min-entropy based MI analysis. This is due to the fact that both quantification techniques result in $\log_2 1 = 0$ for this input. There is no further distinguishability or splitting of equivalence classes among the few values that passed the constraint. Both quantification techniques are not aware of a decrease of the test case count for single instructions.
This is remarkable, because the newly implemented analysis modules using the guessing entropy are able to detect leakages within the single instruction analysis that can not be discovered using the MI analysis only. The guessing entropy adds a viewing angle to the analysis, considering the vulnerability of small sized subsets of the input that are caused by a conditional control flow.

6.3.3 Worst case leakage sample library

Example 6.3.4 The worst case scenario for this consideration is clearly one leaking the input completely. That might happen if there is a memory access that solely depends on the full input.

Listing 6.9: Complete input leakage

```
1 volatile int a[255];
2 int pin_checker(uint8_t pin) {
3   return a[pin];
4 }
```

All quantification techniques are expected to judge this example as gravely leaking code. In this example as shown in Table 6.5 the leakage is indeed 8 bits as expected, since the test cases cover every possible input value. The input is quantified as completely leaked by both MI analyses.

Memory Access Mutual Information	
Instruction 0x0116D3: `a[pin]`	8.000 bits
Other Instructions	0.000 bits
Min-Entropy Memory Access Mutual Information	
Instruction 0x0116D3: `a[pin]`	8.000 bits
Other Instructions	0.000 bits
Guessing Entropy	
Instruction 0x0116D3: `a[pin]`	2.900 guesses
Other Instructions	128.500 guesses
Minimum Guessing Entropy	
Instruction 0x0116D3: `a[pin]`	1.000 guesses
Other Instructions	128.500 guesses

Table 6.5: Summary of the results for the test for a worst case leakage of all inputs.

Furthermore, the guessing entropy shows that there is a seriously high chance that an attacker can guess the PIN within one, respectively two tries. This is a severe result in terms of the confidentiality of the secret inputs which are completely revealed to an attacker who is able to successfully observe the traces.

6.3.4 Test on Microsoft CNG

The Microsoft *Cryptography API: Next Generation* (CNG) is the default cryptography platform used within the Microsoft Windows OS since Windows Vista. The API is provided by the system library `bcrypt.dll`, while the actual implementation is located in the `bcryptprimitives.dll`. The CNG comprises multiple cryptographic algorithms from different algorithmic classes. It provides primitives for random number generation, hashing (e.g. SHA2), signing (e.g. DSA), secret agreements (e.g. Diffie-Hellman) as well as asymmetric (e.g. RSA) and symmetric encryption (e.g. AES) [Mic18].
MicroWalk has been applied to CNG and uncovered leakages within the *ECDSA* and *DSA* as well as AES. For AES there was a 7.96 MI score quantifying the leakage of a subroutine in the T-table implementation variant. The original analysis has taken place for *bcryptprimitives.dll v10.0.17134.1* [WMES18].
For this new test, the *MicroWalk* `PinTracerWrapper.cpp`[8] can be used to again address the T-table based fallback AES implementation of CNG.
The results in Table 6.6 show a large number of instructions, that have a noticeable high leakage quantity. However, the used 5000 random inputs are still a small part of the actual

[8] As it was added to the repository with `commit #e225b5b`: `https://github.com/UzL-ITS/Microwalk/blob/master/PinTracerWrapper/PinTracerWrapper.cpp` [Accessed 28. May 2020]

Memory Access Mutual Information	
Instruction 0x03F6D0: SymCryptAesExpandKeyInternal+24050 and 15 others	12.288 bits
Instruction 0x03F3C1: SymCryptAesExpandKeyInternal+23D41	12.226 bits
Instruction 0x03F47D: SymCryptAesExpandKeyInternal+23DFD and two others	12.219 bits
Min-Entropy Memory Access Mutual Information	
Instruction 0x03F6D0: SymCryptAesExpandKeyInternal+24050 and 15 others	12.288 bits
Instruction 0x03F3C1: SymCryptAesExpandKeyInternal+23D41	12.243 bits
Instruction 0x03F47D: SymCryptAesExpandKeyInternal+23DFD and two others	12.238 bits
Guessing Entropy	
Instruction 0x03F6D0: SymCryptAesExpandKeyInternal+24050 and 15 others	1.000 guesses
Instruction 0x03F3C1: SymCryptAesExpandKeyInternal+23D41	1.031 guesses
Instruction 0x0347D: SymCryptAesExpandKeyInternal+23DFD and three others	1.035 guesses
Minimum Guessing Entropy	
Instruction 0x03F6D0: SymCryptAesExpandKeyInternal+24050 and 35 others	1.000 guesses
Instruction 0x0262D6: SymCryptAesEncryptAsmInternal+346	2.000 guesses

Table 6.6: Summary of the results for the test for *bcryptprimitives.dll v10.0.18362.836*. Some instructions with the found leakage results are listed with their symbolic function name and offset..

set of valid inputs. For these 5000 inputs, as $H(X) = \log_2(5000) \approx 12.288$, the values can be considered completely leaked by different instructions.

The guessing entropy for several leaking instructions is exactly or close to 1, so an attacker requires just a single guess to learn the secret input. This matches the aforementioned observation during the MI analysis that the input values can be considered completely leaked. Even more striking is the result of the minimum guessing entropy, detecting that for a weak input there are multiple instructions leaking all information to the attacker so that she can guess the input correctly within one or respectively two tries.

Concluding, this is the behaviour of a worst case leaking computation as in Example 6.3.4, thus this time not for a specifically designed demonstration example but for the fallback AES encryption of the cryptography platform supplied with every Microsoft Windows system. However, due to the algorithmic characteristics of the T-table implementation of AES, the found leakage was expected [WMES18].

This conclusion also justifies the differences between the quantification techniques to be

rather small for the highly leaking instructions, since the average and the worst case consideration converge for high leakages on all inputs (Example 5.3.2). Still, the guessing entropy of 1 emphasises the full leakage more than the MI result does, as the leakage in bit is returned without a direct comparison to the initial uncertainty for the considered number of inputs.

7 Conclusion

7.1 Summary of the work

In order to enhance the quantification of binary leakages detected by the DBI based framework *MicroWalk*, this thesis has taken account to different quantification approaches from research literature. Namely the min-entropy, the guessing entropy and the minimum guessing entropy have been adapted to the given use case of binaries, that might leak high information to a side-channel attacker observing the binary during runtime.
Three approaches of quantification have been discussed for several example cases of leakage, including computations that are leakage free, that have a small amount of leakage throughout the whole input, and the worst case of computations that leak few or all input values completely.
Especially for the worst case assumption, the min-entropy and minimum guessing entropy have both proven to emphasise the leakage due to the fact, that there are weak inputs revealed via the attacker's side-channel. However, the min-entropy leakage is solely dependent on the number of equivalence classes, the input is split to by the observed traces. It does not further increase for the circumstance, that there might be very small equivalence classes with just few values in them.
This has especially been underlined for the example that the number of different traces Y_i is exactly 2. For the min-entropy, the result is the same if both hashes occurred equally often as it is, if one of the hashes is unique for a special input. To increase the precision of the analysis in this problematic case, the minimum guessing entropy has proven its value, as it depends highly on the size of the smallest equivalence class for uniformly distributed inputs.
Thus, the guessing entropy and the minimum guessing entropy are able to detect a certain type of memory access related leakages (see Example 6.3.3) that can not be discovered using the Shannon entropy or min-entropy MI as implemented in the *MicroWalk* framework.

7.1.1 Future work

All examples, tests and finally the implementation have been conducted for deterministic target programs and uniformly distributed inputs only. This is in accordance to the original scope of the *MicroWalk* framework [WMES18]. However, the bare analysis techniques

are not necessarily restricted to this and might as well be evaluated for non-deterministic programs or non-uniform inputs in the future.
Besides, the usability of the existing analysis could be further improved. The results of the different quantification approaches could be set into a relation to one another to enhance comparability and strengthen the achieved analysis. This might turn out useful, since considering not only one of the quantification techniques has shown to be a significant advance for the analysis results. There will always be use cases for which some of the considered quantification techniques give a rather coarse idea of how severe the threat actually is. To mitigate the impact of such edge cases, it would be helpful, to not regard just one of the quantification techniques but create a bigger picture of the quantity characteristics of an observed leakage.
Furthermore, what also impacts the evaluability of the analysis results is that *MicroWalk* supports no localisation measures of the affected instructions within the control-flow of the binary up to now. Leakages in low level code segments, at the moment identified at instruction level, may have their causes in the high level implementation of a cryptographic algorithm. As well, as an accurate quantification is important for a the software analyst making use of the *MicroWalk* framework, also a precise localisation of leakages is.
Without knowing the actual cause of the leakage, the accurate identification of the present security flaws and fitting countermeasures is more complicated. Also, there is a vast difference between treating a problem in the high level algorithm implementation or within low level functions, both bringing their own risks and challenges when mediating the leakage. Therefore, it might be helpful, to take the control flow into consideration for the analysis results of a future version of *MicroWalk*.

References

[ABB+16] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, 2016. USENIX Association.

[ABH18] José Amigó, Sámuel Balogh, and Sergio Hernández. A Brief Review of Generalized Entropies. *Entropy*, 20:813, 10 2018.

[AKM+15] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On Subnormal Floating Point and Abnormal Timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639, 2015.

[And19] Dennis Andriesse. *Practical Binary Analysis*, chapter 9, pages 224–264. No Starch Press, 2019.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[BK15] Elaine B. Barker and John M. Kelsey. SP 800-90A Revision 1. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, USA, 2015.

[BTBT10] P. Bromiley, N. Thacker, and Evelina Bouhova-Thacker. Shannon Entropy, Renyi Entropy, and Information. 07 2010.

[Cac97] Christian Cachin. Entropy measures and unconditional security in cryptography. 1997.

[Cen19] National Cyber Security Center. Most hacked passwords revealed as UK cyber survey exposes gaps in online security. `https://www.ncsc.gov.uk/news/most-hacked-passwords-revealed-as-uk-cyber-survey-exposes-gaps-in-online-security`, April 2019. [Online; accessed 02-May-2020].

[DRS04] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. In Christian

Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 523–540, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Gen19] David Gens. *OS-level Attacks and Defenses: from Software to Hardware-based Exploits*. PhD thesis, Technische Universität, Darmstadt, 2019.

[Gra90] Robert M. Gray. *Entropy and Information Theory*. Springer Publishing Company, Incorporated, 1st edition, 1990.

[GYCH16] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8:1–27, 10 2016.

[Hu91] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 8–20, 1991.

[Int12] Intel. Pin - A binary instrumentation tool. `https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html`, 2012. [Online; accessed 23-May-2020].

[KB07] Boris Köpf and David Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 286–296, New York, NY, USA, 2007. Association for Computing Machinery.

[Ker83] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, pages 161–191, 1883.

[LM99] Yau-Tsun Steven Li and Sharad Malik. *Microarchitecture Modeling*, pages 53–102. Springer US, Boston, MA, 1999.

[Mas94] J. L. Massey. Guessing and entropy. In *Proceedings of 1994 IEEE International Symposium on Information Theory*, page 204, 1994.

[McL] Tim McLean. A beginner's guide to constant-time cryptography. `https://www.chosenplaintext.ca/articles/beginners-guide-constant-time-cryptography.html`. [Online; accessed 28-April-2020].

[MDS12] R. Martin, J. Demme, and S. Sethumadhavan. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel

attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129, 2012.

[Mic18] Microsoft. Cryptography API: Next Generation. `https://docs.microsoft.com/en-us/windows/win32/seccng/cng-portal`, 2018. [Online; accessed 25-May-2020].

[MKW18] Kai Mindermann, Philipp Keck, and Stefan Wagner. How Usable Are Rust Cryptography APIs? *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Jul 2018.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, page 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[PGBY16] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "make sure dsa signing exponentiations really are constant-time". In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1639–1650, New York, NY, USA, 2016. Association for Computing Machinery.

[Por] Thomas Pornin. BearSSL - A smaller SSL/TLS library. `https://www.bearssl.org/constanttime.html`. [Online; accessed 10-May-2020].

[RBV17] O. Reparaz, J. Balasch, and I. Verbauwhede. Dude, is my code constant time? In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1697–1702, 2017.

[Ré61] Alfréd Rényi. On Measures of Entropy and Information. In *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*, pages 547–561, Berkeley, Calif., 1961. University of California Press.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27(3):379–423, 1948.

[SM06] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[Smi09] Geoffrey Smith. On the Foundations of Quantitative Information Flow. In Luca de Alfaro, editor, *Foundations of Software Science and Computational Structures*, pages 288–302, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[WL07] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 494–505, New York, NY, USA, 2007. Association for Computing Machinery.

[WMES18] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. *CoRR*, abs/1808.05575, 2018.

[YB14] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[ZBPB17] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1789–1806, New York, NY, USA, 2017. Association for Computing Machinery.