Thomas Wedenig, BSc

Exact Inference in Side-Channel Attacks using Tractable Probabilistic Models

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.-Prof. Dipl.-Ing. Dr. techn. Robert Peharz

Institute of Theoretical Computer Science

Graz, October 2023

Abstract

Side-Channel Attacks (SCAs) have demonstrated remarkable potential to attack cryptographic systems and compromise their security by exploiting unintentional information leakage through various physical channels. Using different methods, we can encode the information present in the leakage as probability distributions over intermediate values (input distributions). State-of-the-art methods like the *Soft Analytical Side-Channel Attack* (SASCA) combine these input distributions with knowledge about the cryptographic algorithm by representing the system under attack as a factor graph, in which an attacker performs probabilistic inference to reason about likely key hypotheses. In many real-world scenarios, SASCA must resort to loopy belief propagation, an approximate inference algorithm that does not guarantee convergence or accurate estimates. In this work, we study the applicability of a class of tractable probabilistic models, namely *Probabilistic Circuits* (PCs), to SCA inference problems. Using techniques from *Knowledge Compilation*, we find that we can tractably represent parts of the *Advanced Encryption Standard* (AES) algorithm as a PC and use this fact to replace loopy parts of a factor graph with a PC. Our experiments show that, under some assumptions, we can utilize PCs to tractably perform *exact* inference using a sparse approximation of input distributions, which significantly outperforms the SASCA in a real-world attack scenario. We also provide an information-theoretic upper bound on the approximation error of the input distributions. Moreover, we experiment with *Deep Learning* based SCAs and show the close relationship between our proposed inference method and recent developments in the field of *neuro-symbolic learning*. In this context, we show that we can also learn input distributions that are amenable to exact inference without the need for sparse approximations, albeit at the cost of expressiveness.

Kurzfassung

Seitenkanalangriffe (SCAs) haben bemerkenswertes Potenzial gezeigt, kryptografische Systeme anzugreifen und ihre Sicherheit durch Ausnutzung unbeabsichtigter Informationslecks über verschiedene physische Kanäle zu kompromittieren. Mit verschiedenen Methoden können wir die in dem Leck vorhandene Information als Wahrscheinlichkeitsverteilungen über Zwischenwerte (Eingangsverteilungen) kodieren. Modernste Methoden wie *Soft Analytical Side-Channel Attacks* (SASCAs) kombinieren diese Eingangsverteilungen mit Wissen über den kryptografischen Algorithmus, indem sie das angegriffene System als Faktorgraph darstellen, in dem ein Angreifer probabilistische Inferenz durchführt, um wahrscheinliche Schlüsselhypothesen zu generieren. In vielen realen Szenarien muss SASCA auf *Loopy Belief Propagation* zurückgreifen, einem ungefähren Inferenzalgorithmus, der weder Konvergenz noch genaue Schätzungen garantiert. In dieser Arbeit untersuchen wir die Anwendbarkeit einer Klasse von probabilistischen Modellen, den sogenannten *Probabilistic Circuits* (PCs), für Inferenzprobleme in SCAs. Wir stellen fest, dass wir mithilfe von *Knowledge Compilern* Teile des *Advanced Encryption Standard* (AES) Algorithmus als PC darstellen können und verwenden diese Tatsache, um zyklische Teile eines Faktorgraphen durch einen PC zu ersetzen. Unsere Experimente zeigen, dass wir unter bestimmten Annahmen PCs verwenden können, um *exakte* Inferenz mit einer dünnbesetzten Approximation der Eingabeverteilungen durchzuführen, was SASCA in einem realen Angriffsszenario deutlich übertrifft. Wir zeigen außerdem eine informationstheoretische Obergrenze für den Annäherungsfehler der Eingabeverteilungen. Darüber hinaus experimentieren wir mit Deep Learning-basierten SCAs und zeigen die enge Beziehung zwischen unserer vorgeschlagenen Inferenzmethode und jüngsten Entwicklungen im Bereich des *neurosymbolischen* Lernens. In diesem Zusammenhang zeigen wir, dass wir auch Eingabevertellungen erlernen können, die für exakte Inferenz geeignet sind, ohne dass Annäherungen der Verteilungen erforderlich sind — wenn auch auf Kosten der Ausdrucksfähigkeit dieser Verteilungen.

Acknowledgements

First and foremost, I would like to thank my advisor Robert Peharz for his guidance and support throughout the course of this research. Your positive attitude has been a constant source of motivation and your expertise and feedback have been truly invaluable for this thesis. Thank you for the countless hours we have spent pondering about the challenges I have faced during my research.

I would also like to express my gratitude to all staff members of the Institute of Theoretical Computer Science for providing such a pleasant research environment.

Furthermore, many thanks to Rishub Nagpal, Gaëtan Cassiers, and Christian Toth for the stimulating discussions and for the diverse perspectives they provided.

To my friends and colleagues, thank you for all the thought-provoking dialogues and the wonderful moments we have shared.

I owe special thanks to my family, who have continuously supported me in every regard and have always given me the freedom to follow my passions.

Finally, I want to thank my wonderful girlfriend Katharina, who has – not once – complained about me boring her with the technical details of this thesis.

Contents

1 Introduction

Side-channel attacks (SCAs) allow attackers to learn about secret material used in a cryptographic computation by observing implementation artifacts, such as timing information [**timing_kocher**], electromagnetic leaks [**em_gandolfi**], and power consumption [**dpa_kocher**]. This auxiliary information is commonly called *leakage*. Most contemporary attacks use these leakages to reason about the state of intermediate values in an algorithm in a probabilistic manner, i.e., they output probability mass functions over possible states of intermediate variables. For example, given information about the power consumption of a device that performed encryption using the *Advanced Encryption Standard* (AES), an attacker can use existing methods such as *Template Attacks* [**template_attacks**] to produce probability distributions that capture beliefs of the states of particular variables used in the AES, given the leakage (e.g., over individual bytes of the intermediate variables). Given these distributions and the knowledge of the software implementation of the algorithm, the goal of the attacker is to infer the posterior distribution of the secret key. While probability theory offers rigorous operations on how to perform this inference task (such as marginalization, conditioning, Bayes' rule, and maximum a posteriori), the high time of a naïve computation usually makes it infeasible for practical attacks. As a remedy, state-of-the-art attacks perform *approximate* probabilistic inference: For example, *Soft Analytical Side-Channel Attacks* (SASCAs) [**sasca**] leverage loopy belief propagation, a method for approximate probabilistic inference that neither guarantees accurate estimates nor assures convergence and has limited theoretical underpinnings [**bp**].

Meanwhile, advancements in the fields of knowledge compilation and tractable probabilistic modeling have given rise to methods that can effectively combine both symbolic knowledge (the software implementation of the algorithm under attack) and probabilistic modeling (mass functions over states of variables) within a structure called a *Probabilistic Circuit* (PC) [**sdd**, **psdd**, **dynamic_min_choi**]. PCs can encode expressive probability distributions and, with certain structural constraints, are able to answer inference queries such as marginalization and maximum a posteriori queries in linear time in the size of the circuit [**psdd**].

In this work, we primarily ask two questions: Using a real-world dataset of power traces, can (1) we perform *exact* probabilistic inference with the help of PCs, and if so, (2) does this improve the effectiveness of the side-channel attack when compared to approximate methods like SASCA?

In a nutshell, we find that we can utilize PCs to tractably perform exact inference using sparse approximations of the probability mass functions of intermediate variables (under some *low entropy* assumptions) and show that this method substantially outperforms state-of-the-art approaches that use loopy belief propagation while empirically demonstrating competitive runtimes. Furthermore, we show that, if the distributions of intermediate variables are on the scope of individual *bits*, PCs are able to tractably perform exact inference — even *without* approximation techniques.

While not being the central focus of this work, we also discuss approaches that combine *Deep Learning* based side-channel attacks with our proposed inference pipeline and show interesting connections to the field of *neuro-symbolic learning*.

Throughout most of this work, we will slightly abuse the probability theoretic notation for conciseness and will mostly not differentiate between variables and the values they can take on. For example, instead of writing $p(X = x)$ to denote the probability that random variable X assumes the value x, we simply write $p(x)$ where the random variable follows from the context in order to keep the notation uncluttered.

2 Background

2.1 Nomenclature and Notation

In this work, we will attack parts of the *Advanced Encryption Standard* (AES) [**aes**], a popular block cipher that takes an n-bit *plaintext* and a (secret) n-bit *key* as input, and produces an n-bit *ciphertext* as output. In our experiments, we attack AES-128, i.e., $n = 128$. In distinction to *input variables* (key and plaintext) and *output variables* (ciphertext), the algorithm computes several *intermediate variables*, which are byproducts of the computation of the ciphertext. In particular, we will study subroutines of AES, such as the MixColumn function. When analyzing subroutines, we will reason about *their* inputs, intermediate variables and outputs as well. Hence, the notion of inputs, outputs and intermediates might change depending on the context.

Further, we write $\mathbb{F}_k$ to denote the finite field (Galois field) of characteristic k and thus, write $\mathbb{F}_2^n$ to denote the set of n-bit binary vectors. We also do not explicitly differentiate between an integer $z \in \mathbb{Z}$ and its binary representation.

2.2 Advanced Encryption Standard

The *Advanced Encryption Standard* (AES) describes a symmetric-key encryption algorithm. In the following explanation, we focus on AES-128 for clarity but note that the results of this work can easily be extended to AES-192 or AES-256. When executing AES, we iteratively compute a sequence of routines multiple times (*rounds*). In particular, AES-128 computes 10 such rounds. Moreover, before starting the first round, AES uses a so-called *key schedule* algorithm to expand the 16-byte key $\mathbf{k}$ to obtain 11 *round keys* (each 16-byte) $\mathbf{k}^0, \ldots, \mathbf{k}^{10}$ where $\mathbf{k}^0 = \mathbf{k}$. Each round[1] consists of executing the functions SubBytes, ShiftRows, MixColumns, and AddRoundKey (in this order). Every function acts on a 4×4-byte state matrix $\mathbf{M} \in \mathbb{F}_{256}^{4\times4}$ and outputs a matrix $\mathbf{M}' \in \mathbb{F}_{256}^{4\times4}$, which is then considered to be the input to the next function. We denote bytes in $\mathbf{M}$ as $m_{i,j}$ and bytes in $\mathbf{M}'$ as $m'_{i,j}$. Figure **??** illustrates the functionality of the four functions.

[1]Except for the last round, where no MixColumns is performed.

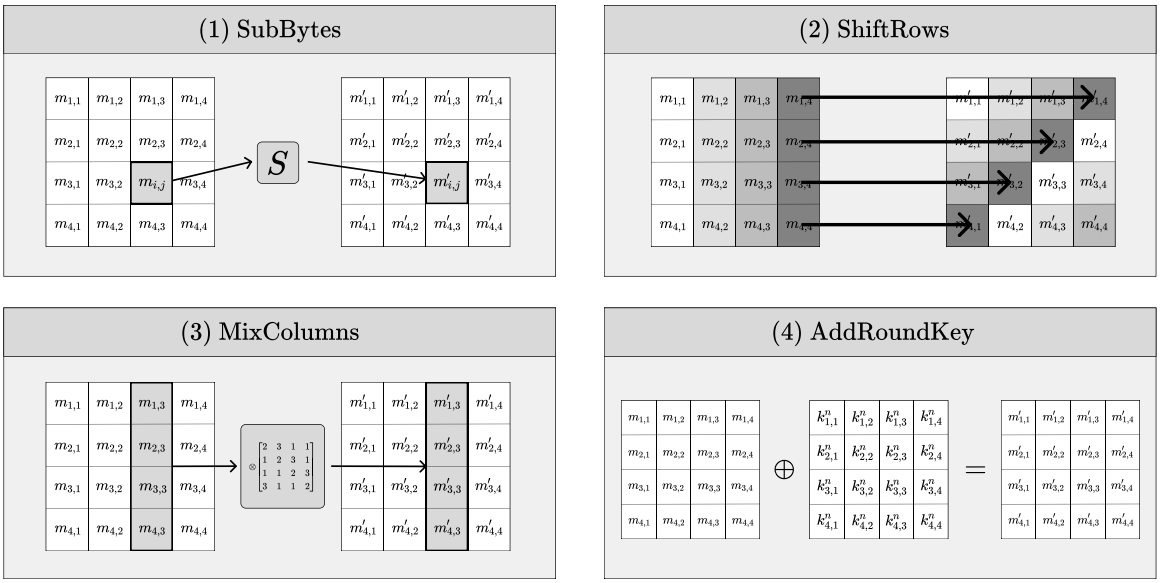


Figure 2.1: A typical round of AES, consisting of SubBytes, ShiftRows, MixColumns, and AddRoundKey.

SubBytes takes every byte $m_{i,j}$ in $\mathbf{M}$ and computes $m'_{i,j} = S(m_{i,j})$, where $S : \mathbb{F}_{256} \to \mathbb{F}_{256}$ is a bijective, non-linear lookup table called the *S-Box* (*Substitution Box*). ShiftRows takes every row in $\mathbf{M}$ and shifts the elements to the left by one, i.e., computes $m'_{i,j} = m_{i,j+i \bmod 4}$ where mod denotes the modulo operator. MixColumns takes every column $\mathbf{m}_i$ in $\mathbf{M}$ and multiplies the column vector with a pre-defined matrix, i.e., the output column $\mathbf{m}'_i$ is given by

$$\mathbf{m}'_i = \mathbf{P} \otimes \mathbf{m}_i \quad \text{with } \mathbf{P} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \tag{2.1}$$

where $\otimes$ denotes a matrix-vector product in the Galois Field $\mathbb{F}_{256}$. As shown later, an implementation of MixColumns only needs xor operations and a lookup table for multiplications by 2 in $\mathbb{F}_{256}$ (called xtime). Finally, AddRoundKey takes each byte $m_{i,j}$ in $\mathbf{M}$ and computes $m'_{i,j} = k^n_{i,j} \oplus m_{i,j}$ where $k^n_{i,j}$ denotes the corresponding byte in the n-th round key ($n \in \{1, \ldots, 10\}$). Moreover, before executing the first round, we fill $\mathbf{M}$ with the 16 plaintext bytes $p_1, \ldots, p_{16}$ and perform AddRoundKey with $\mathbf{k}^0$, which is just the key $\mathbf{k}$. After the final AES round, we can collect the ciphertext in the state matrix $\mathbf{M}$.

2.3 Side-Channel Attacks

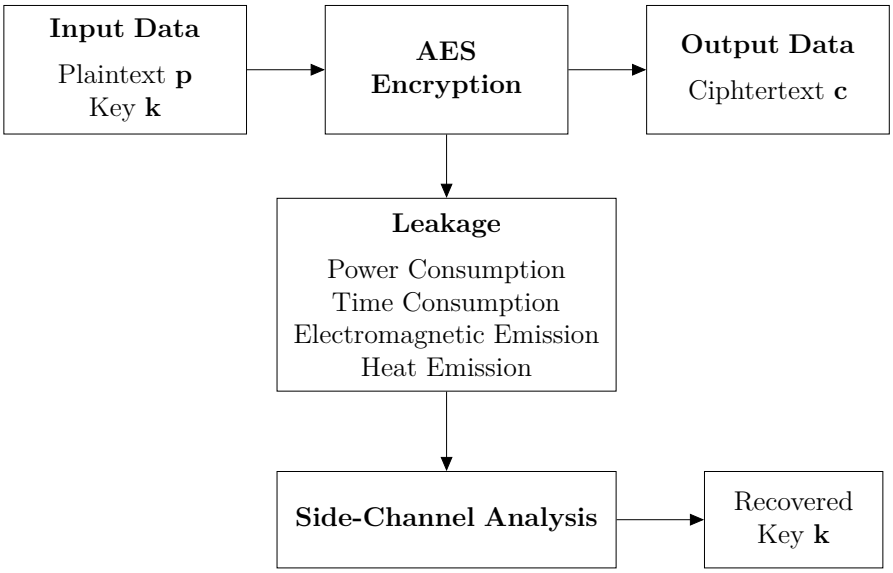


Figure 2.2: Schematic overview of a side-channel attack on the AES encryption function.

Although a cryptographic algorithm may be secure conceptually, its implementation may be not: Side-channel attacks exploit the fact that physical implementations of an algorithm unintentionally leak information about the processed data. For example, consider an encryption algorithm, that takes the plaintext and the cryptographic key as input and outputs the ciphertext. In practice, the implementation of such an algorithm also produces artifacts information that contains information about the input data, particularly the secret key [**scas**]. For example, since the power usage of CMOS transistors depends on the switching activity during the computation, the power consumption of the device performing the encryption is not only dependent on the algorithm but also on the data processed [**intro_sca**]. It has been frequently shown that side-channel attacks are more efficient than the best-known cryptanalytic attacks, which consider the system under attack as an idealized, mathematical model [**intro_sca**]. While adversaries can exploit various side channels such as timing of operations [**timing_kocher**] and electromagnetic emanation [**em_gandolfi**], this work focuses on information about power consumption. A schematic overview of a side channel attack is shown in Figure **??**.

2.3.1 Template Attacks

Consider an adversary attacking an implementation of a cryptographic algorithm running on a target device (e.g., a smart card) and assume that the attacker has access

to a *clone* device that is identical in hardware to the target device. If the attacker has full control over the clone device, they can arbitrarily choose the input to the algorithm (e.g., key and plaintext in a block cipher) and record the side-channel leakage of her choice (e.g., power consumption of the clone device). In this *profiling* phase, they effectively build a dataset of input-leakage pairs, denoted as *profiling dataset*. Since the input and the algorithm are fully known, we can reconstruct any intermediate value and the algorithm's output (assuming a deterministic algorithm).

Due to the leakages being noisy, there exists no deterministic map between processed data and leakage. A natural choice is to build a (generative) probabilistic model of this relationship: More formally, let $\boldsymbol{\ell} \in \mathbb{R}^d$ denote a leakage (e.g., power trace), let $v \in \{0, \ldots, 255\}$ be a byte-valued variable[2] used in the algorithm (e.g., an input, intermediate variable, or output), and let $\mathcal{D} = \{(\boldsymbol{\ell}^{(i)}, v^{(i)})\}_{i=1}^{n}$ denote the profiling dataset. A common choice is to model the likelihood as a multivariate Gaussian, i.e.,

$$p(\boldsymbol{\ell} \mid v) = \mathcal{N}(\boldsymbol{\mu}_v, \boldsymbol{\Sigma}_v) = \frac{1}{\sqrt{(2\pi)^d \cdot |\boldsymbol{\Sigma}_v|}} \cdot \exp\left(-\frac{1}{2}(\boldsymbol{\ell} - \boldsymbol{\mu}_v)^T \boldsymbol{\Sigma}_v^{-1} (\boldsymbol{\ell} - \boldsymbol{\mu}_v)\right) \tag{2.2}$$

where $\boldsymbol{\mu}_v$ is the mean of the leakages for some value v, which we estimate using the *Maximum Likelihood* estimator $\tilde{\boldsymbol{\mu}}_v = \frac{1}{|\mathcal{D}_v|} \sum_{\boldsymbol{\ell}_v \in \mathcal{D}_v} \boldsymbol{\ell}_v$ where $\mathcal{D}_v \subseteq \mathcal{D}$ refers to the subset of the dataset where the variable of interest assumed value v, and $\boldsymbol{\Sigma}_v$ is the positive semi-definite covariance matrix, estimated as

$$\tilde{\boldsymbol{\Sigma}}_v = \frac{1}{|\mathcal{D}_v| - 1} \sum_{\boldsymbol{\ell}_v \in \mathcal{D}_v} (\boldsymbol{\ell}_v - \tilde{\boldsymbol{\mu}}_v) (\boldsymbol{\ell}_v - \tilde{\boldsymbol{\mu}}_v)^T \tag{2.3}$$

Since the leakages $\boldsymbol{\ell}$ are high-dimensional in practice (e.g., $d = 10^5$), one might first apply one or more dimensionality reduction techniques as pre-processing steps. For example, Bronchain et al. [**5min**] use a combination of signal-to-noise ratio estimation to find points of interest in $\boldsymbol{\ell}$ and then perform *Linear Discriminant Analysis* (LDA) to linearly project the compressed leakage into a lower dimensional subspace [**5min**].

Consequently, the adversary constructs 256 Gaussians (*templates*) based on the profiling dataset, one for each unique value v. Since the attacker is usually interested in multiple intermediate variables $v_1, \ldots, v_k$, they repeat this modeling process k times.

During the attack on the target device, the attacker only has access to the leakage and wishes to infer the values of the processed data. We can turn the generative model in Equation **??** (likelihood) into a predictive model by utilizing Bayes' law:

$$p(v \mid \boldsymbol{\ell}) = \frac{p(\boldsymbol{\ell} \mid v) p(v)}{\sum_{v'=0}^{255} p(\boldsymbol{\ell} \mid v') p(v')} \tag{2.4}$$

[2] In this thesis, we mainly focus on distributions over *bytes*. However, we discuss the applicability of our approach to other distribution scopes in Section **??**.

Typically, the prior $p(v)$ is chosen to be uniform, which simplifies Equation **??** to

$$p(v \mid \boldsymbol{\ell}) = \frac{p(\boldsymbol{\ell} \mid v)}{\sum_{v'=0}^{255} p(\boldsymbol{\ell} \mid v')} \tag{2.5}$$

We also note that while modeling distribution over *byte-valued* variables v is a common choice, other works investigate extending the scope of v to multiple bytes (e.g., 2 bytes or 4 bytes) [**32bit**], or to shrink the scope of v to bit-level [**breaking_free**]. Moreover, instead of directly predicting the *value* of a particular variable $p(v \mid \boldsymbol{\ell})$, we can generalize this setup by choosing to predict a *function* of the value $p(f(v) \mid \boldsymbol{\ell})$. $f(v)$ is called a *leakage model* and popular choices include the *Hamming weight*, i.e., the number of bits in v that are set to 1, and the identity function $f(v) = v$.

After obtaining posterior distributions for all variables of interest $p(f(v_1) \mid \boldsymbol{\ell}), \ldots, p(f(v_k) \mid \boldsymbol{\ell})$, the attacker aims to aggregate their beliefs about $v_1, \ldots, v_k$ to obtain distributions over the *key bytes*. Intuitively, beliefs about intermediate variables depending on the key bytes should be propagated and combined into beliefs about the key itself.

2.3.2 Algebraic Side-Channel Attacks

In order to infer beliefs about the key from beliefs about intermediate values, knowledge about the cryptographic algorithm must be combined with the distributions $p(f(v_1) \mid \boldsymbol{\ell}), \ldots, p(f(v_k) \mid \boldsymbol{\ell})$.

One possible way to achieve this is to first transform the algorithm into a boolean representation [**asca**]. Let $\mathcal{A}$ be the cryptographic algorithm and $\mathbf{v} = (v_1, \ldots, v_k)^T$ be the collection of intermediate variables used in $\mathcal{A}$. Then,

$$\mathcal{A}_B(\mathbf{v}) = \begin{cases} 1 & \text{if } \mathbf{v} \text{ is consistent with } \mathcal{A} \\ 0 & \text{else} \end{cases} \tag{2.6}$$

is called the *boolean representation* of $\mathcal{A}$.

If an attacker knew certain intermediate variables with certainty, they could represent both $\mathcal{A}_B$ and their knowledge about intermediate variables in conjunctive normal form (CNF) and use a SAT solver to deduce the key. This approach is known as an *Algebraic Side-Channel Attack* (ASCA) [**asca**].

For example, assume that they first use a template attack to obtain $p(f(v_1) \mid \boldsymbol{\ell}), \ldots, p(f(v_k) \mid \boldsymbol{\ell})$ with $f(v)$ being the Hamming weight of v. Also assume that $p(f(v_1) = 1 \mid \boldsymbol{\ell}) = 1$, i.e., they know with certainty that exactly 1 bit of the byte v_1 is

set to 1. If $v_1^{(1)}, \ldots, v_1^{(8)}$ denote the individual bits of v_1, they can easily encode this leakage information into a boolean formula as follows:

$$\left(v_1^{(1)} \wedge \bigwedge_{i \in \{2,\ldots,8\}} \neg v_1^{(i)}\right) \vee \left(v_1^{(2)} \wedge \bigwedge_{i \in \{1,3,4,\ldots,8\}} \neg v_1^{(i)}\right) \vee \cdots \vee \left(v_1^{(8)} \wedge \bigwedge_{i \in \{1,\ldots,7\}} \neg v_1^{(i)}\right) \quad (2.7)$$

This can then be transformed into a CNF and added to the set of clauses that represent $\mathcal{A}_B$. If the adversary also knows the Hamming weight of all other intermediates with certainty, they can repeat this process and use a SAT solver to find the only valid key assignment in the final CNF.

In practice, however, we cannot extract information from the leakage about all intermediates with absolute certainty. As a remedy, the ASCA first picks the most likely Hamming weight for all intermediates and runs the attack, then proceeds to use the second most likely combination of Hamming weights, and so on [**asca**]. If some combination of Hamming weights is inconsistent with $\mathcal{A}_B$, the resulting CNF will be unsatisfiable.

Clearly, the main downside of this attack is that the attacker has to collapse their probabilistic model about intermediate values to *hard*, deterministic assumptions to be able to provide leakage information to a SAT solver. This leads to the fact that the ASCA is highly sensitive to errors - Renault et al. [**asca_aes**] were only able to solve problems with an error rate under 1%, which is of limited use in practical scenarios [**tasca**].

Consequently, attacks that are less sensitive to noise have been developed: Instead of directly using an error-intolerant SAT solver, *Tolerant Algebraic Side-Channel Analysis* (TASCA) leverages Pseudo-Boolean optimization, a special case of integer programming that solves a constraint optimization problem [**tasca**]. The authors introduce error variables in their set of equations (which assumes a fixed upper bound on the measurement error) and instruct the solver to minimize the total number of error variables [**tasca**]. However, this approach is still inefficient in exploiting soft information and fails if the measurement error is larger than the assumed upper bound [**sasca**].

2.3.3 Soft Analytical Side-Channel Attacks

Instead of using SAT solvers or optimizers, a *Soft Analytical Side-Channel Attack* (SASCA) represents a cryptographic algorithm as a *factor graph*, a type of probabilistic graphical model [**sasca**]. A factor graph is a bipartite graph, consisting of two types of nodes - namely *variable nodes* and *factor nodes*. A factor node represents a function and all connected variable nodes represent arguments to this function. An example is illustrated in Figure **??**.

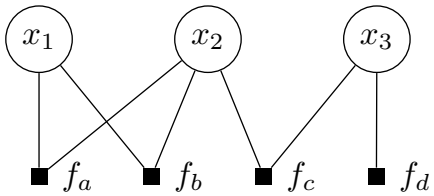


Figure 2.3: A factor graph representing $f(x_1, x_2, x_3) = f_a(x_1, x_2)f_b(x_1, x_2)f_c(x_2, x_3)f_d(x_3)$.

In general, a factor graph which has factor nodes $\mathcal{F} = \{f_1, \ldots, f_n\}$ and variable nodes $\mathcal{V} = \{x_1, \ldots, x_m\}$ specifies a function $f(x_1, \ldots, x_m) = \prod_{i=1}^{n} f_i(\mathrm{ne}(f_i))$ where $\mathrm{ne}(f)$ denotes the set of neighbours of factor f (which are variable nodes). Often, factor graphs are used to represent probability mass functions or probability density functions [**fg_loeliger**]. The SASCA models every function used in the cryptographic implementation (e.g., XOR, SBOX, XTIME) as a factor with binary output [**sasca**]. For example,

$$\textsc{xor}(a, b, c) = \begin{cases} 1 & \text{if } a \oplus b = c \\ 0 & \text{else} \end{cases} \qquad \textsc{sbox}(a, b) = \begin{cases} 1 & \text{if } S(a) = b \\ 0 & \text{else} \end{cases} \tag{2.8}$$

where a, b, c are byte-valued variables and S is an SBOX function, as found in the AES.

All variables are modeled as variable nodes and, additionally to being connected to the function nodes as described in the algorithm, each variable is connected to a factor representing the corresponding posterior distribution obtained e.g. via a template attack. For example, consider the following excerpt of AES:

$$y = k \oplus p, \quad x = S(y)$$

where k is a key byte, p is a *known* plaintext byte, and S as a bijection that denotes the AES S-box. The corresponding factor graph is shown in Figure **??** and effectively represents an unnormalized joint probability distribution over unobserved variables, i.e.,

$$\tilde{p}(k, y, x) = p(k|\boldsymbol{\ell}) \cdot p(y|\boldsymbol{\ell}) \cdot p(x|\boldsymbol{\ell}) \cdot \textsc{xor}(k, p, y) \cdot \textsc{sbox}(y, x)$$

The normalized version reads

$$p(k, y, x) = \frac{1}{Z}\, \tilde{p}(k, y, x) \quad \text{with} \quad Z = \sum_k \sum_y \sum_x \tilde{p}(k, y, x).$$

The goal is to obtain the *marginal* of the key, i.e., compute

$$p(k) = \sum_y \sum_x p(k, y, x).$$

A SASCA computes this quantity using *Belief Propagation* (BP), an iterative message-passing algorithm that is detailed in Section **??**. In a nutshell, the algorithm is guaranteed to output the exact marginal if the factor graph is a tree. While BP can also be applied to *estimate* marginals in cyclic graphs (*loopy BP*), it is neither guaranteed that the algorithm converges nor that the output is an accurate estimate [**bp**].

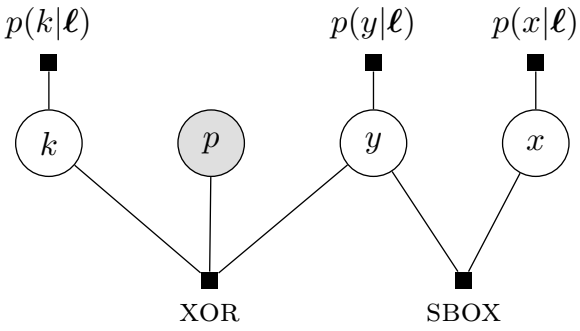


Figure 2.4: A factor graph that incorporates both the posterior distributions over unobserved variables, as well as the algorithmic relationship between variables. Note that the plaintext p is assumed to be known.

2.3.4 Belief Propagation

The following exposition closely follows the work of Christopher Bishop [**bishop**]. Assume we are given a factor graph that is connected (i.e., there exists a path between any two nodes), and contains no cycles (i.e., there exists no sequence of distinct edges that form a path that starts and ends at the same). This factor graph contains a collection of variables $\mathbf{x}$ and we wish to compute the marginal

$$p(x) = \sum_{\mathbf{x} \setminus x} p(\mathbf{x}) \tag{2.9}$$

for a particluar x, where $\mathbf{x} \setminus x$ denotes all variables in $\mathbf{x}$, except x.

As illustrated in Figure **??**, the variable node x is connected to one or more factor nodes, each of which can be viewed as the root of a particular subtree. Let f_s be any neighbour[3] of x and let $F_s(x, \mathbf{x}_s)$ be the subtree connected to f_s, where $\mathbf{x}_s$ is the

[3] f_s is guaranteed to be a factor node since the factor graph is bipartite.

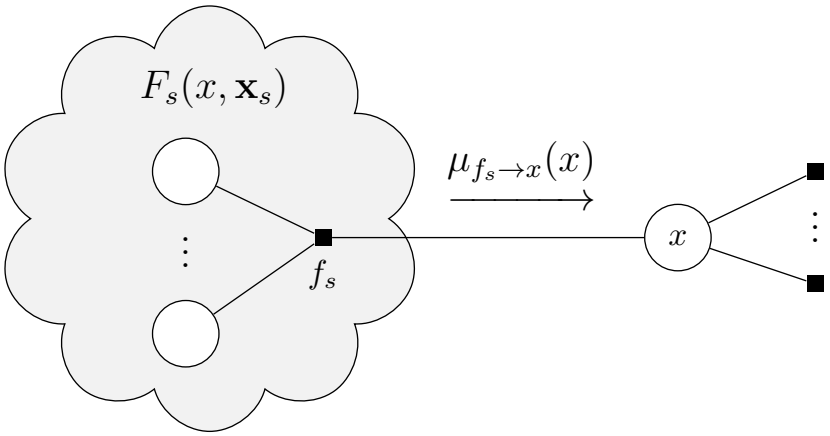


Figure 2.5: Subtree F_s connected to variable node x.

subset of variables in $\mathbf{x}$ that can be found in subtree F_s. With $\mathrm{ne}(x)$ denoting the set of factor nodes that are neighbours of x, we can write

$$p(\mathbf{x}) = \prod_{s \in \mathrm{ne}(x)} F_s(x, \mathbf{x}_s) \tag{2.10}$$

which simply expresses the factorization as a product of subtrees (which are themselves products). Thus,

$$p(x) = \sum_{\mathbf{x} \setminus x} p(\mathbf{x}) = \sum_{\mathbf{x} \setminus x} \prod_{s \in \mathrm{ne}(x)} F_s(x, \mathbf{x}_s) \tag{2.11}$$

$$= \prod_{s \in \mathrm{ne}(x)} \underbrace{\sum_{\mathbf{x}_s} F_s(x, \mathbf{x}_s)}_{\mu_{f_s \to x}(x)} \tag{2.12}$$

where we define $\mu_{f_s \to x}(x) = \sum_{\mathbf{x}_s} F_s(x, \mathbf{x}_s)$ to be a *message* from f_s to x. Note that we are only allowed to pull the sum into the product since for any two $s, s' \in \mathrm{ne}(x)$, the sets $\mathbf{x}_s$ and $\mathbf{x}_{s'}$ are disjoint (since the factor graph is a tree).

As shown in Figure **??**, $F_s(x, \mathbf{x}_s)$ can be decomposed similarly by looking at all subtrees in F_s that are connected to f_s:

$$F_s(x, \mathbf{x}_s) = f_s(x, x_1, \ldots, x_m) G_1(x_1, \mathbf{x}_{s1}) \cdots G_m(x_m, \mathbf{x}_{sm}) \tag{2.13}$$

where $\mathbf{x}_{si}$ denotes the set of variables in subtree G_i, apart from x_i, the direct neighbour

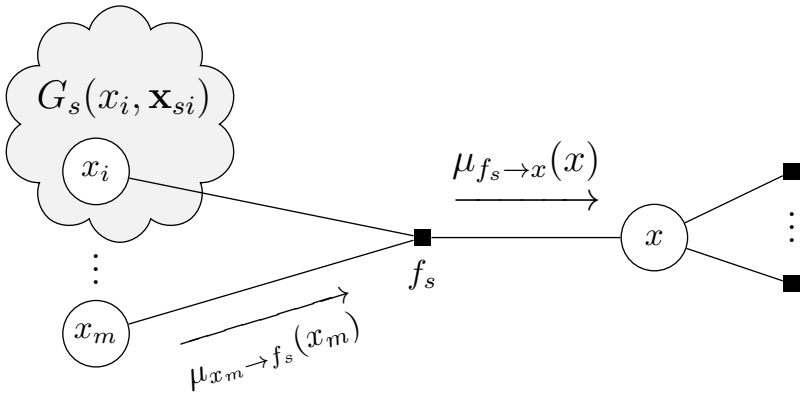


Figure 2.6: Within F_s, we have subtrees G_s connected to factor node f_s.

of f_s. Hence,

$$\mu_{f_s \to x}(x) = \sum_{\mathbf{x}_s} \left(f_s(x, x_1, \ldots, x_m) \prod_{i \in \mathrm{ne}(f_s) \setminus x} G_i(x_i, \mathbf{x}_{si}) \right) \tag{2.14}$$

$$= \sum_{x_1, \ldots, x_m} f_s(x, x_1, \ldots, x_m) \prod_{i \in \mathrm{ne}(f_s) \setminus x} \underbrace{\left(\sum_{\mathbf{x}_{si}} G_i(x_i, \mathbf{x}_{si}) \right)}_{\mu_{x_i \to f_s}(x_i)} \tag{2.15}$$

where $\mu_{x_i \to f_s}(x_i)$ can be viewed as a message from variable node x_i to factor node f_s.

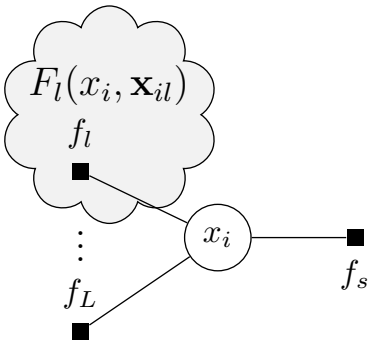


Figure 2.7: Within G_s, we again have subtrees F_l connected to variable node x_i.

As Figure **??** shows, we can again decompose the graph $G_i(x_i, \mathbf{x}_{si})$ into subgraphs in G_i that are connected to x_i:

$$G_i(x_i, \mathbf{x}_{si}) = \prod_{l \in \mathrm{ne}(x_i) \setminus f_s} F_l(x_i, \mathbf{x}_{il}) \tag{2.16}$$

and therefore, the variable-to-factor message can be computed as

$$\mu_{x_i \to f_s}(x_i) = \sum_{\mathbf{x}_{si}} G_i(x_i, \mathbf{x}_{si}) = \sum_{\mathbf{x}_{si}} \left(\prod_{l \in \text{ne}(x_i) \setminus f_s} F_l(x_i, \mathbf{x}_{il}) \right) \tag{2.17}$$

$$= \prod_{l \in \text{ne}(x_i) \setminus f_s} \underbrace{\left(\sum_{\mathbf{x}_{il}} F_l(x_i, \mathbf{x}_{il}) \right)}_{\mu_{f_l \to x_i}(x_i)} \tag{2.18}$$

In short, to compute variable-to-factor messages, we take the product of all incoming factor-to-variable messages. To compute factor-to-variable messages, we sum over the product of the factor and the incoming variable-to-factor messages. If the node is a leaf (i.e., it has a single connection to another node), we set $\mu_{x \to f}(x) = 1$ if the leaf is a variable node and $\mu_{f \to x}(x) = f(x)$ if the leaf is a factor node.

In order to compute the marginal $p(x)$, we start by viewing the variable node x as the root of the tree. Starting from the leaves, we recursively send messages along the (unique) path towards the root node. As soon as the root node has received all messages from its neighbours, we multiply the messages and re-normalize them to obtain $p(x)$ [**bishop**]. If we are interested in all other marginals, we can compute them in a subsequent single downward pass (i.e., from the root towards the leaves) [**bishop**].

As long as the factor graph is a tree, BP computes the exact marginals [**bp**]. In general, as soon as we introduce cycles in the graph, BP has (1) no guarantee of convergence, and even if it converges, (2) has no guarantee of computing an accurate estimate of the true marginals [**bp**].

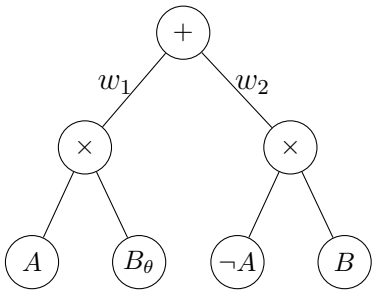
2.4 Probabilistic Circuits

Probabilistic models are central objects in machine learning (ML) and artificial intelligence (AI). Probability theory provides a principled, sound and consistent way to reason under uncertainty. If we knew the *true* data-generating probability distribution, many tasks in ML reduce to *probabilistic inference* [**pc_intro**].

Probabilistic Circuits (PCs) represent a class of probabilistic models. In essence, they can be viewed as parameterized functions that represent probability distributions (discrete, continuous, or a mix of both). However, when compared to other probabilistic models such as (deep) generative models and probabilistic graphical models, PCs leverage certain structural properties in order to answer classes of inference queries (such as marginals, conditionals or maximizations) exactly and efficiently, i.e., in time polynomial in the size of the circuit [**pc_intro**]. In general, the more structure we

impose on a circuit, the more classes of queries we can hope to answer *tractably* (i.e., efficiently) [**compositional_atlas**].

PCs can be considered as computational graphs that recursively mix (*sum nodes*) and factorize (*product nodes*) simpler parametric distributions (e.g., Bernoullis or Gaussians) that are located at the leaves of the graph. An example PC is shown in Figure **??**.



A	B	$p(A,B)$
0	0	0
0	1	w_2
1	0	$w_1(1-\theta)$
1	1	$w_1\theta$

Figure 2.8: A PC over 2 binary random variables A, B with parameters $\{w_1, w_2, \theta\}$. Let $\mathcal{B}(\pi)$ denote a Bernoulli distribution with success probability π. For the 4 leaf distributions, we use a shorthand notation and write A to mean $p_1(A) = \mathcal{B}(1)$, B_θ to mean $p_2(B) = \mathcal{B}(\theta)$, $\neg A$ to mean $p_3(A) = \mathcal{B}(0)$, and B to mean $p_4(B) = \mathcal{B}(1)$.

Definition 1 (Scope)**.** If a PC defines a joint distribution over a set of variables $\mathbf{x}$, each node can be seen as the root of a subgraph that models a distribution over a subset of $\mathbf{x}$. This subset is called the *scope* of the node, denoted $\phi(n)$ where n is a node in the PC. The scope of a leaf node is the set of input variables the base distribution models. If a node is not a leaf, its scope is given by the union of its children's scopes: $\phi(n) = \cup_j \phi(\text{input}(n)_j)$ with $\text{input}(n)$ denoting the vector of input nodes that feed into n. Hence, the root node r always has scope $\phi(r) = \mathbf{x}$.

Definition 2 (Size)**.** The *size* of a PC is the number of edges in the corresponding computational graph.

Definition 3 (Smoothness)**.** A sum node is *smooth* if all its inputs have the same scope. A PC is smooth if all its sum nodes are smooth.

Definition 4 (Decomposability)**.** A product node is *decomposable* if all its inputs have disjoint scopes, i.e., they do not share variables. A PC is decomposable if all of its product nodes are decomposable.

Definition 5 (Determinism)**.** A sum node is *deterministic* if, for any fully-instantiated input, at most one of its children assumes a non-zero value. A PC is deterministic if all sum nodes are deterministic.

The PC shown in Figure **??** is smooth, decomposable and deterministic.

Consider the PC shown in Figure **??**. To compute $p(A, B)$ for a particular assignment, say $A = 0, B = 1$, we evaluate the circuit bottom-up: First, we evaluate the probability mass functions at the leaves, yielding $p_1(A = 0) = 0, p_2(B = 1) = \theta, p_3(A = 0) = 1, p_4(B = 1) = 1$ (from left to right). Then, we perform the multiplications $0 \cdot \theta$ and $1 \cdot 1$ and feed the output in the weighted sum node at the root, computing $w_1 \cdot 0 + w_2 \cdot 1 = w_2 = p(A = 0, B = 1)$.

Theorem 1 (Tractable Marginal Queries)**.** A smooth and decomposable PC representing a joint distribution $p(\mathbf{x})$ can compute any marginal distribution $p(\mathbf{x}_s) = \sum_{\mathbf{x} \setminus \mathbf{x}_s} p(\mathbf{x})$ in linear time in the size of the circuit.[4]

We show a proof for this theorem in Appendix **??**.

Theorem 2 (Tractable MPE Queries)**.** A smooth, decomposable and *deterministic* PC representing a joint distribution $p(\mathbf{x})$ can compute *most probable evidence* (MPE) queries $\operatorname{argmax}_{\mathbf{x}} p(\mathbf{x})$ in linear time in the size of the circuit.

We prove this theorem in Appendix **??**.

Definition 6 (Structured Decomposability)**.** Given a PC whose product nodes all have exactly two inputs,[5] we call this PC *structured decomposable* if it is decomposable and every pair of product nodes n_1, n_2 with the same scope decompose in the same way: $(\phi(n_1) = \phi(n_2)) \Rightarrow (\forall j : \phi(\text{input}(n_1)_j) = \phi(\text{input}(n_2)_j))$ [**pc_intro**].

Of course, there are many decompositions to choose from (each of which gives a different PC). We will now introduce an object that specifies a particular decomposition for every product node, a so-called *vtree*.

Definition 7 (Vtree)**.** A *vtree*, or *variable tree*, is a full, rooted binary tree whose leaves are in a one-to-one correspondence to the variables in the PC [**new_comp_lang**].

Figure **??** shows the vtree that corresponds to the PC shown in Figure **??** (we say that the PC *respects* this vtree). For every product node in the PC, the decomposition of its scope is governed by exactly one *internal* node in the vtree. In this example, there is only one internal vtree node.

[4] In this work, we are only concerned with discrete probability distributions. However, in general, smooth and decomposable PCs can also compute marginals over continuous densities, i.e., integrals.

[5] Every PC can be converted into a circuit that fulfills this condition in exchange for a polynomial increase in size [**compositional_atlas**].

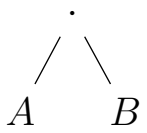


Figure 2.9: The unique vtree corresponding to the structured decomposable PC in Figure **??**.

2.5 Knowledge Compilation

We can represent knowledge bases (i.e., statements in propositional logic) using different *languages*, which impose different structural constraints on the representation of statements [**kcm**]. For example, *Conjunctive Normal Form* (CNF) and *Disjunctive Normal Form* (DNF) are two well-known languages [**kcm**]. Importantly, a language must trade-off (1) how compactly it can represent a given knowledge base and (2) the class of *queries* it supports in polytime [**kcm**]. For example, consider a knowledge base that is represented in both CNF and DNF. If we wish to enumerate all satisfying assignments (models), we observe that the DNF representation admits an answer to this query in polytime, while the CNF representation does not. The process of converting a knowledge base from one language (representation) to another is called *Knowledge Compilation*.

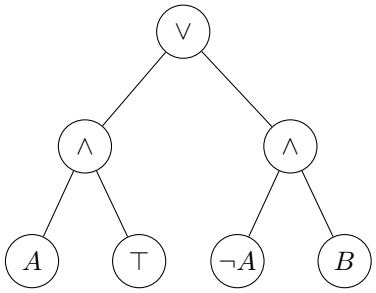
Again consider the PC in Figure **??**: *By construction* of the circuit, the assignment $A = 0, B = 0$ will always have zero probability — no matter what parameters w_1, w_2, θ we choose. We will now describe a principled way to build PCs that define distributions over the *models of a given propositional theory*.

2.5.1 Sentential Decision Diagrams

Consider a boolean formula $f(\mathbf{z})$ that maps instantiations of a set of binary variables $\mathbf{z}$ to either $\bot$ or $\top$. Given an instantiation $\mathbf{z}$, we can compute if f is satisfied (i.e., f outputs $\top$) by iteratively deciding on single variables $z \in \mathbf{z}$: For example, consider $f(A, B) = A \vee B$. To evaluate f for a given assignment for A, B, we may first check if $A = \top$: If so, we can immediately return $\top$. Otherwise, we must check if $B = \top$. If so, we return $\top$, otherwise $\bot$. This decision process can be represented by a *binary decision tree*, which can easily be turned into an *Ordered Binary Decision Diagram* (OBDD) by eliminating redundancies in the decision tree [**obdd**]. A *Sentential Decision Diagram* (SDD) extends the notion of an OBDD: While in OBDDs, decisions are based on *single variables*, SDDs can decide on entire *sentences*, making them a strict superset of OBDDs [**sdd**]. An SDD that represents the example above is shown in Figure **??**. Moreover, it has been shown that SDDs are exponentially more succinct than OBDDs, i.e., there exists a family of boolean functions where each member has polynomial

SDD size, but exponential OBDD size [**sdd_vs_obdd**]. An SDD can be seen as a type of logical circuit and, as we will later see, is closely related to PCs.

Definition 8 (Compressed Partitions)**.** Let $f(\mathbf{x}, \mathbf{y})$ be a boolean function over disjoint sets of variables $\mathbf{x}$ and $\mathbf{y}$. For any such f, we can write $f = (p_1(\mathbf{x}) \wedge s_1(\mathbf{y})) \vee \cdots \vee (p_k(\mathbf{x}) \wedge s_k(\mathbf{y}))$ and we call p_i *primes* and s_i *subs* [**sdd**, **psdd**]. Moreover, we can construct this decomposition in a way that $p_i \wedge p_j = \bot$ for $i \neq j$, $p_1 \vee \cdots \vee p_k = \top$, and $p_i \neq \bot$ for all i. Further, we restrict the subs to be distinct, i.e., $s_i \neq s_j$ for all $i \neq j$. We call $\{(p_1, s_1), \ldots, (p_k, s_k)\}$ a *compressed* $\mathbf{x}$*-partition* of f [**sdd**].



A	B	$f(A,B)$
0	0	0
0	1	1
1	0	1
1	1	1

Figure 2.10: An SDD representing $f(A, B) = (A \wedge \top) \vee (\neg A \wedge B) = A \vee B$. Note that we cannot directly attach A and B to an *or* node since this would violate the smoothness property of the circuit.

Definition 9 (SDD)**.** Let v be a vtree over binary variables $\mathbf{z}$. A *Sentential Decision Diagram* (SDD) is a computational graph where each internal node is either a $\vee$-node (*logical or*) or a $\wedge$-node (*logical and*), while leaf nodes correspond to variables, their negation, $\top$ (true), or $\bot$ (false) [**sdd**]. W.l.o.g., we demand that every $\vee$-node n has $k \geq 1$ $\wedge$-nodes as children, denoted $n_1, \ldots, n_k$. Each n_i has exactly two children, denoted p_i, s_i. Then, n represents a boolean function $g_n(\mathbf{x}, \mathbf{y})$ (where $\mathbf{x}, \mathbf{y}$ are disjoint sets of binary variables) such that $\{(p_i, s_i)\}_{i=1}^{k}$ is a compressed $\mathbf{x}$-partition of g_n. Further, there exists a node v' in the vtree v such that $\mathbf{x} = v'_l, \mathbf{y} = v'_r$ where v'_l, v'_r denote the set of variables mentioned in the left and right subtree of v', respectively. We say that n is *normalized* w.r.t. v'. Moreover, we say that an SDD *respects* a vtree v iff every $\vee$-node n in the SDD is *normalized* w.r.t. a vtree node in v.

For example, the SDD $\vee$-node in Figure **??** has two primes $A, \neg A$, and two subs B_θ, B. In essence, primes form a partition of the space of assignments and lead to deterministic circuits.

For a given boolean function f and vtree, we can construct an SDD that is *unique* (canonical SDD). Since the size of the resulting SDD heavily depends on the vtree, minimizing the size of the SDD amounts to finding a good vtree for f [**sdd**, **dynamic_min_choi**].

If f is in *Conjunctive Normal Form* (CNF), has n variables and treewidth w, there exists a canonical SDD of size $O(n2^w)$ [**sdd**].

Since SDDs can be combined using any boolean operation in polytime [**sdd**], we can compile a CNF f into an SDD by first constructing a circuit for each clause (which is trivial), followed by conjoining the resulting SDDs [**sdd**].

2.5.2 Probabilistic Sentential Decision Diagrams

It helps to think of SDDs as a kind of *unnormalized* PC, which we can easily turn into a proper PC by (1) replacing all *and* nodes with product nodes, (2) replacing all *or* nodes with sum nodes (with convex weights), and (3) replacing $\top$ leaf nodes with Bernoulli distributions with a free parameter θ [**psdd**]. The resulting circuit is a special case of a PC that is smooth, structured decomposable, and deterministic and was introduced under the name *Probabilistic Sentential Decision Diagram* (PSDD) [**psdd**]. For example, the SDD shown in Figure **??** was turned into the PSDD shown in Figure **??**.

Clearly, no matter how we parameterize a PSDD, it can only assign a positive probability mass to variable assignments that satisfy the original formula f we compiled into the underlying SDD.

It follows from the definition of an SDD that every sum node in a PSDD has one or multiple product nodes as children, all of which have exactly two inputs, which are again termed *primes* (left children) and *subs* (right children).

2.6 Circuit Multiplication

Consider a set of variables $\mathbf{x}$ and two PSDDs (i.e., smooth, structured decomposable, and deterministic PCs) $p(\mathbf{y})$ and $q(\mathbf{z})$ with $\mathbf{y}, \mathbf{z} \subseteq \mathbf{x}$ and sizes $|p|, |q|$, respectively. We wish to also represent the *product* $f(\mathbf{x}) = p(\mathbf{y}) \cdot q(\mathbf{z})$ as a PSDD (*circuit multiplication*). In this work, we will consider probability distributions that can be written as a product of factors. As we will later see, we can first (1) compile the individual factors into PCs and (2) use circuit multiplication to multiply the factors on the circuit level. We will then use the resulting PSDD to perform inference operations.

Definition 10 (Vtree Projection)**.** Let v bet a vtree over variables $\mathbf{x}$. To project v onto variables $\mathbf{y} \subseteq \mathbf{x}$, we successively remove every maximal subtree v_0 whose variables are not in $\mathbf{y}$, while replacing the parent of v_0 with its sibling [**sbn**].

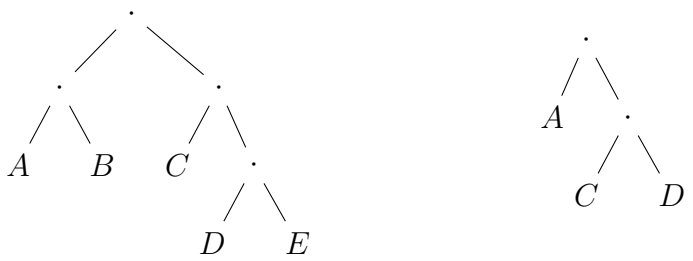


Figure 2.11: A vtree (left) and its projection onto $\{A, C, D\}$ (right).

Crucially, multiplying PSDDs is tractable: If p respects vtree v_p and q respects vtree v_q, f can be computed in $O(|p| \cdot |q|)$ if v_p and v_q are projections of a common vtree v and $|p|, |q|$ denote the size of p and q, respectively [**tractable_ops**]. We say that p and q are *compatible*. In general, $f(\mathbf{x})$ is not a distribution but must be divided by its normalization constant $\kappa = \sum_{\mathbf{x}} f(\mathbf{x})$, which can easily be computed during circuit multiplication without altering the asymptotic time complexity [**tractable_ops**]. While multiplying two circuits takes quadratic time, we note that in general, multiplying N PSDDs is therefore exponential in N.

Appendix **??** describes an algorithm that performs circuit multiplication of two PSDDs in quadratic time. While this algorithm assumes that both PSDDs are normalized w.r.t. the same vtree, this can be extended to projections of a common vtree without changing the asymptotic time complexity [**tractable_ops**].

2.7 Entropy and Kullback-Leibler Divergence

In this work, we will analyze approximations of probability mass functions by means of information-theoretic measures, such as *Entropy* and *Kullback-Leibler Divergence*.

Definition 11 (Entropy)**.** Let $p(\mathbf{x})$ be a probability mass function over variables $\mathbf{x}$. The entropy $H(p)$ is defined as

$$H(p) = \mathbb{E}\left[-\log\left(p(\mathbf{x})\right)\right] = -\sum_{\mathbf{x}} p(\mathbf{x}) \cdot \log\left(p(\mathbf{x})\right) \tag{2.19}$$

Informally, this quantifies the *expected information* about $\mathbf{x}$, according to p. If p is a uniform distribution, i.e., $\forall \mathbf{x} \in \mathcal{X} : p(\mathbf{x}) = \frac{1}{|\mathcal{X}|}$, then $H(p)$ is maximized. On the other hand, if all probability mass is concentrated on a single point, i.e., $\exists \mathbf{x} \in \mathcal{X} : p(\mathbf{x}) = 1$, then $H(p) = 0$. Accordingly, we can interpret H as a measure of diffusion.

Definition 12 (Kullback-Leiber Divergence)**.** Let $p(\mathbf{x}), q(\mathbf{x})$ be probability mass functions over variables $\mathbf{x}$. The *Kullback-Leibler (KL) Divergence* between p and q, written[6] $D_{KL}(p||q)$, is defined as

$$D_{KL}(p||q) = \mathbb{E}_p \left[\log \left(\frac{p(\mathbf{x})}{q(\mathbf{x})} \right) \right] = \sum_{\mathbf{x}} p(\mathbf{x}) \cdot \log \left(\frac{p(\mathbf{x})}{q(\mathbf{x})} \right) \tag{2.20}$$

where we demand that $\forall \mathbf{x} : (p(\mathbf{x}) > 0) \Rightarrow (q(\mathbf{x}) > 0)$, i.e., the support of q must not be smaller than the support of p.

Note that $D_{KL}(p||q)$ is non-negative for any p, q and is only 0 if and only if p, q are identical, i.e., $\forall \mathbf{x} : p(\mathbf{x}) = q(\mathbf{x})$.

[6]We sometimes abuse notation and write $D_{KL}(p(\mathbf{x}) \mid\mid q(\mathbf{x}))$ to emphasize the set of variables the distributions depend on, respectively.

3 Methods

In this work, we attack an unprotected implementation of the *Advanced Encryption Standard* (AES), a popular symmetric-key encryption algorithm [**aes**]. We are given a dataset $\mathcal{D} = \{(\boldsymbol{\ell}^{(i)}, \mathbf{k}^{(i)}, \mathbf{p}^{(i)})\}_{i=1}^{n}$ which contains n samples which each consist of a leakage $\boldsymbol{\ell}^{(i)} \in \mathbb{R}^d$, a b-byte key $\mathbf{k}^{(i)}$, and a b-byte plaintext $\mathbf{p}^{(i)}$. Often, we will reason about individual bytes in $\mathbf{k}$ and $\mathbf{p}$, denoted $k_j, p_j \in \mathbb{F}_2^8$, $1 \leq j \leq b$. Further, assume we have access to an algorithm, that, given some $\boldsymbol{\ell}$, produces probability mass functions over intermediate variables, i.e., $p(v_j \mid \boldsymbol{\ell})$, where v_j is an arbitrary intermediate variable.

3.1 Problem Description

Recall from Section **??** that in the AES, the key schedule produces keys $\mathbf{k}^0, \ldots, \mathbf{k}^{10}$ where $\mathbf{k}^0$ is just the key $\mathbf{k}$. Thus, before executing the first round of AES, we call AddRoundKey, which computes the XOR between the key $\mathbf{k}$ and the plaintext $\mathbf{p}$. Thus, this operation, followed by the first round of AES, is a popular attack target as it directly involves the key we wish to recover, and the plaintext we assume to know. We also emphasize that we do not attack the key schedule in our work. Algorithm **??** shows the first operations of AES encryption (up to the first MixColumns call), which will be the target of this work. Moreover, we skip the first ShiftRows operation, since it merely corresponds to a fixed re-labeling of the input bytes (e.g., the true bytes k_6, p_6 are called k_2, p_2) and does not affect probabilistic inference.

Algorithm 1 Simplified Beginning of AES-128

Input: Key bytes $k_1, \ldots, k_{16}$, Plaintext bytes $p_1, \ldots, p_{16}$

1: **for** i in $1, \ldots, 16$ **do**
2: $\quad y_i \leftarrow k_i \oplus p_i$ ▷ AddRoundKey
3: $\quad x_i \leftarrow S(y_i)$ ▷ SubBytes
4: **end for**
▷ We skip ShiftRows because it only corresponds to re-labeling k_i, p_i
5: **for** i in $0, 4, 8, 12$ **do**
6: $\quad x_{i+1}^{(m)}, \ldots, x_{i+4}^{(m)} \leftarrow \text{MixColumn}(x_{i+1}, \ldots, x_{i+4})$ ▷ MixColumns
7: **end for**

Algorithm 2 MixColumn

Input: Input bytes $x_1, \ldots, x_4$

1: $x_{12},\ x_{23},\ x_{34},\ x_{41} \leftarrow (x_1 \oplus x_2), (x_2 \oplus x_3), (x_3 \oplus x_4), (x_4 \oplus x_1)$
2: $g \leftarrow x_{12} \oplus x_{34}$
3: $\tilde{x}_{12},\ \tilde{x}_{23},\ \tilde{x}_{34},\ \tilde{x}_{41} \leftarrow \text{XTIME}(x_{12}), \text{XTIME}(x_{23}), \text{XTIME}(x_{34}), \text{XTIME}(x_{41})$
4: $x'_{12},\ x'_{23},\ x'_{34},\ x'_{41} \leftarrow (\tilde{x}_{12} \oplus g), (\tilde{x}_{23} \oplus g), (\tilde{x}_{34} \oplus g), (\tilde{x}_{41} \oplus g)$
5: $x_1^{(m)},\ x_2^{(m)},\ x_3^{(m)},\ x_4^{(m)} \leftarrow (x_1 \oplus x'_{12}), (x_2 \oplus x'_{23}), (x_3 \oplus x'_{34}), (x_4 \oplus x'_{41})$
6: **return** $x_1^{(m)},\ x_2^{(m)},\ x_3^{(m)},\ x_4^{(m)}$

As discussed in Section **??**, when performing a SASCA, we need to convert the algorithmic description into a *factor graph*. When doing so, the resulting factor graph (shown in Figure **??**) turns out to be *cyclic/loopy*. The reason for this is the structure of the MixColumns routine. As described in Section **??**, directly running belief propagation (BP) to compute marginals in this graph results in an approximate solution.

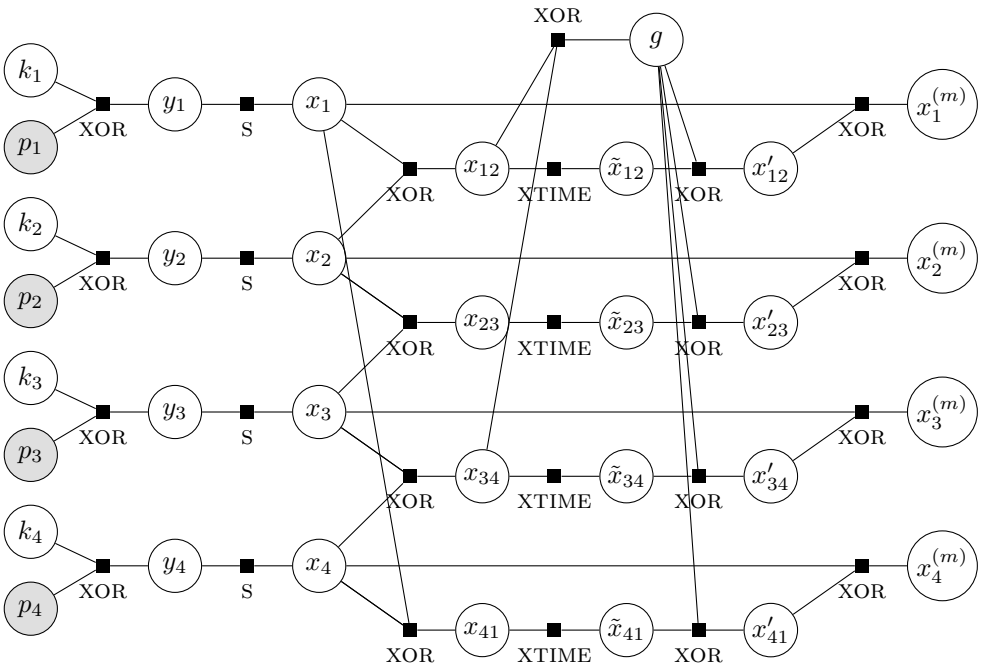


Figure 3.1: Simplified factor graph showing operations on the first 4 bytes key and plaintext bytes in AES (Algorithm **??**). Importantly, note that this graph contains *cycles*. Every variable node v (except p_i, which is assumed to be known) has an additional factor $p(v|\boldsymbol{\ell})$, which we hide since it impairs clear visualization.

A key insight of our approach is the fact that we can, in some cases, *compile the loopy parts of the graph into a structure that supports tractable marginalization*. This effectively *lumps* together loopy parts of the factor graph into a *single* factor, resulting

in an *acyclic* factor graph (Figure **??**).

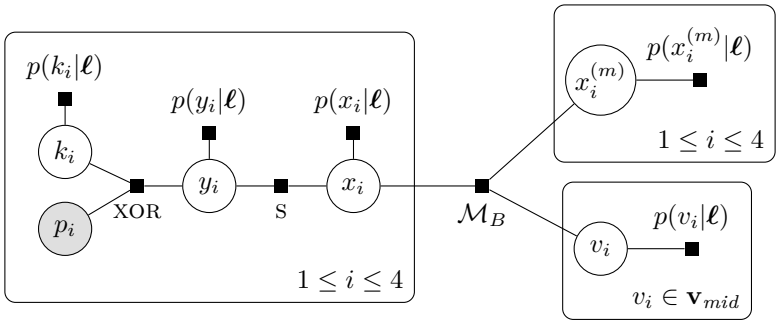


Figure 3.2: We aggregate the loopy MixColumn subgraph into a single (large) boolean factor $\mathcal{M}_B$. In this view, we show the additional posterior distribution factors hidden in Figure **??**. To simplify visualization, we use *plate* notation: Each subgraph in a rectangle repeats as often as specified.

Note that simply abstracting a loopy subgraph as a factor node is of no use if we do not have efficient means to perform inference *within* the introduced node: Assume that given the factor graph in Figure **??**, we wish to marginalize out all variables except k_i. Let $\mathbf{v}$ be the collection of variables involved in the MixColumns computation, i.e.,

$$\mathbf{v} = (\underbrace{x_1, \ldots, x_4}_{\text{Inputs } \mathbf{v}_{in}}, \quad \underbrace{x_{12}, \ldots, x'_{41}}_{\text{Intermediates } \mathbf{v}_{mid}}, \quad \underbrace{x_1^{(m)}, \ldots, x_4^{(m)}}_{\text{Outputs } \mathbf{v}_{out}}) \tag{3.1}$$

To use the notation defined in Section **??**, $\mathcal{M}_B(\mathbf{v})$ is the boolean representation of MixColumns, evaluating to 1 if $\mathbf{v}$ is consistent with MixColumn, and to 0 else.

As illustrated in Equation **??**, the message that the $\mathcal{M}_B$ node must send to its neighbour $x_i \in \mathbf{v}_{in}$ is given by

$$\mu_{\mathcal{M}_B \to x_i}(x_i) = \sum_{\mathbf{v} \setminus x_i} \mathcal{M}_B(\mathbf{v}) \cdot \prod_{v_j \in \mathbf{v} \setminus x_i} p(v_j|\boldsymbol{\ell}) \tag{3.2}$$

For a single value of x_i, computing this loop in a naïve manner (i.e., iterating over all combinations of $\mathbf{v}$) would take 2^{136} loop iterations. However, we do not need to take into account combinations of $\mathbf{v}$ that are inconsistent with MixColumn. Instead, we could, for a fixed value of x_i, loop over all possible combinations of *remaining input bytes* $\mathbf{v}_{in} \setminus x_i$, compute their corresponding intermediate and output values, and only consider these assignments of $\mathbf{v}$ (since all other assignments yield $M_B(\mathbf{v}) = 0$ and thus, do not contribute to the sum). Formally, with

$$\mathcal{V}_i(y) = \{\mathbf{v} \mid \mathcal{M}_B(\mathbf{v}) = 1 \wedge x_i = y\} \tag{3.3}$$

compute

$$\mu_{\mathcal{M}_B \to x_i}(x_i) = \sum_{\mathbf{v} \in \mathcal{V}_i(x_i)} \prod_{v_j \in \mathbf{v} \setminus x_i} p(v_j | \boldsymbol{\ell}) \tag{3.4}$$

For all values $x_i \in \{0, \ldots, 255\}$, this takes 2^{32} loop iterations in total, where each iteration takes time $O(N + \text{time}(\textsc{MixColumn}))$, where N is the number of bytes in $\mathbf{v}$. Since we are only interested in marginals w.r.t. key bytes, we only need to compute messages for $x_1, \ldots, x_4$, thus running this procedure 4 times.

Again, this is a naïve approach as (1) the computational complexity grows exponentially with the number of inputs, and, importantly, (2) we do not take advantage of the *structure* of $\mathcal{M}_B$, treating it as a black box.[1]

3.2 Message Computation as Circuit Multiplication

Consider the message computation in Equation **??**. To compute the message for some x_i, we wish to represent both $\mathcal{M}_B(\mathbf{v})$ and all probability mass functions (PMFs) $p(v_j|\boldsymbol{\ell}), v_j \in \mathbf{v} \setminus x_i$ as probabilistic sentential decision diagrams (PSDDs). Afterwards, we could compute the *circuit product* of these PSDDs and marginalize out all variables except x_i in linear time in the size of the resulting circuit.

We leverage existing SDD compilation techniques [**dynamic_min_choi**] to find a vtree $v_{\mathcal{M}_B}$ for a CNF representation of $\mathcal{M}_B$ such that the corresponding SDD is succinct. We then project $v_{\mathcal{M}_B}$ onto smaller vtrees whose scope is always a single byte. Next, we compile each PMF $p(v_j|\boldsymbol{\ell})$ into a PSDD that respects the corresponding projected vtree for this byte. This ensures that all circuits are *compatible*. Figure **??** sketches the end-to-end compilation process.

This approach differs from existing compilation techniques for probabilistic graphical models (PGMs) [**sbn**, **tractable_ops**] in the way that we first perform a vtree search such that the SDD compilation of $\mathcal{M}_B$ is tractable in the first place.

3.3 CNF Representation of MixColumn

Given the algorithmic description of $\mathcal{M}$ (Algorithm **??**), we wish to construct a boolean representation $\mathcal{M}_B(\mathbf{v})$ in CNF. For this task, we use the Python package `pyeda` [**pyeda**] which supports logic minimization and symbolic boolean algebra on multivariate arrays of boolean variables. In essence, we replace all variable assignments

[1] Here, we only use the fact that $\mathcal{M}$ is a deterministic function with inputs $x_1, \ldots, x_4$.

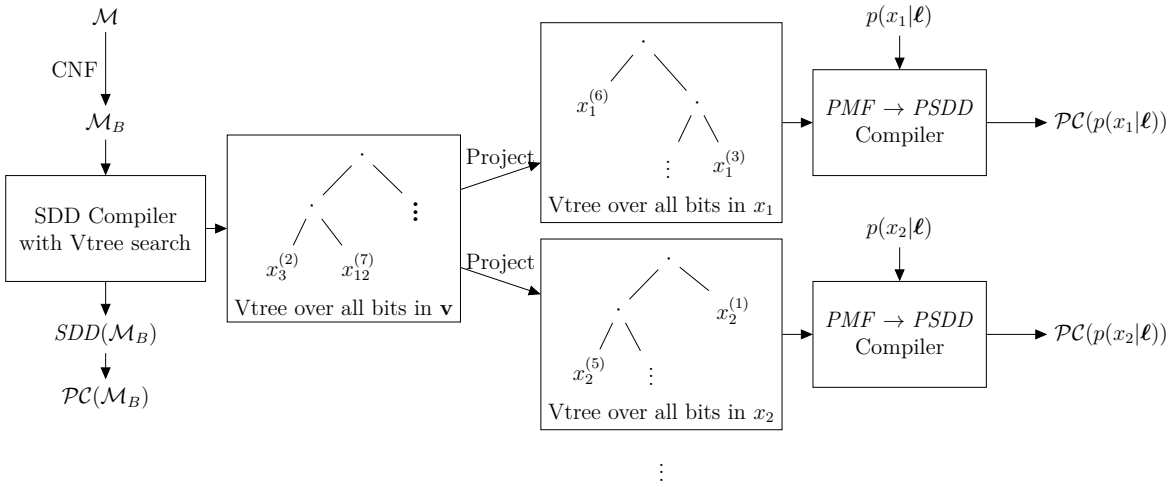


Figure 3.3: Given the algorithmic description of MixColumn ($\mathcal{M}$) and the posterior distributions $p(\cdot|\boldsymbol{\ell})$ as inputs to our compilation pipeline, we end up with PSDD representations of all input distributions, denoted $\mathcal{PC}(\cdot)$. Moreover, all PSDDs are pairwise *compatible* for downstream circuit multiplication tasks.

in Algorithm **??** with equality constraints to obtain a boolean representation of $\mathcal{M}$. Since we model all 21 bytes in $\mathbf{v}$ (and do not introduce further auxiliary variables), the resulting CNF expression reasons over $21 \cdot 8 = 168$ boolean variables. We find that the final CNF expression consists of 648 clauses and has a *size* of 2649, where `pyeda` defines *size* recursively: Each literal has size 1 and each operator $(\wedge, \vee)$ has size equal to the sum of the sizes of its children plus 1. The average number of literals within a clause is 3.09.

3.4 SDD Compilation

To compile our CNF $\mathcal{M}_B$ into an SDD, we leverage the open-source SDD compiler suite[2] by Choi and Darwiche. More precisely, we utilize both the *bottom-up* compiler introduced in [**dynamic_min_choi**], as well as the *top-down* compiler in [**top_down_comp**]. The bottom-up compiler implements a dynamic vtree search algorithm, which heuristically searches through the space of vtrees to minimize the size of the resulting SDD [**dynamic_min_choi**]. The compiler takes an initial vtree as input, which will be modified during compilation. We compile $\mathcal{M}_B$ using a variant of different initial vtrees and find that a vtree constructed with the top-down compiler [**top_down_comp**] (`miniC2D`[3], using a hypergraph with fixed balance factor)

[2] `http://reasoning.cs.ucla.edu/sdd/`
[3] `http://reasoning.cs.ucla.edu/minic2d/`

empirically produces the most succinct SDD representation with a size of ≈ 12000 edges (≈ 4000 nodes) and takes < 10 seconds to compile on a modern laptop CPU.

3.5 Compiling PMFs into PSDDs

Let $\mathbf{z} \in \mathbb{F}_2^8$ be a bitstring of length 8 (byte) with individual bits $z_1, \ldots, z_8$. Given a probability mass function (PMF) $p(\mathbf{z})$ and a vtree v over $z_1, \ldots, z_8$, we wish to produce a PSDD that respects v and represents $p(\mathbf{z})$ *exactly*. In essence, we can achieve this by recursively computing conditionals of p according to the structure of v. A simple example is shown in Figure **??** and a simplified pseudocode algorithm for compiling an arbitrary PMF into a PSDD is given in Appendix **??**.

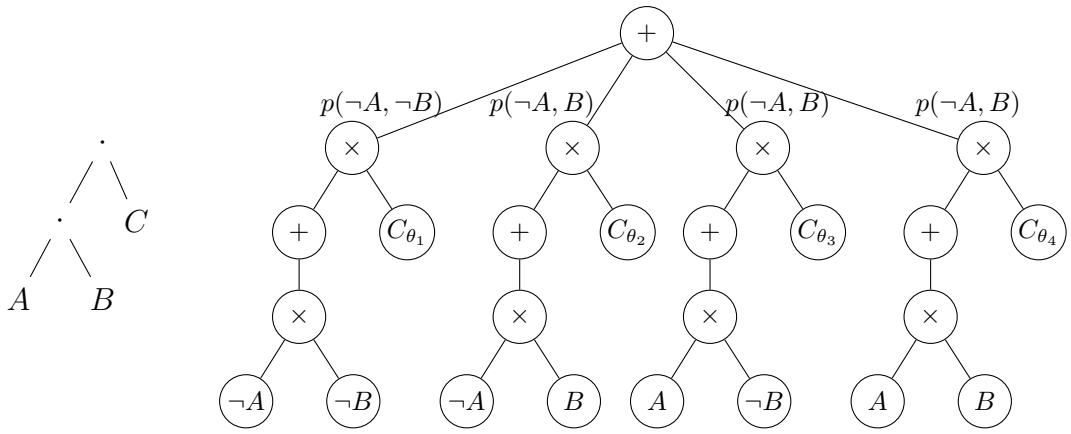


Figure 3.4: A vtree over 3 bits A, B, C (left) and a PSDD respecting this vtree (right) representing an arbitrary PMF $p(A, B, C)$. We use the factorization $p(A, B, C) = p(C|A, B) \cdot p(A, B)$ and set $\theta_1 = p(C|\neg A, \neg B), \theta_2 = p(C|\neg A, B), \theta_3 = p(C|A, \neg B), \theta_4 = p(C|A, B)$.

3.6 PSDD Multiplication

To obtain the message in Equation **??** via circuit multiplication, we need to compute

$$\mu_{\mathcal{M}_B \to x_i}(x_i) = \sum_{\mathbf{v} \setminus x_i} \mathcal{PC}\left(\mathcal{M}_B(\mathbf{v})\right) \cdot \prod_{v_j \in \mathbf{v} \setminus x_i} \mathcal{PC}\left(p(v_j|\boldsymbol{\ell})\right) \tag{3.5}$$

where $\mathcal{PC}(\cdot)$ denotes the circuit representation of its input and all multiplications denote circuit multiplications. In this case, all PCs are PSDDs. As discussed in Section **??**, multiplying two compatible circuits takes quadratic time. For each message, we perform $(N-1)$ multiplications (where N is the number of bytes in $\mathbf{v}$). For simplicity, if we assume that each circuit has size s, the entire multiplication chain takes at most

$O(s^N)$ time, which is exponential in the number of bytes. We emphasize that this is a worst-case bound and note that, depending on the structure of the circuits, the number of operations can be much lower in practice.

Unfortunately, this multiplication chain is still intractable for arbitrary posteriors as the circuit grows prohibitively large quickly. However, we do not need to perform *all* of the circuit multiplications, just to marginalize out all but one variable in the resulting circuit. Instead, after performing a single multiplication, we can immediately marginalize out the byte whose distribution we have just multiplied. The rationale behind this approach is the fact that marginalizing out a set of variables from a PSDD results in a *strictly smaller* smooth and decomposable PC (albeit non-deterministic).

$$\mu_{\mathcal{M}_B \to x_i}(x_i) = \sum_{v_{N-1}} \cdots \left(\sum_{v_2} \left(\sum_{v_1} \mathcal{PC}(\mathcal{M}_B(\mathbf{v})) \cdot \mathcal{PC}(p(v_1|\boldsymbol{\ell})) \right) \cdot \mathcal{PC}(p(v_2|\boldsymbol{\ell})) \right) \cdot \cdots \cdot \mathcal{PC}(p(v_{N-1}|\boldsymbol{\ell})) \tag{3.6}$$

where $v_1, \ldots, v_{N-1} \in \mathbf{v} \backslash x_i$. However, since many bits of *different* bytes $v_1, \ldots, v_{N-1}$ are heavily entangled in $\mathcal{PC}(\mathcal{M}_B(\mathbf{v}))$, marginalizing out a single byte does not significantly reduce the size of the circuit. Thus, in practice, this optimization is not enough to tractably compute the entire multiplication chain.

In essence, there are 3 approaches to this problem that rely on approximation: (1) Approximate the sum $\sum_{\mathbf{v} \backslash x_i}$ (this is what loopy BP does), (2) approximate $\mathcal{M}_B(\mathbf{v})$ (e.g., by removing clauses in the CNF), or (3) approximate the posterior distributions $p(v_j|\boldsymbol{\ell})$.

We choose the third option and investigate methods to perform repeated multiplication tractably by approximating the input distributions.

3.7 Input Approximations

Recall that we wish to compute

$$\mu_{\mathcal{M}_B \to x_i}(x_i) = \sum_{\mathbf{v} \backslash x_i} \mathcal{PC}\left(\mathcal{M}_B(\mathbf{v})\right) \cdot \prod_{v_j \in \mathbf{v} \backslash x_i} \mathcal{PC}\left(p(v_j|\boldsymbol{\ell})\right) \tag{3.7}$$

via means of SDD compilation and circuit multiplication. However, in practice, we cannot compute this multiplication chain efficiently: Recall that if we multiply N PSDDs with sizes $s_1, \ldots, s_N$, the multiplication chain takes $O(\prod_{i=1}^{N} s_i)$ operations in general. To alleviate the computational burden, we wish to keep the sizes s_i as small as possible. As discussed previously, one way to achieve this is by means of *approximating* the distributions we compile into PSDDs, such that we do not need as many PSDD nodes to represent the distributions. Next, we discuss two variants of

this approach, namely (1) introducing conditional independencies, and (2) sparsifying the distributions before compiling them.

3.7.1 Conditional Independence

Consider a byte x as a vector of bits $x = (x^{(1)}, \ldots, x^{(8)})^T \in \mathbb{F}_2^8$ and let $p(x^{(1)}, \ldots, x^{(8)})$ denote a probability mass function. By the chain rule of probability, we can, for any ordering $x^{(i_1)}, \ldots, x^{(i_8)}$, write

$$p\left(x^{(1)}, \ldots, x^{(8)}\right) = p\left(x^{(i_1)}\right) p\left(x^{(i_2)} \mid x^{(i_1)}\right) \cdots p\left(x^{(i_8)} \mid x^{(i_7)}, \ldots, x^{(i_1)}\right) \tag{3.8}$$

For every byte distribution we compile into a PSDD, we are given the corresponding (projected) vtree v with leaves $x^{(1)}, \ldots, x^{(8)}$. Assume that traversing v in post-order visits the leaves in the order $x^{(i_1)}, \ldots, x^{(i_8)}$. The routine that recursively compiles p into a PSDD (shown in Algorithm **??**) then decomposes p exactly as shown in Equation **??**. Since the resulting PSDD makes decisions *based on every bit*, it is a natural idea to decrease the size of the PSDD representation $\mathcal{PC}\left(p(x)\right)$ by *removing decisions*, which corresponds to introducing *conditional independence* assumptions. For example, if we assume that $x^{(i_1)}$ is independent from the rest of the bits, we could write

$$p\left(x^{(1)}, \ldots, x^{(8)}\right) = p\left(x^{(i_1)}\right) p\left(x^{(i_2)}\right) \cdots p\left(x^{(i_8)} \mid x^{(i_7)}, \ldots, x^{(i_2)}\right) \tag{3.9}$$

and thus, never base a decision on $x^{(i_1)}$ in the compiled PSDD. For every bit in $x^{(i_1)}, \ldots, x^{(i_8)}$, we can either choose to *decide* on that bit, or neglect the decision. Thus, there exist $2^8 = 256$ PSDDs with different conditional independence assumptions, where 255 of them strictly decrease the size of the PSDD (since we neglect at least one decision). However, in practice, there are no conditional independencies in the given distributions, and introducing them quickly leads to empirically bad approximations. We will next discuss a different approach that can, empirically, compress the PSDD representations very well, while achieving a good approximation of the input PMFs.

3.7.2 Input Sparsification

When performing a single-trace attack, it is reasonable to assume that at least *some* posterior distributions $p(v_j|\boldsymbol{\ell}), v_j \in \mathbf{v}$ have *low entropy*, i.e., the probability mass is concentrated on only a few values.[4]

[4]If we assume the opposite, i.e., that all posterior distributions have high entropy, then it usually becomes unrealistic to perform single-trace attacks in the first place.

Consequently, we approximate these distributions by *sparsfying* them: We can interpret a PMF $\mathbf{p}$ as a d-dimensional point on the unit simplex, i.e.,

$$\mathbf{p} \in \Delta_d \quad \text{with } \Delta_d = \left\{ (p_1, \ldots, p_d)^T \mid \forall i : p_i \leq 0, \|\mathbf{p}\|_1 = 1 \right\} \tag{3.10}$$

where $\|\mathbf{p}\|_1 = \sum_{i=1}^{d} |p_i|$ denotes the L_1 norm. In our case, we have $d = 256$.

For a given distribution $\mathbf{p}$ and mass threshold $0 < \pi \leq 1$, we produce an approximation $\tilde{\mathbf{p}}$ such that the maximum number of components $\tilde{p}_i$ are 0 while $\|\mathbf{p}\|_1 \geq \pi$. Finally, we renormalize by setting $\tilde{\mathbf{p}} \leftarrow \tilde{\mathbf{p}} \oslash \|\tilde{\mathbf{p}}\|_1$, where $\oslash$ denotes Hadamard (component wise) division. In other words, we clamp the smallest entries in $\mathbf{p}$ to 0 while preserving probability mass $\geq \pi$. We call $\tilde{\mathbf{p}}$ a *π-sparse* approximation of $\mathbf{p}$.

Crucially, for an appropriate π, $\tilde{\mathbf{p}}$ can represent *modes* of $\mathbf{p}$ very well and can be succinctly encoded in a PSDD. Since we need to multiply our MixColumn PSDD with a *product* of posterior distributions $\prod_{v_j \in \mathbf{v} \setminus x_i} p(v_j | \boldsymbol{\ell})$, we wish to approximate multiple distributions. We will now introduce an information-theoretic upper bound on the approximation error that is introduced by sparsification.

Theorem 3. Let $p(\mathbf{v} \mid \boldsymbol{\ell}) = \prod_{j=1}^{K} p(v_j \mid \boldsymbol{\ell})$ be a factorized distribution and let $\tilde{p}(\mathbf{v} \mid \boldsymbol{\ell})$ be a sparse approximation where $0 \leq k \leq K$ distributions have been sparsified with mass preserving parameter π. If $\mathcal{J} \subseteq \{1, \ldots, K\}$ is the set of indices of sparsified distributions ($|\mathcal{J}| = k$), we can thus write

$$\tilde{p}(\mathbf{v} \mid \boldsymbol{\ell}) = \prod_{j \in \tilde{J}} \tilde{p}_\pi(v_j \mid \boldsymbol{\ell}) \cdot \prod_{j \notin \tilde{J}} p(v_j \mid \boldsymbol{\ell}) \tag{3.11}$$

where $\tilde{p}_\pi$ denotes a π-sparse approximation of p. Then, the (reverse) Kullback-Leibler Divergence D_{KL} between $\tilde{p}(\mathbf{v} \mid \boldsymbol{\ell})$ and $p(\mathbf{v} \mid \boldsymbol{\ell})$ can be bounded:

$$D_{KL}\left(\tilde{p}(\mathbf{v} \mid \boldsymbol{\ell}) \;||\; p(\mathbf{v} \mid \boldsymbol{\ell})\right) \leq -\log(\pi) \cdot k \tag{3.12}$$

We prove this fact in Appendix **??**.

3.8 Most Probable Key Assignment

Until now, we have discussed approaches to obtain marginals over key bytes $p(k_i \mid \boldsymbol{\ell})$ via BP. After obtaining these distributions, many attacks in the literature [**5min**, **32bit**] then assume that

$$p(k_1, \ldots, k_n \mid \boldsymbol{\ell}) = \prod_{i=1}^{n} p(k_i \mid \boldsymbol{\ell}) \tag{3.13}$$

with n denoting the number of key bytes. However, this is usually a strong assumption, as in reality, the key bytes might *not* be conditionally independent given the leakages. A thorough analysis of this assumption can be found in Section .

Moreover, the aim of an attacker is not to compute the full joint distribution, but to learn

$$\underset{k_1,\ldots,k_n}{\operatorname{argmax}} \; p(k_1, \ldots, k_n \mid \boldsymbol{\ell}) \tag{3.14}$$

or, in general, the top-N most probable key assignments. Computing (or estimating) marginals from the joint is usually done due to computational advantages, since directly querying the joint distribution is often intractable. However, given our problem, we will now describe a method that uses deterministic PCs to tractably[5] compute the most probable key assignment in the resulting posterior distribution.

Recall the factor graph in Figure **??**. The key idea of this approach is to essentially summarize the entire factor graph within a single deterministic PC, which can answer MPE queries tractably. For each i, we can, starting from the very left of the factor graph, first send a BP message from k_i to the XOR node (as described in Section **??**), which sends its message to y_i. Next, y_i sends its message to the SBOX factor node, which passes its message to the x_i node (left-to-right message passing). Finally, x_i collects this message and sends its message $\mu_{x_i \to \mathcal{M}_B}(x_i)$ to the $\mathcal{M}_B$ factor node. From the perspective of $\mathcal{M}_B$, it receives a message from each x_i, as well as from its other neighbours (which are just the posterior distributions of intermediate values). Recall that $\mathcal{M}_B$ is represented as a PSDD and that we can compile all incoming messages into compatible PSDDs. Instead of computing marginals, we now wish to compute

$$\mathbf{v}^* = \underset{\mathbf{v}}{\operatorname{argmax}} \; \mathcal{PC}\left(\mathcal{M}_B(\mathbf{v})\right) \cdot \prod_{v_j \in \mathbf{v} \setminus \mathbf{v}_{in}} \mathcal{PC}\left(p(v_j|\boldsymbol{\ell})\right) \cdot \prod_{i=1}^{4} \mathcal{PC}\left(\mu_{x_i \to \mathcal{M}_B}(x_i)\right) \tag{3.15}$$

Recall that we can represent the entire expression we wish to compute the argmax of into a single PC by means of circuit multiplication. As the resulting circuit is *still determinsitic*, we can compute the argmax in linear time in the size of the circuit. Note that $\mathbf{v}^*$ also contains the most probable assignment for input variables[6] $\mathbf{v}_{in}$, denoted $x_1^*, \ldots, x_4^*$. Finally, we use the deterministic relationship between $x_1^*, \ldots, x_4^*$ and the key bytes (given the plaintext bytes) to recover the most probable key assignment $k_1, \ldots, k_4$.

Moreover, we can easily generalize this algorithm to find the $N \in \mathbb{N}$ most probable key assignments (top-N MPE query). We discuss the applicability of this approach to general problems in Section **??**.

[5] Assuming that circuit multiplication was tractable in the first place.
[6] This is because given $\mathbf{v}_{in}$, $\mathbf{v}_{mid}$ and $\mathbf{v}_{out}$ follow deterministically.

3.9 Scope of Posterior Distributions

Up to now, we have always assumed that every posterior distribution $p(v|\boldsymbol{\ell})$ reasons over a single byte, i.e., $v \in \{0, \ldots, 255\}$. However, in some state-of-the-art side-channel attacks [**bit_posterior1**, **bit_posterior2**, **breaking_free**], the authors estimate distributions over individual *bits* and assume that $p(v|\boldsymbol{\ell}) = \prod_{i=1}^{8} p(v^{(i)}|\boldsymbol{\ell})$. The message computation in Equation **??** then becomes

$$\mu_{\mathcal{M}_B \to x_i}(x_i) = \sum_{\mathbf{v} \setminus x_i} \mathcal{M}_B(\mathbf{v}) \cdot \prod_{v_j \in \mathbf{v} \setminus x_i} p(v_j|\boldsymbol{\ell}) \tag{3.16}$$

$$= \sum_{\mathbf{v} \setminus x_i} \mathcal{M}_B(\mathbf{v}) \cdot \prod_{v_j \in \mathbf{v} \setminus x_i} \prod_{k=1}^{8} p\left(v_j^{(k)}|\boldsymbol{\ell}\right) \tag{3.17}$$

In other words, the product over byte-distributions reduces to a distribution of fully-factorized Bernoullis.

Theorem 4. Let $\mathbf{x} = (x_1, \ldots, x_n)^T$ be a collection of binary random variables and $p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i)$ a product of Bernoulli distributions. For any vtree v over $\mathbf{x}$, we can construct a PC that encodes $p(\mathbf{x})$ and respects v in time $O(n)$. The resulting PC has size $O(n)$ and is smooth, structured decomposable, and deterministic (i.e., the resulting circuit is a PSDD).

A constructive proof of this theorem is given in Appendix **??**. Hence, we can easily represent the product of Bernoullis as a single PC that is compatible with $\mathcal{PC}(\mathcal{M}_B(\mathbf{v}))$ — without explicit circuit multiplications. We can thus compute a message using a single circuit multiplication:

$$\mu_{\mathcal{M}_B \to x_i}(x_i) = \sum_{\mathbf{v} \setminus x_i} \mathcal{PC}(\mathcal{M}_B(\mathbf{v})) \cdot \mathcal{PC}\left(\prod_{v_j \in \mathbf{v} \setminus x_i} \prod_{k=1}^{8} p\left(v_j^{(k)}|\boldsymbol{\ell}\right)\right) \tag{3.18}$$

As shown next, the remaining circuit multiplication is also easy from a computational perspective.

Theorem 5. Assume p_1 is a PSDD over n binary random variables that respects vtree v and has size s_1, and p_2 is a PSDD with size $O(n)$ that encodes a distribution of fully-factorized Bernoullis and was constructed as shown in Appendix **??**. Computing the circuit product $p_1 \cdot p_2$ using Algorithm **??** takes $O(s_1)$ time and the resulting circuit is of size $O(s_1)$.

A proof of this theorem is given in Appendix **??**. Consequently, computing belief propagation messages via circuit multiplications is particularly advantageous in the case where the given posterior distributions factorize on the bit level.

4 Experimental Results

4.1 Dataset

To evaluate our method, we use a dataset of asynchronously captured power traces from an SMT32F415 (ARM) microcontroller [**dlsca_defcon**] running *tinyAES* [**tinyaes**]. The dataset $\mathcal{D} = \{(\boldsymbol{\ell}^{(i)}, \mathbf{k}^{(i)}, \mathbf{p}^{(i)})\}_{i=1}^{n}$ contains $n = 131,072$ items, each of which consists of a high-dimensional power trace $\boldsymbol{\ell} \in \mathbb{R}^{50000}$, and 128-bit (16 byte) key and plaintext vectors $\mathbf{k}, \mathbf{p} \in \mathbb{F}_2^{128}$. A leakage $\ll$ contains the power consumption of the device during all 10 rounds of AES. However, since we only attack the first round[1] of AES, we only use the first $20,000$ values in each $\boldsymbol{\ell}$ in all experiments (i.e., $\boldsymbol{\ell} \in \mathbb{R}^{20000}$). Other than that, we do not perform additional pre-processing steps. In $\mathcal{D}$, we have 512 unique keys $\mathbf{k}$. For each unique key, the dataset contains 256 elements, each with a different plaintext $\mathbf{p}$. During the attack phase, we assume the plaintext to be known (*known plaintext attack*).

We use 20% of $\mathcal{D}$ as a test set $\mathcal{D}_{\text{test}}$, while the remaining 80% of $\mathcal{D}$ are again split into a training set $\mathcal{D}_{\text{train}}$ (82.5% of $\mathcal{D} \setminus \mathcal{D}_{\text{test}}$) and a validation set $\mathcal{D}_{\text{val}}$ (17.5% of $\mathcal{D} \setminus \mathcal{D}_{\text{test}}$). The set of unique keys in $\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}}, \mathcal{D}_{\text{val}}$ are non-overlapping, i.e., when validating and testing a side channel attack, it must reason about keys $\mathbf{k}$ it has never seen during profiling/training.

4.2 Deep Learning SCA

Consider again the problem of estimating $p(v|\boldsymbol{\ell})$ for some variable v. As described in Section **??**, template attacks learn a *generative* model $p(\boldsymbol{\ell}|v)$ and use Bayes' law to recover $p(v|\boldsymbol{\ell})$. In the field of machine learning, *discriminative models* that directly estimate $p(v|\boldsymbol{\ell})$ have shown to outperform generative models on several tasks [**gen_vs_disc**, **crf**]. Therefore, we also experiment with using *Deep Neural Networks* which are trained to directly predict the distribution[2] $p(v|\boldsymbol{\ell})$. *Deep Learning based*

[1] To be precise, we attack exactly the operations shown in Algorithm **??**.

[2] In general, as discussed in Section **??**, we can predict $p(f(v)|\boldsymbol{\ell})$ where f can be chosen from a variety of plausible leakage models. In our experiments, we again only consider the identity function $f(x) = x$ for simplicity.

Side-Channel Attacks (DLSCAs) are not novel and have been extensively studied in the literature and have shown promising results in many applications [**dlsca_ascad**, **breaking_free**, **dlsca_in_practice**, **dlsca_aes**, **dlsca_defcon**]. However, when using a standard *Convolutional Neural Network* (CNN) architecture, we find that in our experiments (detailed in Chapter **??**), the performance of a DLSCA is subpar compared to the template attack pipeline that is described next. Nevertheless, DLSCAs provide particularly interesting flexibilities when performing exact inferences with PCs, which are discussed in the Appendix (Section **??**).

4.3 Template Attack

For a given leakage $\boldsymbol{\ell}$, we utilize the exact setup described in [**5min**] to obtain a set of posterior distributions $p(v_j \mid \boldsymbol{\ell})$ for all intermediate variables $v_j \in \mathbf{v}$. Specifically, during profiling, we use $\mathcal{D}_{\text{train}}$ to compute $u \in \mathbb{N}$ points of interest for each $v_j \in \mathbf{v}$, as described in [**5min**]. Let $\boldsymbol{\ell}'_v \in \mathbb{R}^u$ denote the vector that contains the points of interest of intermediate variable $v \in \mathbf{v}$, extracted from leakage $\boldsymbol{\ell}$.

Next, we use *Linear Discriminant Analysis* (LDA) to reduce the dimensionality of a given $\boldsymbol{\ell}'_v$ [**lda**]. Simply speaking, we find a particular linear projection $\mathbf{A} \in \mathbb{R}^{r \times u}$ into an r-dimensional subspace with $r \ll u$. We denote each projected result as $\mathbf{l}_v = \mathbf{A}\boldsymbol{\ell}'_v$. In our experiments, we choose $u = 512$ and $r = 8$.

Finally, as discussed in Section **??**, we build *Guassian Templates* by estimating parameters for $p(\mathbf{l} \mid v)$ for each value v can assume, i.e., 256 distributions for each intermediate value in our experiments. We can then use Bayes' rule to infer $p(v \mid \mathbf{l})$, while assuming that

$$p(v \mid \mathbf{l}) \approx p(v \mid \boldsymbol{\ell}) \tag{4.1}$$

In subsequent sections, we will slightly abuse notation for consistency and will write $p(v \mid \boldsymbol{\ell})$ to represent $p(v \mid \mathbf{l})$, unless explicitly stated otherwise.

4.4 Evaluation

Recall that, since we attack AES-128, there exists a true 16-byte key, denoted $\mathbf{k}^* = (k_1^*, \ldots, k_{16}^*)^T$, for each leakage $\boldsymbol{\ell}$. Unless stated otherwise, our evaluation uses the exact factor graph presented in Figure **??** to obtain marginal posterior distributions $p(k_1^* \mid \boldsymbol{\ell}), \ldots, p(k_4^* \mid \boldsymbol{\ell})$, given the distributions $p(v_1 \mid \boldsymbol{\ell}), \ldots, p(v_n \mid \boldsymbol{\ell})$ that we obtain using a template attack. As we only attack the first round of AES, we could *independently* attack all 4 invocations of MixColumn and reason about the corresponding 4-byte

subkeys independently. However, we only attack the first 4 key bytes (which are involved in a single MixColumn computation) and extrapolate the result to a full 16-byte key by assuming that the other 3 invocations of MixColumn are equally difficult to attack.

Definition 13 (Rank)**.** Given a probability mass function $p(\mathbf{x})$ over variables $\mathbf{x} \in \mathcal{X}$, and a particular assignment $\mathbf{x}' \in \mathcal{X}$, we define the *rank* of $\mathbf{x}'$ under p as

$$\text{rank}(\mathbf{x}') = |\{\mathbf{x} \in \mathcal{X} \mid p(\mathbf{x}) > p(\mathbf{x}')\}| \tag{4.2}$$

Note that $0 \leq \text{rank}(\mathbf{x}') \leq |\mathcal{X}|-1$. If the underlying probability measure is unambiguous, we will just write *rank of* $\mathbf{x}'$.

In our evaluation of an attack, we assume that the attacker has a fixed compute budget that they use to enumerate key hypotheses. Let $\mathbf{k}_{i:j} = (k_i, k_{i+1}, \ldots, k_j)^T$ (with $i > j$) be a subkey consisting of $j - i + 1$ bytes. Recall that, as in many attacks, we assume

$$p(\mathbf{k}_{1:4} \mid \boldsymbol{\ell}) = p(k_1 \mid \boldsymbol{\ell}) \cdots p(k_4 \mid \boldsymbol{\ell}) \tag{4.3}$$

Therefore, it is straightforward to enumerate the N distinct subkeys $\mathbf{k}_{1:4}$ with the highest proability in the joint $p(\mathbf{k}_{1:4}|\boldsymbol{\ell})$ (for sufficiently small N). In our evaluation, we assume that the other distribution over 4-byte subkeys $p(\mathbf{k}_{5:8}|\boldsymbol{\ell}), p(\mathbf{k}_{9:12}|\boldsymbol{\ell}), p(\mathbf{k}_{13:16}|\boldsymbol{\ell})$ are *identical* to $p(\mathbf{k}_{1:4}|\boldsymbol{\ell})$, i.e., for each 4-byte input, they all output the same probability. For a particular leakage $\boldsymbol{\ell}$, assume that the true subkey $\mathbf{k}^*_{1:4} = (k^*_1, \ldots, k^*_4)^T$ has $\text{rank}(\mathbf{k}^*_{1:4}) < N$. Then, there exists a strategy that takes at most N^4 key hypotheses to obtain the true key $\mathbf{k}^*$, namely enumerating the N most likely 4-byte subkeys for each distribution $\{p(\mathbf{k}_{i:i+3} \mid \boldsymbol{\ell})\}_{i \in \{1,5,9,13\}}$, and trying all N^4 combinations[3].

Definition 14 (Success Rate)**.** Given a single leakage $\boldsymbol{\ell}$ (*single-trace attack*), and the corresponding 16-byte key and plaintext $\mathbf{k}, \mathbf{p}$, we define an attack to be *successful* if $\text{rank}(\mathbf{k}_{1:4}) < 2^8$ under the produced posterior distribution $p(\mathbf{k}_{1:4} \mid \boldsymbol{\ell})$. Under the assumptions detailed above, this corresponds to an attack that can find the full key $\mathbf{k}$ by enumerating at most 2^{32} key hypotheses. The *success rate* is then defined as the fraction of successful attacks on a dataset $\{(\boldsymbol{\ell}^{(i)}, \mathbf{k}^{(i)}, \mathbf{p}^{(i)})\}_{i=1}^n$.

As simple enumeration is often infeasible, other works [**32bit**, **5min**] need to *estimate* the rank of the true key in the resulting posterior distribution using various approaches [**key_enumeration**, **study_key_enum**]. Due to our assumptions, we can directly enumerate the $N = 2^8$ unique keys with the highest probability in $p(\mathbf{k}_{1:4} \mid \boldsymbol{\ell})$ and check if the true key is contained in this set.

[3]Note that this is *not equivalent* to enumerating the top N^4 assignments in the product $\prod_{i \in \{1,5,9,13\}} p(\mathbf{k}_{i:i+3} \mid \boldsymbol{\ell})$.

Definition 15 (Average Rank)**.** For each *successful* attack, we record the actual rank of the true subkey rank($\mathbf{k}_{1:4}$). The mean of these values is denoted as *Average Rank*. Note that this metric does not include the key ranks of unsuccessful attacks.

4.5 Results

As a baseline, we consider the trivial inference problem where we neglect all information present in the MixColumn routine, i.e., we only consider the factor graph in Figure **??**. Clearly, this factor graph does not contain cycles and thus, belief propagation returns the exact key marginals $p(k_1 \mid \boldsymbol{\ell}), \ldots, p(k_4 \mid \boldsymbol{\ell})$.

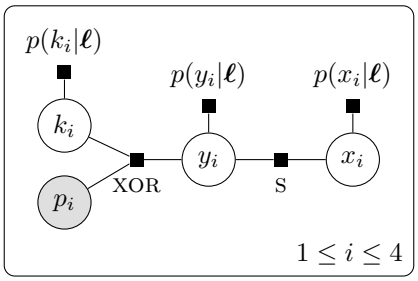


Figure 4.1: Factor graph that corresponds to the reduced inference problem we use as our *Baseline*.

In all of our experiments, we do not construct templates for the key bytes, i.e., when running BP, we always start the algorithm by setting $p(k_i|\boldsymbol{\ell}) = \text{Unif}(\mathbb{F}_2^8)\ \forall i \in \{1, \ldots, 4\}$, where $\text{Unif}(\mathcal{X})$ denotes a uniform distribution over the elements $\mathbf{x} \in \mathcal{X}$. The reason for this choice is that, empirically, we find that directly estimating the distributions $\{p(k_i|\boldsymbol{\ell})\}_{i=1}^4$ using a template attack and using them in the BP algorithm leads to significant performance degradation in all experiments (w.r.t. all metrics considered).

As detailed in Chapter **??**, we compile MixColumn into a PSDD and *circuit multiply* it with PSDD representations of intermediate posteriors. However, to make this computation tractable, we leverage the sparsity of some distributions to substantially reduce the size of the compiled circuits. Therefore, we choose to sparsify all distributions of inputs $p(v|\boldsymbol{\ell}), v \in \mathbf{v}_{in}$ since, empirically, we observe that these distributions have low entropy:

$$\tilde{\mu}_{\mathcal{M}_B \to x_i}(x_i) = \sum_{\mathbf{v} \setminus x_i} \mathcal{PC}\left(\mathcal{M}_B(\mathbf{v})\right) \cdot \prod_{v_i \in \mathbf{v}_{in}} \mathcal{PC}\left(\tilde{p}_\pi(v_i|\boldsymbol{\ell})\right) \cdot \prod_{v_j \in \mathbf{v} \setminus \mathbf{v}_{in}} \mathcal{PC}\left(p(v_j|\boldsymbol{\ell})\right) \tag{4.4}$$

where $\tilde{p}_\pi(v_i|\boldsymbol{\ell})$ is the π-sparse approximation of $p(v_i|\boldsymbol{\ell})$, as described in Section **??**. As an optimization, note that $\tilde{\mu}_{\mathcal{M}_B \to x_i}(x_i)$ already includes the distribution $p(x_i|\boldsymbol{\ell})$ in the multiplication sequence and thus, we omit multiplying with this distribution during

BP. This decreases the size of the resulting circuit during multiplication. Apart from this, we perform vanilla BP in the factor graph shown in Figure **??** and repeat the experiment with different values of $\varepsilon = 1 - \pi$. Since we use the PSDDs to perform marginalization, we denote this method as PSDD + MAR (*marginalization*).

With the same sparse approximations, i.e., $\tilde{p}_\pi(v|\boldsymbol{\ell}),\ \forall v \in \mathbf{v}_{in}$, we also use the algorithm presented in Section **??** to compute the top-256 most probable key assignments directly in the joint distribution, denoted as PSDD + MPE (*most probable evidence*). Recall that an MPE query is a linear time operation in decomposable and deterministic PCs, which particularly motivates our choice of using PSDDs to model this inference problem. Finally, we compare both the baseline and the PSDD approaches with 3 SASCAs with different numbers of loopy BP iterations (namely, $3, 50$ and 100 iterations) on the factor graph shown in Figure **??**. To analyze the influence of sparsifying $p(v|\boldsymbol{\ell}), v \in \mathbf{v}_{in}$, we also run the SASCAs with the sparse approximations (with different ε, where $\varepsilon = 0$ means *no sparsification*). The results are depicted in Table **??** (rounded to two decimal places).

	Success Rate			
Inference Method	$\varepsilon = 10^{-2}$	$\varepsilon = 10^{-5}$	$\varepsilon = 10^{-8}$	$\varepsilon = 0$
Baseline	79.35%	79.57%	79.59%	**79.59**%
SASCA (3 BP iterations)	84.89%	84.88%	84.91%	**84.91**%
SASCA (50 BP iterations)	89.36%	90.45%	90.41%	**90.45**%
SASCA (100 BP iterations)	89.36%	90.27%	**90.69**%	90.52%
PSDD + MAR	93.40%	97.81%	**98.02**%	N/A
PSDD + MPE	93.67%	99.42%	**99.93**%	N/A

Table 4.1: Success rate of different inference methods, measured on the validation dataset $\mathcal{D}_{\text{val}}$. $\varepsilon = 1 - \pi$ controls the level of sparsity in the approximated distributions, where $\varepsilon = 0$ means no approximation. For the Baseline and SASCA, we can tractably compute results for $\varepsilon = 0$, while this is not possible using the PSDD approaches (denoted by N/A).

After finding the best ε for the PSDD approaches ($\varepsilon = 10^{-8}$), we use this value to run evaluation on our test set $\mathcal{D}_{\text{test}}$. We also report *average rank* for each method and show the results in Table **??**.

Inference Method	**Success Rate**	**Average Rank**
Baseline	79.21%	21.84
SASCA (3 BP iterations)	84.65%	16.62
SASCA (50 BP iterations)	90.50%	6.64
SASCA (100 BP iterations)	90.35%	6.17
PSDD + MAR ($\varepsilon = 10^{-8}$)	98.04%	1.83
PSDD + MPE ($\varepsilon = 10^{-8}$)	**99.89**%	**0.34**

Table 4.2: Success rate and average rank measured on the test dataset $\mathcal{D}_{\text{test}}$. All SASCAs were performed with $\varepsilon = 0$.

As seen in Table **??**, performing exact inference on approximations of the distributions substantially outperforms SASCA: Not only do we observe an increase in success rate of up to 9%, but also severely reduce the number of key hypotheses we need to enumerate during an attack (Average Rank). In absolute terms, we see that the PSDD + MPE approach can effectively recover almost all secret keys with a single trace.

4.5.1 Isolation of the Loopy Subgraph

Recall that we only replace the loopy parts of the factor graph in Figure **??** with a PSDD factor. This allows us to perform BP on an acyclic factor graph with high-dimensional factors, which effectively shifts most of the inference burden from message passing to circuit operations. Since we leave the remaining subgraph unchanged, both the SASCA, as well as our PSDD + MAR approach partially perform BP on the same graph. To study the effect of BP on the loopy subgraph in isolation, we run experiments that are equivalent to the ones above, except for setting $p(y_i|\boldsymbol{\ell}) = \text{Unif}(\mathbb{F}_2^8)\ \forall i \in \{1, \ldots, 4\}$. This restricts the inference methods to use only leakages in the MixColumn routine (except for the plaintext addition). The respective results are shown in Table **??**.

Inference Method (MixColumns only)	**Success Rate**			
	$\varepsilon = 10^{-2}$	$\varepsilon = 10^{-5}$	$\varepsilon = 10^{-8}$	$\varepsilon = 0$
Baseline	79.35%	79.57%	79.59%	**79.59**%
SASCA (3 BP iterations)	81.28%	82.26%	82.30%	**82.30**%
SASCA (50 BP iterations)	86.82%	**88.35**%	88.10%	88.15%
SASCA (100 BP iterations)	87.19%	88.25%	**88.46**%	88.32%
PSDD + MAR ($\varepsilon = 10^{-8}$)	93.27%	97.75%	**98.08**%	N/A
PSDD + MPE ($\varepsilon = 10^{-8}$)	93.67%	99.42%	**99.93**%	N/A

Table 4.3: Success rate on $\mathcal{D}_{\text{val}}$ when setting $p(y_i|\boldsymbol{\ell}) = \text{Unif}(\mathbb{F}_2^8)\ \forall i \in \{1, \ldots, 4\}$.

5 Discussion

5.1 Interpretation of Experimental Results

The measurements we conduct using the validation dataset (Table **??**) and the test dataset (Table **??**) both show that the PSDD approaches that perform *exact* inference on an approximation of the inference problem ($\varepsilon > 0$) significantly and systematically outperform both SASCA and the baseline in terms of success rate and average rank. By also running SASCA and the baseline using the sparsified distributions, we isolate the influence of *approximate inference* and observe that, usually, the performance of all methods becomes worse as we increase ε. In other words, loopy BP does not benefit from sparse approximations of the distribution factors in our experiments. Thus, we conclude that it is indeed the fact that we perform exact inference that is responsible for the increase in performance.

When performing a SASCA, we tune the number of loopy BP iterations using $\mathcal{D}_{\text{val}}$ and find that a larger number of iterations is generally advantageous in terms of success rate. However, as discussed in Section **??**, this clearly increases the computational cost of the attack. Moreover, an effect of *diminishing returns* can be observed: While 50 BP iterations yield much better results than merely 3 iterations, 100 iterations increase the success rate only marginally.

Recall that, by definition, every successful attack has a bounded computational cost: At most 2^{32} key hypotheses are required to find the true 16-byte key if the attack was successful.[1] However, our experiments show that most attacks require orders of magnitude fewer operations: Using the test set $\mathcal{D}_{\text{test}}$, successful attacks using our PSDD + MPE ($\varepsilon = 10^{-8}$) approach need, on average, $\approx 150,000$ key hypotheses to find the true 16-byte key (i.e., brute-forcing ≈ 17.2 bits). For comparison, these are ≈ 150 times fewer key hypotheses than the best SASCA (100 BP iterations) on $\mathcal{D}_{\text{test}}$.

Interestingly, when only performing inference using leakages from the MixColumn routine (Table **??**), we observe that only the SASCAs benefit from learning the distributions $p(y_i|\boldsymbol{\ell})$. All other methods perform equally well when neglecting information about y_i. We also conclude that a lot of information about the key is already encoded

[1] Under the assumptions detailed in Section **??**.

in $p(x_i \mid \boldsymbol{\ell})$, as only using these distributions (baseline) to predict the key already yields a success rate of 79.59%.

5.2 Marginalization vs. MPE

Recall that in the PSDD + MAR approach, after performing many circuit multiplications, we use the resulting PSDD to compute a marginal (i.e., the BP message in **??**). Then, we perform BP to obtain byte marginals $p(k_i \mid \boldsymbol{\ell})$. However, as we wish to enumerate entire 4-byte keys, we assume that

$$p(k_1, \ldots, k_4 \mid \boldsymbol{\ell}) = \prod_{i=1}^{4} p(k_i \mid \boldsymbol{\ell}) \tag{5.1}$$

i.e., we assume that the key bytes are *conditionally independent*, given $\boldsymbol{\ell}$. However, as soon as $\boldsymbol{\ell}$ captures information about variables that depend on more than a single key byte, this assumption is, in general, incorrect. In our use case, $\boldsymbol{\ell}$ includes leakages from MixColumn, which mixes a transformed version of *different* key bytes.

In contrast, the PSDD + MPE approach does not make this assumption, as it *directly* extracts the N most probable key assignments using the actual joint distribution $p(k_1, \ldots, k_4 \mid \boldsymbol{\ell})$ induced by the factor graph. We conclude that, if possible, directly querying the joint distribution is beneficial for the performance of the side-channel attack.

However, it is not straightforward to extend our presented MPE inference approach to more general settings: For example, in the case where we attack multiple traces, we would need to represent the instantiation of MixColumn as a PSDD for every trace and multiply *all* of them to obtain a single PSDD. Moreover, we would need to represent all other factors (SBOX, XOR) as PSDDs and also circuit multiply them with the previous result — an endeavor which is, without further assumptions, most probably computationally infeasible in practice. Even in the single-trace setting, if we compile multiple parts of a factor graph into separate PSDDs, we need to circuit multiply all of them to obtain a PC that can answer our MPE query tractably, which can be computationally difficult.

5.3 PSDD vs. Naïve Loop

To make circuit multiplication in Equation **??** tractable, we construct π-sparse approximations of $p(v \mid \boldsymbol{\ell}),\ \forall v \in \mathbf{v}_{in}$, since we empirically observe that these distributions have low entropy in our particular use case. In fact, under this assumption, we can

also use a naïve algorithm that generates the same output as a PSDD: Recall that a simple loop can compute

$$\mu_{\mathcal{M}_B \to x_i}(x_i) = \sum_{\mathbf{v} \in \mathcal{V}_i(x_i)} \prod_{v_j \in \mathbf{v} \setminus x_i} p(v_j | \boldsymbol{\ell}) \tag{5.2}$$

with

$$\mathcal{V}_i(y) = \{\mathbf{v} \mid \mathcal{M}_B(\mathbf{v}) = 1 \wedge x_i = y\} \tag{5.3}$$

for all $x_i \in \{0, \ldots, 255\}$ in a total of 2^{32} loop iterations. However, if $\tilde{p}_\pi(\mathbf{v}_{in} | \boldsymbol{\ell}) = \prod_{v \in \mathbf{v}_{in}} \tilde{p}_\pi(v | \boldsymbol{\ell})$ is sparse, we only need to consider $\mathbf{v}_{in}$ that have positive probability: Let

$$\tilde{\mathcal{V}}_i(y) = \{\mathbf{v} \mid \mathcal{M}_B(\mathbf{v}) = 1 \wedge x_i = y \wedge \tilde{p}_\pi(\mathbf{v}_{in} \mid \boldsymbol{\ell}) > 0\} \tag{5.4}$$

and let $nnz(p(x))$ denote the number of nonzeros in the probability mass function $p(x)$. Then, we can compute $\mu_{\mathcal{M}_B \to x_i}(x_i)$ for all $x_i \in \{0, \ldots, 255\}$ using $\prod_{v \in \mathbf{v}_{in}} nnz(\tilde{p}_\pi(v | \boldsymbol{\ell}))$ loop iterations by calculating

$$\mu_{\mathcal{M}_B \to x_i}(x_i) = \sum_{\mathbf{v} \in \tilde{\mathcal{V}}_i(x_i)} \prod_{v_j \in \mathbf{v} \setminus x_i} p(v_j | \boldsymbol{\ell}) \tag{5.5}$$

Note that we can easily compute $\tilde{\mathcal{V}}_i(y)$ by using an implementation of MixColumn: We enumerate all $\mathbf{v}_{in}$ that have positive probability and run MixColumn to produce $\mathbf{v}_{mid}, \mathbf{v}_{out}$. We can also use a similar approach to reproduce the output of the PSDD + MPE method: After performing left-to-right message passing as detailed in Section **??**, we again enumerate all $\mathbf{v}_{in}$ with positive probability. For each such $\mathbf{v}_{in}$, we calculate the corresponding $\mathbf{v}_{mid}, \mathbf{v}_{out}$, and, with $\mathbf{v} = (\mathbf{v}_{in}, \mathbf{v}_{mid}, \mathbf{v}_{out})^T$, compute

$$f(\mathbf{v}) = \prod_{v_j \in \mathbf{v} \setminus \mathbf{v}_{in}} p(v_j | \boldsymbol{\ell}) \cdot \prod_{i=1}^{4} \mu_{x_i \to \mathcal{M}_B}(x_i) \tag{5.6}$$

Then, take the N $\mathbf{v}$ assignments that achieve the highest score[2] $f(\mathbf{v})$ and, as described in Section **??**, compute the corresponding N key hypotheses. As before, this algorithm needs $\prod_{v \in \mathbf{v}_{in}} nnz(\tilde{p}_\pi(v | \boldsymbol{\ell}))$ loop iterations.

Both methods also work if, instead of assuming that the distributions of $\mathbf{v}_{in}$ have low entropy, we assume that the distributions of *outputs* $p(v | \boldsymbol{\ell}), v \in \mathbf{v}_{out}$ have low entropy: We enumerate all $\mathbf{v}_{out}$ with positive probability and use the *inverse* MixColumn^{-1} to derive the corresponding inputs $\mathbf{v}_{in}$ and intermediates $\mathbf{v}_{mid}$.

However, as these methods again neglect the *internal structure* of MixColumn,

[2] We break ties randomly.

it is less straightforward to apply them if (1) we want to exploit a combination of low entropy distributions over $\mathbf{v}_{in}$ *and* over $\mathbf{v}_{out}$, or (2) we wish to leverage low entropy distributions of variables $v \in \mathbf{v}_{in}$, or (3) both. For example, assume that we have a single low entropy distribution over some input $v_i \in \mathbf{v}_{in}$, two low entropy distributions over intermediates $v_{m1}, v_{m2} \in \mathbf{v}_{mid}$, and a single low entropy distribution over some output $v_o \in \mathbf{v}_{out}$. After sparsifying these distributions, we can first multiply the sparsified distributions with $\mathcal{PC}(\mathcal{M}_B)$ in the PSDD multiplication chain, which exploits the sparsity of *all* distributions simultaneously.[3] The naïve loop approach detailed above cannot fully leverage this combination of sparse distributions. However, we note that in our particular use case, we can describe MixColumn by a set of *linear equations*, since the algorithm only consists of linear operations (xor, xtime). This way, we can use a solver to overcome the challenges we have just described. Regardless, this naïve algorithm is difficult to extend to general, non-linear algorithms, while the PSDD representation can natively exploit the discussed combinations of sparse distributions.

5.4 Runtime

We explicitly note that we do not employ sophisticated profiling techniques and that the implementation of most algorithms we provide should be viewed as research artifacts and their performance could be significantly improved with sufficient engineering effort. Nevertheless, attack runtime is a crucial part of our evaluation: As discussed above, a trivial algorithm can perform exact probabilistic inference — albeit at the cost of computational complexity. All SASCAs were performed using the SCALib [**scalib**] library, which efficiently implements BP using the *Rust* programming language [**scalib**, **rust**]. A single SASCA takes tens to hundreds of milliseconds on a modern laptop CPU, depending on the number of BP iterations. Changes to the sparsity parameter ε do not influence the runtime significantly.

The runtime of our PSDD methods consists of an offline SDD compilation step (< 10 seconds on a modern CPU) that is performed *once*, and an online attack phase, whose performance heavily depends on ε: On a modern laptop CPU, a single attack takes hundreds of milliseconds for $\varepsilon = 10^{-2}$ and seconds for $\varepsilon \in \{10^{-5}, 10^{-8}\}$. The main computational bottleneck of these attacks is the sequence of circuit multiplications, which are based on open-source C++ implementations[4] that mostly operate in a sequential manner and do not leverage parallelism (e.g., to compute multiple inference problems simultaneously). We also run the naïve loop implementations of these attacks described in Section **??** and observe that a highly-parallel `NumPy` [**numpy**]

[3] We validate this claim by randomly generating synthetic (sparse) PMFs and running the circuit multiplication chain.

[4] `https://github.com/hahaXD/psdd_nips`, `https://github.com/hahaXD/psdd`

implementation yields competitive runtimes compared to SASCA (tens of milliseconds on a modern laptop CPU).

5.5 SDD Compilation

There are two main challenges in our PSDD approaches that depend on each other: (1) Succinctly compiling $\mathcal{M}_B$ into an SDD, and (2) tractably computing the sequence of PSDD multiplications. To demonstrate the relationship between these tasks, assume that we could succinctly compile $\mathcal{M}_B$ into an SDD (using dynamic vtree minimization [**dynamic_min_choi**]) that respects a vtree v such that for all bytes $v_i \in \mathbf{v}$, there exists an internal vtree node v' that has *exactly* the bits of v_i under it (namely $v_i^{(1)}, \ldots, v_i^{(8)}$). In other words, the resulting SDD represents a boolean formula that disentangles all bytes $v_i \in \mathbf{v}$. Given such an SDD, even without sparse approximations, the chain of PSDD multiplications can trivially be computed, since for any $v_i \in \mathbf{v}$, the multiplication $\mathcal{PC}\left(\mathcal{M}_B(\mathbf{v})\right) \cdot \mathcal{PC}\left(p(v_i|\boldsymbol{\ell})\right)$ yields a circuit of the same size as $\mathcal{PC}\left(\mathcal{M}_B(\mathbf{v})\right)$.

Unfortunately, compiling such an SDD is infeasible in practice, since the individual bits of *different* bytes are heavily entangled in the MixColumn routine. While in this work, we utilize a compiler that solely optimizes the *size* of the compiled SDD [**dynamic_min_choi**], future work can strike a trade-off between these tasks by building compilers that optimize both of these objectives simultaneously.

6 Conclusions and Future Work

In this work, we introduce a novel method that, given sparse approximations of posterior distributions (e.g., obtained by template attacks), utilizes PCs to perform exact probabilistic inference in the context of side-channel attacks. Moreover, if the posterior distributions happen to be on the scope of individual bits (as in some attacks in the literature, see [**bit_posterior1**, **bit_posterior2**]), we show that exact inference is even tractable *without* approximations of the posterior distributions.

While our results demonstrate significant improvements over state-of-the-art methods, many fruitful research directions remain, including:

Inference in the joint key distribution. Based on our results, we additionally conclude that it is beneficial for attack performance to avoid the approximation of the full joint key posterior with a product of byte marginals. Since this is sometimes computationally infeasible, investigating other simplifications of the joint distribution (e.g. by means of sparsification or less strict conditional independence assumptions) is a promising future research direction.

CNF Representation. A key challenge of this work is to compile the algorithmic description of a cryptographic routine into a succinct SDD representation. To do so, we first construct a boolean representation in Conjunctive Normal Form (CNF) and use existing SDD compilation techniques that consume this CNF. However, we claim that CNFs are an inherently *unnatural* intermediate representation of the algorithm: (1) Although the goal is a hierarchical circuit representation, CNFs are *flat* descriptions that do not leverage any kind of decision hierarchy, and (2) even simple cryptographic algorithms can sometimes only be represented using large CNFs, especially since XOR operations suffer from an exponential blowup in the number of clauses if they are embedded in CNF [**asca**]. The latter problem is well-known and has been addressed in other fields: For example, *MiniSAT* [**minisat**], a popular SAT solver, was extended to explicitly handle XOR operations more efficiently (*CryptoMiniSAT* [**cryptominisat**]). Future work may thus consider other normal forms, such as *Algebraic Normal Form* (ANF), which are more suitable for cryptographic algorithms and may investigate building dedicated SDD compilers for these languages.

Holistic SDD Compilation. A second, orthogonal issue is the fact that the SDD compiler is oblivious to the downstream circuit multiplication tasks: The compiler performs a search through the space of vtrees to heuristically optimize the size of

the resulting SDD [**dynamic_min_choi**]. Future work may extend the compiler to optimize a *multi-goal* objective, i.e., to search for vtrees that both (1) induce a compact SDD representation, and (2) are suitable for the sequence of circuit multiplications (as discussed in Section **??**).

Sparsity-aware Vtree Search. Before compiling the SDD, we already possess knowledge about *which* posterior distributions we wish to sparsify. Hence, the vtree search procedure can incorporate this information and prioritize vtrees where bits that occur in sparse distributions are close to the root of the vtree: From a top-down perspective, this encourages the induced SDD to quickly make decisions based on the bits for which only few possible assignments exist. Multiplying the circuits that represent the sparse distributions will thus prune large parts of the compiled circuit.

Order of PSDD Multiplication Future work may also systematically investigate the influence of the *order* in the circuit multiplication sequence on the runtime of the attack: In this work, we conduct our experiments by multiplying circuits in the natural order induced by the vector $\mathbf{v}$ (except for sparse distributions, which we always multiply first). We also experiment with different orders, such as the one presented in [**tractable_ops**], and find slight differences in runtime.

Neuro-Symbolic Learning Our experiments do not fully explore the flexibility of DLSCA to train networks that are both (1) expressive and (2) amenable to exact inference by learning to parameterize a (compatible) single PC that is subsequently multiplied with the constraint circuit $\mathcal{PC}(\mathcal{M}_B(\mathbf{v}))$, as shown in [**spl**]. Future work may study architectures to trade off learning difficulty and expressivity of the parameterized circuit, all while producing a compatible circuit.

Appendix

Appendix A

Deep Learning Side Channel Attacks

A.1 Connections to Neuro-Symbolic Learning

On a high level, recall that we use PSDDs to perform tractable inference (e.g., marginalization or maximization) in the circuit product specified by

$$\underbrace{\mathcal{PC}(\mathcal{M}_B(\mathbf{v}))}_{\text{Symbolic Knowledge}} \cdot \prod_{v_i \in \mathbf{v}} \underbrace{\mathcal{PC}(p(v_i \mid \boldsymbol{\ell}))}_{\text{Learned from Data}} \tag{A.1}$$

We can think of $\mathcal{M}_B$ as symbolic knowledge, while the distributions $p(v_i \mid \boldsymbol{\ell})$ are learned from data in some way (e.g., template attacks or DLSCA). This formulation is reminiscent of recent works in *neuro-symbolic learning*, which focuses on bridging the fields of subsymbolic and symbolic artificial intelligence [**spl**]. For example, Ahmed et al. have recently proposed *Semantic Probabilistic Layers* (SPLs), which guarantee the output of neural networks to satisfy a constraint in a structured-output prediction problem [**spl**]. One of their methods utilizes a PC to represent symbolic constraints via knowledge compilation techniques while using deep neural networks to output the parameters of a second PC, which are both multiplied together to constrain the support of the latter PC to assignments that satisfy the constraints [**spl**].

This clearly shows the similarities between our inference approach and SPL, especially when we think about $\prod_{v_i \in \mathbf{v}} \mathcal{PC}(p(v_i|\boldsymbol{\ell}))$ as a *single* PC whose parameters could directly be the output of a neural network. While this circumvents the computational problem of multiplying many circuits, we find that the resulting learning problem is vastly more difficult than first learning the individual distributions $p(v_i|\boldsymbol{\ell})$. Thus, we only report experiments where we train on the latter task.

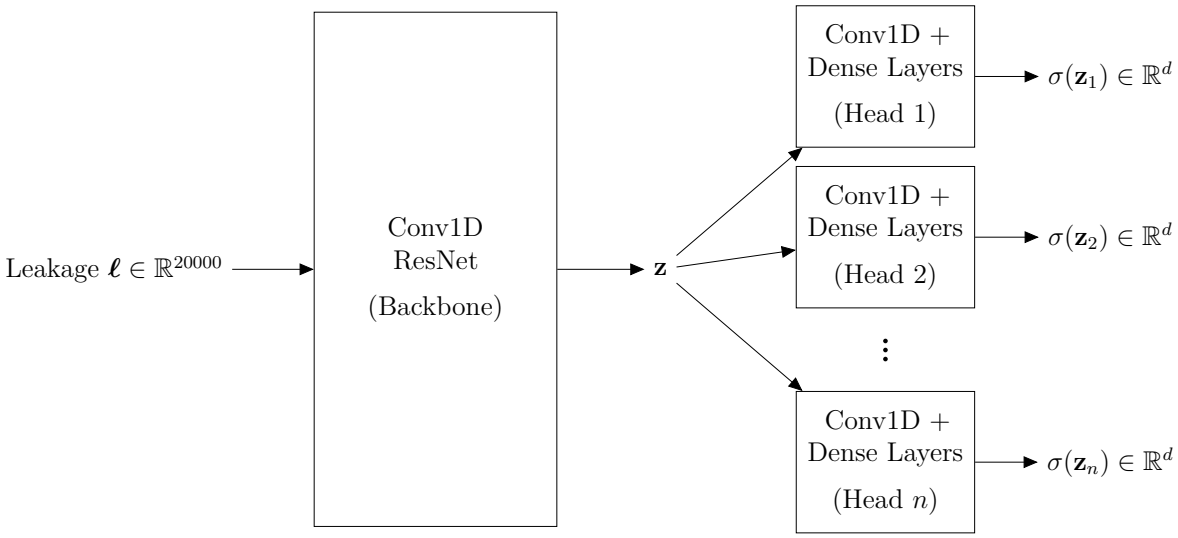


Figure A.1: We use a shared backbone to produce an embedding $\mathbf{z} \in \mathbb{R}^{256\times625}$, which is passed to n heads, each producing an output $\sigma(\mathbf{z}_i) \in \mathbb{R}^d$. We run two experiments: (1) We set $\sigma(\mathbf{z}_i) = \text{softmax}(\mathbf{z}_i)$ and $d = 256$ (*byte distributions*), and (2) we set $\sigma(\mathbf{z}_i) = \text{sigmoid}(\mathbf{z}_i)$ and $d = 8$ (*bit distributions*).

A.2 Experimental Setup

The general neural network architecture is shown in Figure **??**, is implemented in *PyTorch* [**pytorch**] and largely follows the open-source implementation[1] of [**dlsca_defcon**]. Particularly, our architecture can be classified as a *Convolutional Neural Network* (CNN) [**cnn_fukushima**, **cnn_lecun**], as we make heavy use of 1D convolutional layers (`Conv1d`). Next, we describe the model architecture in more detail.

Since the *Backbone* consumes a high-dimensional power trace $\boldsymbol{\ell}$, we employ downsampling strategies to reduce the dimensionality of the input data: First, we utilize *maximum pooling* (i.e., a `MaxPool1d` layer) with a kernel size of 4, a stride of 4, and a dilation of 1. Next, we run the output of this operation to a stack of 12 *Blocks*, where each block consists of 3 sequences of a `Conv1d` layer, followed by a *Batch Normalization* layer [**batchnorm**] (`BatchNorm1d`) and the ReLU activation function, defined as $\text{ReLU}(x) = \max(0, x),\ x \in \mathbb{R}$ (applied elementwise). Moreover, each block uses a *residual connection* as introduced in [**resnet**], i.e., we add the input a block receives to the output it returns. However, we slightly modify this setup and add another `Conv1d` layer to the residual connection (i.e., the input flows through a convolutional layer before being added to the output), as we find that this improves performance empirically. As common in the literature [**cnn_lecun**], we keep increasing the number of filters in the convolutional layers, while decreasing the dimensionality of each feature map using strides > 0. The final block outputs the embedding $\mathbf{z} \in \mathbb{R}^{256\times625}$ with 256

[1] `https://github.com/google/scaaml`

feature maps, each consisting of 625 values. This embedding is then passed to the individual *Heads*: Each head first runs $\mathbf{z}$ through 3 blocks (as defined above), followed by performing average pooling to produce a 512-dimensional embedding, which is passed to a network consisting of 2 fully connected layers (*Dense* layers, often called *Linear* layers) with ReLU activation. After the first dense layer, we again employ batch normalization using `BatchNorm1d`. The second layer outputs $\sigma(\mathbf{z}_i) \in \mathbb{R}^d$ where $d = 256$ and $\sigma(\mathbf{z}_i) = \text{softmax}(\mathbf{z}_i)$ in our first experiment (*byte distributions*) and $d = 8$ and $\sigma(\mathbf{z}_i) = \text{sigmoid}(\mathbf{z}_i)$ in our second experiment (*bit distributions*). In both cases, we create $n = 21$ heads since we wish to model the posterior distributions $p(v|\boldsymbol{\ell})$ of 21 variables $v \in \mathbf{v}$.

Definition 16 (Softmax Activation Function)**.** With $\mathbf{z} = (z_1, \ldots, z_d)^T \in \mathbb{R}^d$, we define

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{d} \exp(z_j)} \tag{A.2}$$

for $i \in \{1, \ldots, d\}$. Note that $\text{softmax}(\mathbf{z}) \in \mathbb{R}^d$ is a PMF for all $\mathbf{z} \in \mathbb{R}^d$.

Definition 17 (Sigmoid Activation Function)**.** With $\mathbf{z} = (z_1, \ldots, z_d)^T \in \mathbb{R}^d$, we define

$$\text{sigmoid}(\mathbf{z})_i = \frac{1}{1 + \exp(-z_i)} \tag{A.3}$$

for $i \in \{1, \ldots, d\}$. As $0 < \text{sigmoid}(\mathbf{z})_i < 1$ for all $\mathbf{z} \in \mathbb{R}^d$, we can interpret $\text{sigmoid}(\mathbf{z})_i$ as the success parameter of a Bernoulli distribution.

As a regularizer, we set each input value of the final fully connected layer to 0 with probability $p = 0.1$ during training (*Dropout*). Both network architectures consist of ≈ 30 million trainable parameters, which we fit by using the *Adam* optimizer [**adam**] with a learning rate of 0.001 to minimize the mean of the *sample losses* over $\mathcal{D}_{\text{train}}$: Given $\boldsymbol{\ell}$, the sample loss is the average cross entropy between the distributions the network outputs (i.e., $\{\sigma(\mathbf{z}_i)\}_{i=1}^n$) and the empirical distributions $p_e(v_i|\boldsymbol{\ell})$, which output 1 if v_i assumes the true value of the corresponding intermediate, and 0 else. When $\sigma(\mathbf{z}_i) = \text{sigmoid}(\mathbf{z}_i)$, we use the *binary* cross entropy loss function. For both models, we train for 30 epochs using a batch size of 16 and use $\mathcal{D}_{\text{val}}$ to evaluate the validation loss after every epoch. We save a snapshot of the current best model (w.r.t. validation loss) to disk after every epoch and use the overall best model to perform our experiments.

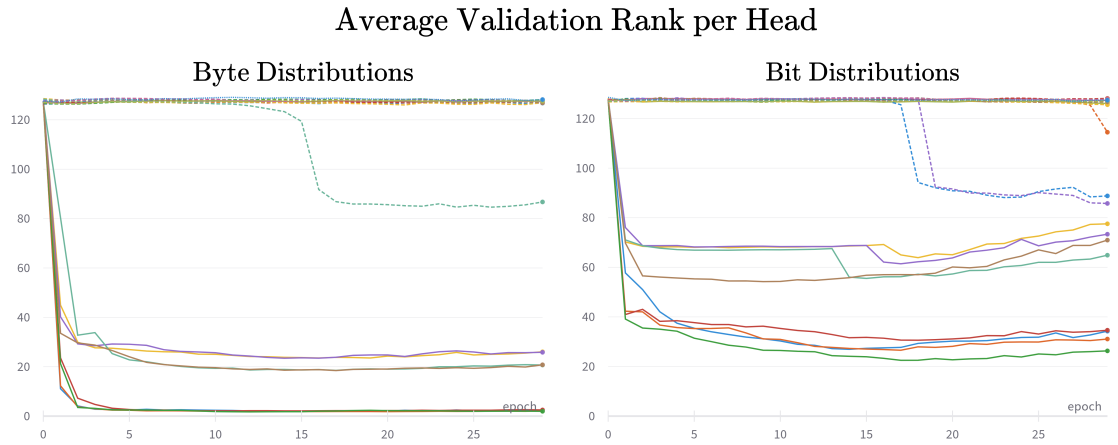


Figure A.2: During training, we monitor the average rank of each of the true intermediates $v \in \mathbf{v}$ in the corresponding output distribution, measured on the validation set $\mathcal{D}_{\text{val}}$. We observe that the fully factorized bit distributions lack the expressivity to faithfully represent the posterior distributions.

A.3 Experimental Results and Discussion

Figure **??** shows the average validation rank per head during training. We observe that some intermediates are much easier to accurately predict than others: The four heads predicting $p(v|\boldsymbol{\ell})$ for $v \in \mathbf{v}_{in}$ show the lowest average validation ranks in both experiments, while many other distributions yield an average rank of ≈ 127.5. This is the expected rank if the position of the true value is uniformly random — effectively providing no information about the true intermediate.

More precisely, if we first use $\mathcal{D}_{\text{val}}$ to calculate the mean rank of true input values $v \in \mathbf{v}_{in}$ under the respective distribution the neural network predicts, i.e., $p(v|\boldsymbol{\ell})$, and average these 4 means, we obtain an average validation rank of 2.21 in the *byte distributions* experiment, and an average validation rank of 31.58 in the *bit distributions* experiment.

We repeat the set of main experiments in Section **??**, but instead of obtaining the posterior distributions $p(v|\boldsymbol{\ell})$ via template attacks, we use our neural networks to predict these distributions. In the *byte distributions* experiment, we again have to sparsify the distributions $p(v|\boldsymbol{\ell})$ $\forall v \in \mathbf{v}_{in}$ to make the PSDD computations tractable (i.e., $\varepsilon > 0$). Note that the experimental results that follow have been obtained with $\varepsilon \in \{10^{-2}, 10^{-3}\}$, as smaller values of ε made the experiments intractable. However, as explained in Section **??**, we do not need any sparsification in the *bit distribution* experiment, since PSDD multiplication is trivial when performed with circuits that represent fully-factorized Bernoulli distributions. In order to compute a baseline and run SASCA using the predicted *bit distributions* (i.e., Bernoulli parameters), we first

evaluate the fully-factorized distributions for all bitstrings $x \in \mathbb{F}_2^8$ and pass the resulting PMF (with the scope of a byte, respectively) to SASCA and to the computation of the baseline (as defined in Section **??**). The results of the experiments are shown in Table **??**.

Inference Method (MixColumns only)	**Byte Distributions**		**Bit Distributions**	
	Success Rate	Avg. Rank	Success Rate	Avg. Rank
Baseline	43.62%	41.54	1.14%	45.15
SASCA (3 BP it.)	42.65%	42.30	1.24%	47.28
SASCA (50 BP it.)	61.53%	18.13	2.06%	48.82
SASCA (100 BP it.)	61.49%	18.48	2.06%	47.71
PSDD + MAR ($\varepsilon = 10^{-2}$)	83.92%	7.15	/	/
PSDD + MAR ($\varepsilon = 10^{-3}$)	84.78%	7.34	/	/
PSDD + MAR ($\varepsilon = 0$)	N/A	N/A	2.70%	44.40
PSDD + MPE ($\varepsilon = 10^{-2}$)	92.31%	1.60	/	/
PSDD + MPE ($\varepsilon = 10^{-3}$)	**97.03**%	2.12	/	/
PSDD + MPE ($\varepsilon = 0$)	N/A	N/A	**10.75**%	78.40

Table A.1: Success rate and average rank (as defined in Section **??**), measured on $\mathcal{D}_{\text{test}}$. Since we only model distributions of intermediate values in MixColumn, these experiments can be compared to Table **??** where we set $p(y_i|\boldsymbol{\ell}) = \text{Unif}(\mathbb{F}_2^8)\ \forall i \in \{1, \ldots, 4\}$. As we do not need approximation techniques to perform exact inference using the *bit distributions*, we do not run experiments with $\varepsilon > 0$ (denoted by /). Again, we denote experiments that we cannot tractably perform with N/A.

Interestingly, when focusing on the *byte distributions* experiment, the performance gap between SASCA and the PSDD approaches is substantially larger than the gap in our main experiments: PSDD + MPE ($\varepsilon = 10^{-3}$) can successfully attack 97.03% of test traces — amounting to $\approx 35\%$ *more* successful attacks than the best SASCA approach.

As discussed above, the network that outputs *bit distributions* is not capable of expressing the *true* posterior distributions $p(v|\boldsymbol{\ell}),\ v \in \mathbf{v}$ well. We conclude that the assumption that these distributions factorize into individual Bernoullis is thus flawed in our attack scenario. Nevertheless, given the *bit distributions*, we can perform exact inference without the need for sparsification: PSDD + MPE ($\varepsilon = 0$) can still successfully recover 10.75% of the keys — amounting to more than 5 times as many successful attacks when compared to the best SASCA. Moreover, compared to PSDD + MAR, our results show that it is particularly beneficial to perform MPE queries against the joint in this scenario.

Appendix B

Algorithms

B.1 Tractable Circuit Multiplication

Algorithm **??** shows a recursive routine that takes two PSDDs n_1, n_2 and the vtree v they both respect as input and outputs a PSDD n and a constant κ such that

$$\forall \mathbf{x} : p(\mathbf{x}) = \kappa^{-1} \cdot p_1(\mathbf{x}) \cdot p_2(\mathbf{x}) \tag{B.1}$$

where $p_1(\mathbf{x}), p_2(\mathbf{x})$ denote the distributions represented by n_1, n_2, respectively, and $p(\mathbf{x})$ denotes the distribution represented by the ouput PSDD n. The algorithm uses a cache since a PC is not necessarily a tree, but a directed acyclic graph (DAG).

Recall that, w.l.o.g., every PSDD sum node has one or many product node children, each of which computes the product of its left child (prime) and its right child (sub). Thus, the children of every sum node m can be viewed as a set of prime-sub pairs p, s with corresponding weights w, i.e.,

$$\text{children}(m) = \{(p_i, s_i, w_i)\}_{i=1}^{k} \tag{B.2}$$

where k is the number of children of the m. In Algorithm **??**, we recursively traverse sum nodes of the PSDD and implicitly deal with product nodes using the notion of prime-sub-weight triplets we have just introduced.

B.2 PMF to PSDD Compilation

Algorithm **??** depicts a recursive routine for compiling an arbitrary PMF p to a PSDD $\mathcal{PC}(p)$ that respects a given vtree v. To start the compilation, we call res $\leftarrow$ COMPILE_PMF(v, p, True) and find the root node of $\mathcal{PC}(p)$ in res.node. Merging multiple sum nodes (with a single child) into a single sum node is achieved as follows:

We collect all children and attach them to a single sum node, where the weight of each child is given by the *weight* parameter in the corresponding node object.

Algorithm 3 MULTIPLY, as introduced in [**tractable_ops**]

Input: PSDDs n_1, n_2, vtree node v (n_1, n_2 are normalized w.r.t. v)
Output: PSDD n, constant κ

if (n_1, n_2) in cache **then** ▷ Check if previously computed (PC is a DAG)
  $n, \kappa \leftarrow \text{cache}(n_1, n_2)$
  return n, κ
end if
if v is a leaf **then** ▷ n_1, n_2 are literals, Bernoullis or $\bot$
  if $n_1 = \bot$ or $n_2 = \bot$ or (n_1, n_2 are literals and $n_1 = \neg n_2$) **then**
    return $\bot, 0$ ▷ n_1 and n_2 are incompatible
  else if n_1, n_2 are literals and $n_1 = n_2$ **then**
    return $n_1, 1$
  else if n_1 is positive literal and n_2 is Bernoulli **then**
    return $n_1, \theta(n_2)$ ▷ $\theta(n)$ is Bernoulli parameter of n
  else if n_1 is negative literal and n_2 is Bernoulli **then**
    return $n_1, (1 - \theta(n_2))$
  else if n_1 is Bernoulli and n_2 is positive literal **then**
    return $n_2, \theta(n_1)$
  else if n_1 is Bernoulli and n_2 is negative literal **then**
    return $n_2, (1 - \theta(n_1))$
  else if n_1 is Bernoulli and n_2 is Bernoulli **then**
    $\theta_1, \theta_2 \leftarrow \theta(n_1), \theta(n_2)$
    return $n_1, \theta_1 \cdot \theta_2 \cdot (\theta_1 \cdot \theta_2 + (1 - \theta_1) \cdot (1 - \theta_2))^{-1}$ ▷ Normalization
  end if
else
  $\gamma, \kappa \leftarrow \{\}, 0$
  for (p, s, w) in children(n_1) **do**
    for (p', s', w') in children(n_2) **do**
      $m_1, k_1 \leftarrow \text{MULTIPLY}(p, p', \text{left}(v))$ ▷ Multiply primes recursively
      if $k_1 \neq 0$ **then** ▷ Check if primes were compatible
        $m_2, k_2 \leftarrow \text{MULTIPLY}(s, s', \text{right}(v))$ ▷ Multiply subs recursively
        $\eta \leftarrow k_1 \cdot k_2 \cdot w \cdot w'$ ▷ Compute new weights of (m_1, m_2)
        $\kappa \leftarrow \kappa + \eta$ ▷ Accumulate weights for normalization
        add (m_1, m_2, η) to γ
      end if
    end for
  end for
  $\gamma \leftarrow \{(m_1, m_2, \eta \cdot \kappa^{-1}) \mid (m_1, m_2, \eta) \in \gamma\}$ ▷ Normalize weights
  $n \leftarrow$ unique PSDD node with children γ ▷ Cache lookup for unique nodes
end if
add n, κ to cache
return (n, κ)

Algorithm 4 COMPILE_PMF

Input: Vtree v, PMF p, is_rightmost

if v is a leaf **then**
  if is_rightmost **then**
    return {node: true node, weight: 1, bitstring: null}
  else
    $x \leftarrow v$.literal
    return [{node: literal $\neg x$, weight: $p(\neg x)$, bitstring: [0]},
      {node: literal x, weight: $p(x)$, bitstring: [1]}]
  end if
end if
$p_l \leftarrow p$(vars in left(v)) ▷ Marginal distribution over left vars
nodes_left $\leftarrow$ COMPILE_PMF(left(v), p_l, False)
nodes $\leftarrow$ []
for all n_l in nodes_left where n_l.weight > 0 **do**
  $p_r \leftarrow p$(vars in v | vars in left(v) $= n_l$.bitstring)
  nodes_right $\leftarrow$ COMPILE_PMF(right(v), p_r, is_rightmost)
  for all n_r in nodes_right where n_r.weight > 0 **do**
    $n \leftarrow$ sum node with single product child with prime n_l and sub n_r
    if n_r.bitstring $\neq$ null **then**
      $b \leftarrow$ concat n_l.bitstring and n_r.bitstring
    else
      $b \leftarrow$ null
    end if
    add {node: n, weight: n_l.weight $\cdot$ n_r.weight, bitstring: b} to nodes
  end for
end for
if v is root or nodes_right is singleton set with true node **then**
  $n \leftarrow$ merge nodes into single sum node
  return {node: n, weight: 1, bitstring: null}
end if
return *nodes*

Appendix C

Proofs

C.1 Proof of Theorem ??

Proof. Assume $p(\mathbf{x})$ to be a smooth and decomposable PC and we want to marginalize out a subset $\mathbf{x}_s$. Without loss of generality, we can assume that every path from the root to a leaf is composed of *alternating* sum and product nodes, starting with a sum node[1] with weights $w_1, \ldots, w_n$. Since on the top level, $p(\mathbf{x})$ is a proper mixture of $p_1(\mathbf{x}), \ldots, p_n(\mathbf{x})$, we can write

$$\sum_{\mathbf{x}_s} p(\mathbf{x}) = \sum_{\mathbf{x}_s} \sum_{i=1}^{n} w_i p_i(\mathbf{x}) \tag{C.1}$$

$$= \sum_{i=1}^{n} w_i \sum_{\mathbf{x}_s} p_i(\mathbf{x}) \tag{C.2}$$

effectively *pushing* the sum $\sum_{\mathbf{x}_s}$ through the sum node. Similarly, the sum $\sum_{\mathbf{x}_s}$ decomposes over product nodes: Assume, w.l.o.g., that each component $p_i(\mathbf{x})$ is a product of distributions $p'_1(\mathbf{x}_1), \ldots, p'_k(\mathbf{x}_k)$ with disjoint scopes $\mathbf{x}_1, \ldots, \mathbf{x}_k$. Then,

$$\sum_{i=1}^{n} w_i \sum_{\mathbf{x}_s} p_i(\mathbf{x}) = \sum_{i=1}^{n} w_i \sum_{\mathbf{x}_s} \prod_{j=1}^{k} p'_j(\mathbf{x}_j) \tag{C.3}$$

$$= \sum_{i=1}^{n} w_i \prod_{j=1}^{k} \sum_{\mathbf{x}_s \cap \mathbf{x}_j} p'_j(\mathbf{x}_j) \tag{C.4}$$

i.e., sums decompose into *smaller* ones. We can repeat these two steps until we reach a leaf. Since leaves are properly normalized probability distributions, a sum over them trivially yields 1. Consequently, to compute $\sum_{\mathbf{x}_s} p(\mathbf{x})$, we replace all leaf nodes with scope $\in \mathbf{x}_s$ with a constant 1 and perform a *single forward pass* (i.e., we follow the

[1] If we had a PC with multiple *consecutive* sum or product nodes along a path, we could easily merge them. If the root node is a product node, we can always add a unary sum node with weight 1.

computational graph from leaves to root), collecting the exact marginal at the root node. □

C.2 Proof of Theorem ??

Proof. Assume a PC $p(\mathbf{x})$ to be smooth, decomposable, and deterministic. Due to determinism, every sum node has at most one child that evaluates to a non-zero value and contributes to the sum. Consider a sum node with children $p_1(\mathbf{x}), \ldots, p_n(\mathbf{x})$ and weights $w_1, \ldots, w_n$. Then, we can write

$$\sum_i w_i p_i(\mathbf{x}) = \max_i w_i p_i(\mathbf{x}) \tag{C.5}$$

If we want to maximize the mixture, we can use this fact to *push* the max operation through the sum node:

$$\max_{\mathbf{x}} \sum_i w_i p_i(\mathbf{x}) = \max_{\mathbf{x}} \max_i w_i p_i(\mathbf{x}) \tag{C.6}$$

$$= \max_i \max_{\mathbf{x}} w_i p_i(\mathbf{x}) \tag{C.7}$$

When encountering a product node with children $p'_1(\mathbf{x}_1), \ldots, p'_k(\mathbf{x}_k)$ with disjoint scopes $\mathbf{x}_1, \ldots, \mathbf{x}_k$, we see that

$$\max_{\mathbf{x}} \prod_j p'_j(\mathbf{x}_j) = \prod_j \max_{\mathbf{x}_j} p'_j(\mathbf{x}_j) \tag{C.8}$$

As a consequence, we can recursively push the max operation from the root node to the leaves. When the leaf is a simple parametric distribution, computing its maximum (or the maximizing argument) is usually trivial (e.g., Bernoulli, Gaussian). To summarize, to compute $\max_{\mathbf{x}} p(\mathbf{x})$, we replace all leaf nodes with the maximum value it can assume and compute a *single forward pass* while replacing all sum nodes with max nodes. To find $\operatorname{argmax}_{\mathbf{x}} p(\mathbf{x})$, we can use the same algorithm, but also track the arguments that maximize a node. □

C.3 Proof of Theorem ??

Proof. Let $p(\mathbf{v}) = \prod_{j=1}^{K} p^{(j)}(v_j)$ and let $\mathcal{V}_j$ be the set of values v such that the π-sparse approximation $\tilde{p}^{(j)}_{\pi}(v)$ assigns positive mass to v (i.e., its support)[2]. More precisely,

[2]We omit conditioning on $\boldsymbol{\ell}$ for succinctness.

the relation between the j^{th} input distribution $p^{(j)}$ and its approximation is given by

$$\tilde{p}_{\pi}^{(j)}(v) = \begin{cases} p^{(j)}(v) \cdot Z_j^{-1} & \text{if } v \in \mathcal{V}_j \\ 0 \quad \text{else} \end{cases} \tag{C.9}$$

with $Z_j = \sum_{v \in \mathcal{V}_j} p^{(j)}(v)$. As defined in Theorem **??**, let $\mathcal{J} \subseteq \{1, \ldots, K\}$ denote the set of k indices of π-sparse approximations and recall that

$$\tilde{p}(\mathbf{v}) = \prod_{j \in \mathcal{J}} \tilde{p}_{\pi}^{(j)}(v_j) \cdot \prod_{j \notin \mathcal{J}} p^{(j)}(v_j) \tag{C.10}$$

Note that for arbitrary PMFs $q(\mathbf{x}), q'(\mathbf{x}), r(\mathbf{y})$ over disjoint sets of variables $\mathbf{x}, \mathbf{y}$, we have

$$\begin{aligned} D_{KL}\left(q(\mathbf{x})r(\mathbf{y}) \;||\; q'(\mathbf{x})r(\mathbf{y})\right) &= \sum_{\mathbf{x}} \sum_{\mathbf{y}} q(\mathbf{x})r(\mathbf{y}) \cdot \log\left(\frac{q(\mathbf{x})r(\mathbf{y})}{q'(\mathbf{x})r(\mathbf{y})}\right) && \text{(C.11)} \\ &= \sum_{\mathbf{x}} q(\mathbf{x}) \log\left(\frac{q(\mathbf{x})}{q'(\mathbf{x})}\right) \cdot \sum_{\mathbf{y}} r(\mathbf{y}) && \text{(C.12)} \\ &= D_{KL}\left(q(\mathbf{x}) \;||\; q'(\mathbf{x})\right) && \text{(C.13)} \end{aligned}$$

i.e., identical factors cancel in the computation of the KL-divergence. Thus,

$$D_{KL}\left(\tilde{p}(\mathbf{v}) \;||\; p(\mathbf{v})\right) = D_{KL}\left(\prod_{j \in \mathcal{J}} \tilde{p}_{\pi}^{(j)}(v_j) \;||\; \prod_{j \in \mathcal{J}} p^{(j)}(v_j)\right) \tag{C.14}$$

Let $Z = \prod_{j \in \mathcal{J}} Z_j$ and $\mathcal{V} = \mathcal{V}_{j_1} \times \cdots \times \mathcal{V}_{j_k}$ where $\mathcal{J} = \{j_1, \ldots, j_k\}$. We will make use of the fact that $\forall \mathbf{v} = (v_{j_1}, \ldots, v_{j_k})^T \in \mathcal{V}$, we have

$$\begin{aligned} \prod_{j \in \mathcal{J}} \tilde{p}_{\pi}^{(j)}(v_j) &= \prod_{j \in \mathcal{J}} \frac{1}{Z_j} p^{(j)}(v_j) && \text{(C.15)} \\ &= \frac{1}{Z} \prod_{j \in \mathcal{J}} p^{(j)}(v_j) && \text{(C.16)} \end{aligned}$$

Expanding the KL-divergence via its definition, we can heavily simplify:

$$D_{KL}\left(\tilde{p}(\mathbf{v}) \mid\mid p(\mathbf{v})\right) = D_{KL}\left(\prod_{j\in\mathcal{J}} \tilde{p}_{\pi}^{(j)}(v_j) \mid\mid \prod_{j\in\mathcal{J}} p^{(j)}(v_j)\right) \tag{C.17}$$

$$= \sum_{\mathbf{v}\in\mathcal{V}} \prod_{j\in\mathcal{J}} \tilde{p}_{\pi}^{(j)}(v_j) \cdot \log\left(\frac{\prod_{j\in\mathcal{J}} \tilde{p}_{\pi}^{(j)}(v_j)}{\prod_{j\in\mathcal{J}} p^{(j)}(v_j)}\right) \tag{C.18}$$

$$= \sum_{\mathbf{v}\in\mathcal{V}} \frac{1}{Z} \prod_{j\in\mathcal{J}} p^{(j)}(v_j) \cdot \log\left(\frac{\frac{1}{Z}\prod_{j\in\mathcal{J}} p^{(j)}(v_j)}{\prod_{j\in\mathcal{J}} p^{(j)}(v_j)}\right) \tag{C.19}$$

$$= \sum_{\mathbf{v}\in\mathcal{V}} \frac{1}{Z} \prod_{j\in\mathcal{J}} p^{(j)}(v_j) \cdot \log\left(\frac{1}{Z}\right) \tag{C.20}$$

$$= \frac{-\log(Z)}{Z} \prod_{j\in\mathcal{J}} \sum_{v_j\in\mathcal{V}_j} p^{(j)}(v_j) \tag{C.21}$$

$$= \frac{-\log(Z)}{Z} \prod_{j\in\mathcal{J}} Z_j = \frac{-\log(Z)}{Z} Z = -\log(Z) \tag{C.22}$$

By construction of the approximations, we have $Z_j \geq \pi, \forall j \in \mathcal{J}$:

$$-\log(Z) = -\log\left(\prod_{j\in\mathcal{J}} Z_j\right) \tag{C.23}$$

$$\leq -\log\left(\pi^k\right) = -\log(\pi)\cdot k \tag{C.24}$$

□

C.4 Proof of Theorem ??

Proof. Let v be a vtree over variables $x_1, \ldots, x_n$. To construct a PC representation of $p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i)$ that respects v, we will transform v into a PC as follows: Replace every internal node in v with a product node and every leaf in v (denoted x_i) with a Bernoulli distribution with success parameter $\theta_i = p(x_i)$. Since the resulting PC has no sum nodes, the circuit is trivially smooth and deterministic[3]. Since v is a tree and the leaves of v are in a one-to-one correspondence to the variables $x_1, \ldots, x_n$, every internal node in v must have variables in its left subtree $\mathbf{x}_l$ and variables in its right subtree $\mathbf{x}_r$ such that $\mathbf{x}_l \cap \mathbf{x}_r = \emptyset$. Therefore, the resulting circuit is decomposable. Further, the PC is, trivially, structured decomposable since each product node has a unique scope. As v is a full binary tree, it has $2n-1 \in O(n)$ nodes and $2n-2 \in O(n)$ edges. As every node and edge is visited exactly once in the construction process of

[3]Clearly, these properties hold even if we introduce unary sum nodes.

the PC, the time and space complexity of this algorithm is $\in O(n)$ (assuming that creating nodes and edges in a PC is a constant time operation). □

C.5 Proof of Theorem ??

Proof. As p_2 has a single node for every vtree node, the inner loop in Algorithm **??** (that loops over children(n_2)) collapses to a single iteration. As it can be shown that Algorithm **??** visits every node in p_1 exactly once [**tractable_ops**], the resulting time complexity is $O(s_1)$. If s_2 denotes the size of p_2, we also note that $s_1 \geq s_2$ as p_2 is the smallest PSDD that respects the corresponding vtree. □