Exact Soft Analytical Side-Channel Attacks using Tractable Circuits

Thomas Wedenig [1] **Rishub Nagpal** [1] **Gaëtan Cassiers** [2] **Stefan Mangard** [1] **Robert Peharz** [1]

Abstract

Detecting weaknesses in cryptographic algorithms is of utmost importance for designing secure information systems. The state-of-the-art *soft analytical side-channel attack* (SASCA) uses physical leakage information to make probabilistic predictions about intermediate computations and combines these "guesses" with the known algorithmic logic to compute the posterior distribution over the key. Since exact inference is assumed to be prohibitive, the attack is commonly performed via loopy belief propagation, which, however, lacks guarantees in terms of convergence and inference quality. In this paper, we develop a fast and exact inference method for SASCA, denoted as ExSASCA, by leveraging knowledge compilation and tractable probabilistic circuits. When attacking the *Advanced Encryption Standard* (AES), the most widely used encryption algorithm to date, ExSASCA outperforms SASCA by more than 31% top-1 success rate absolute. By leveraging sparse belief messages, this performance is achieved with little more computational cost than SASCA, and about 3 orders of magnitude less than exact inference via exhaustive enumeration. Even with dense belief messages, ExSASCA still uses 6 times less computations than exhaustive inference.

1. Introduction

Cryptographic algorithms are the backbone of our modern information society, securing communication and data from daily business to governmental secrets. Cryptographic attacks take an adversarial perspective and play a central role in defining, analysing and designing cryptographic algorithms. Specifically, *side-channel attacks* attempt to learn about secret material used in a cryptographic computation by observing implementation artifacts, commonly called *leakages*, such as timing information (Kocher, 1996), electromagnetic emissions (Gandolfi et al., 2001), and power consumption (Kocher et al., 1999). Most contemporary attacks use these leakages to reason about the internal state of a cryptographic algorithm in a probabilistic manner. Specifically, in *template attacks* (Chari et al., 2002) the attacker has access to a clone of the device under attack and can run the algorithm repeatedly with randomly generated key and plaintexts as inputs. Simultaneously, the attacker also records leakage information during all runs, e.g. the *power trace* of the device. Using this data, the attacker can easily build a set of probabilistic models, predicting distributions of intermediate values conditional on the leakage.

The attacker's goal is now to combine these value distributions ("soft guesses") and the (hard logical) knowledge about the algorithm to infer the posterior distribution of the secret key. A naïve exhaustive computation of this posterior is either infeasible or might take hours of compute time for a single leakage, which is prohibitive for practical attacks. As a remedy, the state-of-the-art *soft analytical side-channel attack* (SASCA) (Veyrat-Charvillon et al., 2014) uses loopy belief propagation (BP) on a factor graph representation of the cryptographic algorithm, to produce an approximate key posterior. However, loopy BP has limited theoretical underpinnings as neither convergence is granted, nor does it guarantee accurate posterior estimates in case of convergence (Knoll, 2022). Consequently, while SASCA *can* detect weaknesses in cryptosystems, an unsuccessful SASCA does *not* imply any certificate that the leakage cannot be exploited further. On the contrary, it is straightforward to construct examples where inference is in fact easy but loopy BP fails catastrophically.

In this paper, we develop a fast and exact inference algorithm for SASCA, denoted as *ExSASCA*, for the *Advanced Encryption Standard (AES)*, the most prominent and widely used encryption algorithm to date. To this end, we build on recent results from knowledge compilation (Darwiche & Marquis, 2002; Darwiche, 2011) and tractable probabilistic circuits (Kisa et al., 2014; Vergari et al., 2020). Similarly as in SASCA, we start from a factor graph representation of AES, whose core part, MIXCOLUMNS, is a highly loopy subgraph involving 168 binary variables (bits). While SASCA performs loopy BP, we instead compile MIXCOLUMNS into

[1]Graz University of Technology, Austria [2]UCLouvain, Belgium. Correspondence to: Thomas Wedenig <thomas.wedenig@tugraz.at>.

Accepted by the Structured Probabilistic Inference & Generative Modeling workshop of ICML 2024, Vienna, Austria. Copyright 2024 by the author(s).

a compact *(probabilistic) sentential decision diagram* (SDD, PSDD) (Darwiche, 2011; Kisa et al., 2014), a circuit representation allowing a wide range of logic and probabilistic inference routines (Vergari et al., 2020). On a conceptual level, this amounts to replacing MIXCOLUMNS with a *single* factor and reducing the factor graph to a *tree*, on which we can perform exact message passing (Koller & Friedman, 2009), albeit involving a factor with 2^{168} entries. The compiled PSDD is compact, containing only 19,000 sums and products, and allows inference in *polynomial time of the circuit size*.

In particular, for the key posterior we require mainly *factor multiplication* (worst-case quadratic in the circuit size) and *factor summation* (linear in the circuit size), which are readily provided by tractable circuits and have previously been employed in structured Bayesian networks (Shen et al., 2019) and neuro-semantic approaches (Ahmed et al., 2022). When attacking standard AES, we leverage the fact that the value distributions are typically concentrated on byte values with the same hamming weight, allowing us to work with sparse messages. With this simplification, an out-of-the box implementation of ExSASCA yields an improvement of *31% top-1 success rate absolute* in comparison to conventional SASCA. Here, the required computation for ExSASCA is on the same level as SASCA, which is *three orders of magnitude* less than in the only other known exact inference algorithm based on exhaustive enumeration.

Our out-of-the-box implementation ExSASCA is not amenable to *protected* AES which yields dense value distributions. For this case we develop a novel dynamic compilation strategy for SDDs, reducing inference to a weighted model counting problem. The resulting inference machine still requires 6 times less computations than exhaustive inference and can be applied to both protected and unprotected AES. ExSASCA consistently outperforms SASCA on all protection levels. Overall, our main contributions are:

- We propose ExSASCA, an exact SASCA implementation, which substantially improves the success rate of attacks on AES, while using far less computational resources than inference by exhaustive enumeration.
- We develop a novel dynamic compilation framework for circuit-based inference in large probabilistic systems, which can be generally applied, as for example in structured Bayesian networks (Shen et al., 2019).
- With our method we open a new avenue to study vulnerability in cryptosystems; in the long run our techniques might lead to stronger theoretical guarantees in cryptographic algorithms, for example using results from circuit complexity (de Colnet & Mengel, 2021) for proving the non-existence of a tractable circuit representation of particular cryptographic algorithms.

2. Background

2.1. Advanced Encryption Standard

In this work, we attack the *Advanced Encryption Standard* (AES) (Daemen & Rijmen, 1998), the most popular symmetric-key block cipher used to date. AES takes an 128-bit *plaintext* $\mathbf{p}$ and a secret b-bit *key* $\mathbf{k}$ as input, and produces a 128-bit *ciphertext* $\mathbf{c}$ as output. In our experiments, we attack AES-128, i.e., $b = 128$. We refer to bytes of the key and plaintext as *input variables* and the bytes of the ciphertext as output variables; besides this, the algorithm also computes several byte-valued *intermediate variables* in the course of computing the ciphertext. Further, let $\mathbb{F}_c$ denote the finite field (Galois field) of characteristic c. AES encryption is performed in 10 iterations (so-called rounds), where each round essentially consists of the functions SUBBYTES, SHIFTROWS, MIXCOLUMNS, and ADDROUNDKEY. Figure 1 illustrates such a round.

2.2. Side-Channel Attacks

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 (Spreitzer et al., 2018). 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 data dependent (Standaert, 2010). 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 (Standaert, 2010). While adversaries can exploit various side channels such as timing of operations (Kocher, 1996) and electromagnetic emanation (Gandolfi et al., 2001), this work focuses on power traces.

2.2.1. TEMPLATE ATTACKS

In a template attack the adversarial has access to a *clone* of the target device (e.g., a smart card). First, in the so-called *profiling phase*, the attacker queries the clone device with a large number of random inputs, i.e. key and plaintext in case of AES, and records side-channel leakage of their choice. Let v be an arbitrary byte-valued variable involved in AES (input, intermediate variable, or output). The profiling phase yields a dataset $\{(\boldsymbol{\ell}^{(i)}, v^{(i)})\}_{i=1}^{n}$, where n is the number of profiling runs and $\boldsymbol{\ell} \in \mathbb{R}^d$ denotes the leakage. This data is then used to construct a likelihood model $p(\boldsymbol{\ell} \mid v)$, a so-called *template*. A common choice are multivariate Gaussians with mean $\boldsymbol{\mu}_v$ and covariance $\boldsymbol{\Sigma}_v$, i.e., $p(\boldsymbol{\ell} \mid v) = \mathcal{N}(\boldsymbol{\ell}; \boldsymbol{\mu}_v, \boldsymbol{\Sigma}_v)$ for each value $v \in \{0, \ldots, 255\}$. Hence, n is typically at least a few thousand, to make sure that for each value there are sufficiently many leakage samples.

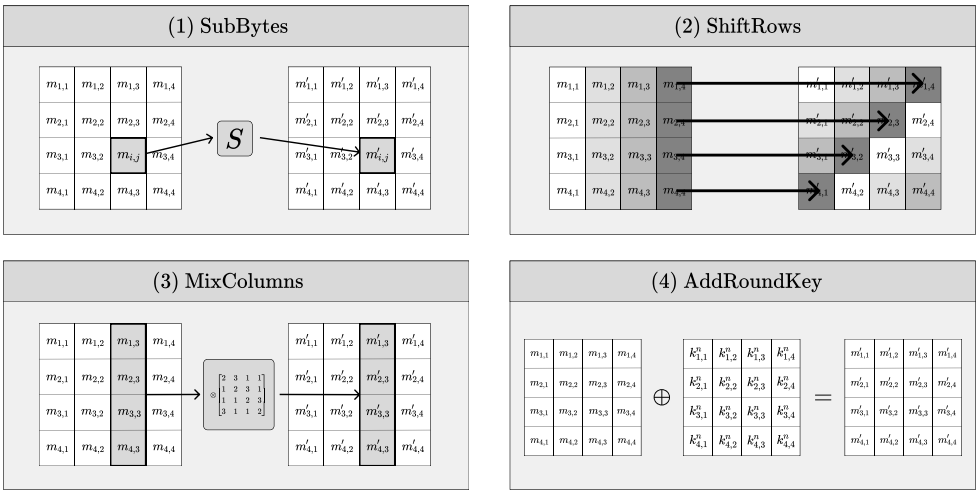

Figure 1. A round of AES takes a 4×4-byte matrix $\mathbf{M}$ and computes a series of functions: (1) SUBBYTES applies a non-linear bijection S to each byte individually, (2) SHIFTROWS shifts the rows of the input matrix to the left, (3) MIXCOLUMNS computes a linear function which takes each input column and performs a matrix-vector product in the Galois Field $\mathbb{F}_{256}$ and (4) ADDROUNDKEY, a byte-wise XOR operation with the "round key" (which is just the key in the first round).

Since the leakages $\boldsymbol{\ell}$ are high-dimensional (e.g., $d \approx 10^5$ for power traces), one usually first applies some dimensionality reduction. We adopt the approach from (Bronchain et al., 2021), using a combination of interest point detection in $\boldsymbol{\ell}$ and linear discriminant analysis. We also experimented extensively with various neural network architectures to construct more expressive templates, which, however, consistently performed worse than (Bronchain et al., 2021). Finally, in the attack phase, only the leakage $\boldsymbol{\ell}$ is observed. Using Bayes' law and a uniform prior $p(v)$, we get a *local distribution* (*belief*) for v, conditional on the leakage:

$$p(v \mid \boldsymbol{\ell}) = \frac{p(\boldsymbol{\ell} \mid v)}{\sum_{v'=0}^{255} p(\boldsymbol{\ell} \mid v')} \tag{1}$$

The procedure above, explained for a generic value v, is done for several byte variables $v_1, \ldots, v_k$ computed in AES. Usually, one selects variables which are "close to the key" in the computational path, as these will typically be more correlated with the secret key than variables "further away."

2.2.2. SOFT ANALYTICAL SIDE-CHANNEL ATTACKS

After obtaining distributions $p(v_1 \mid \boldsymbol{\ell}), \ldots, p(v_k \mid \boldsymbol{\ell})$ for all variables of interest, the attacker aims to aggregate them into a posterior distributions over the *key bytes*. Intuitively, the beliefs about intermediate variables, which depend operationally and logically on the secret key, should be propagated backwards and combined into beliefs about the key. This is naturally expressed using a *factor graph* (Kschischang et al., 2001; Veyrat-Charvillon et al., 2014), where variable nodes (circles) correspond to the byte-valued variables and factors (black squares) represent indicator functions that model the logical relationship between variables.

Figure 2 (left) shows the factor graph for part of the first round of AES, consisting of ADDROUNDKEY, SUBBYTES and one column multiplication of MIXCOLUMNS. We do not model the SHIFTROWS operation since it merely corresponds to a fixed re-labeling of the input bytes and does not affect probabilistic inference. Note that the nodes representing the plaintext are shaded, meaning that they are observed, which is a common assumption in this type of attacks (Veyrat-Charvillon et al., 2014). While the hardware implementation of the SBOX also leaks information in practice, we simply abstract it as a 256×256-dimensional binary factor (i.e., a lookup table).

Every variable (blank node) in the factor graph is equipped with a local distribution derived via a template attack, as described in Section 2.2.1. Note that the figure shows only one of the four parallel branches of the first AES round; the other three branches are of identical structure and can be treated independently. In this work we attack, as common in literature (Veyrat-Charvillon et al., 2014), only the first round of AES; attacking later rounds is doable, but is of diminishing return.

The desired posterior over the key bytes is now simply given as a *factor product* of all involved factors, followed by a summation over all unobserved variables, except for the key bytes (Kschischang et al., 2001). Evidently, a naïve implementation is infeasible, since the total factor product is of size 2^{232} (involving 29 bytes). However, exploiting the fact that the factor graph encodes an algorithm and that all variables follow deterministically from the key (and plaintext), one actually just needs to enumerate the values of the 4 key bytes and evaluate the factor graph for each of

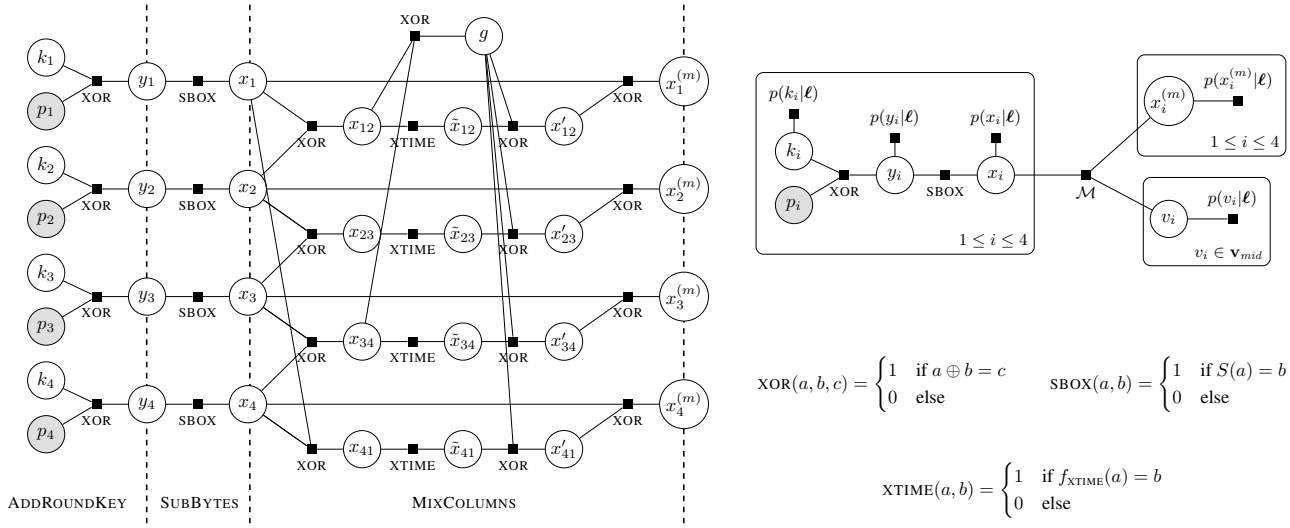

Figure 2. (left) Factor graph over the first four key bytes and plaintext bytes and their operations in AES. Black squares denote logical factors that represent AES operations. For example, an XOR-factor is 1 if and only if the variable on its right is the *bit-wise exclusive or* of the two variables on its left and 0 otherwise. Similarly, SBOX encodes a bijection S between byte-values and XTIME encodes a multiplication with 2 in $\mathbb{F}_{256}$ (abstracted as f_{XTIME}). Every unobserved (blank) variable node has an additional factor $p(v \mid \boldsymbol{\ell})$ (omitted for sake of visual clarity). Shaded variable nodes are observed. (right) The same factor graph, but where the loopy MixColumns part has been summarized in a single high-dimensional factor $\mathcal{M}$ (represented by a PSDD). We use plate notation to illustrate structurally identical parts. The set $\mathbf{v}_{mid}$ contains all intermediate variables in the MixColumns function.

these 2^{32} combinations. The desired key posterior is then given by re-normalizing the computed factors.

As inference by exhaustive key enumeration is computationally expensive, the *soft analytical side-channel attack* (SASCA) (Veyrat-Charvillon et al., 2014) applies loopy belief propagation (BP) (Kschischang et al., 2001) to approximate the desired conditional distribution over key bytes $p(k_i \mid \boldsymbol{\ell})$. Loopy BP requires about three order of magnitudes less compute than exhaustive inference and is thus considered a practically relevant attack. However, while SASCA *might* detect weaknesses in a cryptoalgorithm, an unsuccessful application of SASCA is no certificate towards security of the attacked system. In particular, loopy BP is notoriously unpredictable, as neither convergence is granted, nor does convergence imply that the posterior has been successfully approximated (Knoll, 2022). In fact, it is straightforward to design easy inference problems where loopy BP fails catastrophically.

3. Exact Soft Analytical Side-Channel Attacks

In this paper, we develop the first exact attack on AES-128 which substantially improves over SASCA and uses far less computational resources than exhaustive enumeration. Our high-level strategy is to reduce the problem to a tree-shaped factor graph, for which BP becomes an exact inference algorithm (Kschischang et al., 2001; Koller & Friedman, 2009). Note that the central problem in Figure 2 (left) is the sub-graph corresponding to MixColumns, which involves 21 byte-valued variables and has many loops. We can now summarize this part into a *single* logical factor $\mathcal{M}(\mathbf{v})$ as shown in Figure 2 (right), where $\mathbf{v}$ contains all 21 variables. Thus, $\mathcal{M}(\mathbf{v})$ is a binary function evaluating to 1 if and only if $\mathbf{v}$ corresponds to variable assignment consistent with the entire MixColumns computation.

Running BP in the resulting tree-shaped factor graph yields the exact marginals, by propagating messages from $\mathcal{M}$ to the input bytes x_i, computed as (Kschischang et al., 2001),

$$\mu_{\mathcal{M}\to x_i}(x_i) = \sum_{\mathbf{v}\setminus x_i} \mathcal{M}(\mathbf{v}) \cdot \prod_{v_j \in \mathbf{v}\setminus x_i} p(v_j|\boldsymbol{\ell}) \qquad (2)$$

which requires only two operations, namely *factor multiplication* and *factor summation*. However, naïvely computing these messages requires enumerating all 2^{168} possible assignments for $\mathbf{v}$, which is clearly intractable.

In this paper, we therefore build on recent results from knowledge compilation (Darwiche & Marquis, 2002; Darwiche, 2011) and tractable probabilistic circuits (Kisa et al., 2014; Vergari et al., 2020). The aim of *knowledge compilation* (Darwiche & Marquis, 2002) is to convert a logical formula, e.g. given in conjunctive normal form (CNF), into a different target representation that can tractably (i.e., in polynomial time) answer certain queries (e.g., the SAT problem). In our context, the relevant formula is the boolean factor $\mathcal{M}$ for MixColumns.

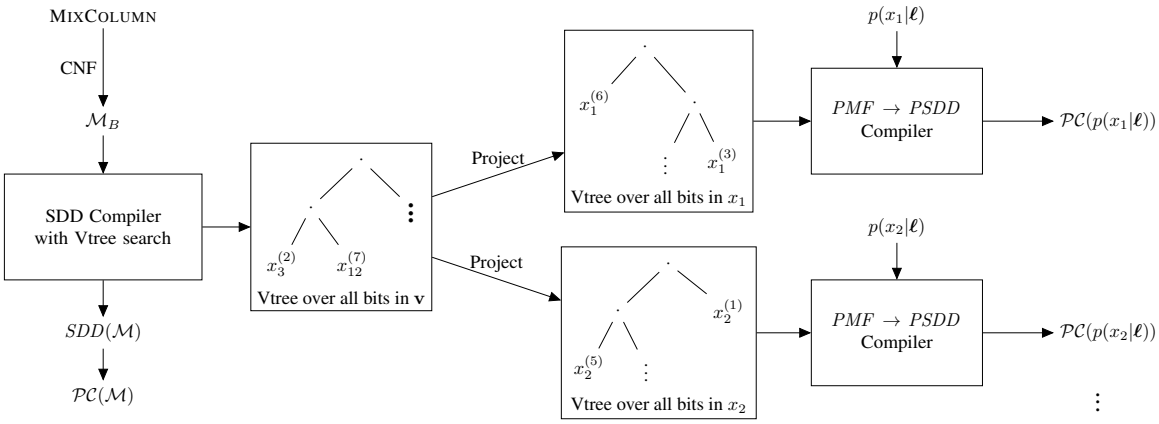

Figure 3. Given the algorithmic description of MIXCOLUMN ($\mathcal{M}$) and the local beliefs $p(v \mid \boldsymbol{\ell}), v \in \mathbf{v}$ as inputs to our compilation pipeline, we yield a PSDD representations of all input distributions, denoted $\mathcal{PC}(p(v \mid \boldsymbol{\ell}))$. Moreover, all PSDDs are pairwise *compatible* for downstream circuit multiplication tasks.

Specifically, we compile $\mathcal{M}$ into a *sentential decision diagram* (SDDs) (Darwiche, 2011) which is subsequently converted into its probabilistic extension, *probabilistic SDDs* (PSDDs) (Kisa et al., 2014). PSDDs are a special case of probabilistic circuit, a structured representation of high-dimensional probability distributions allowing a wide range of tractable probabilistic inference routines (Vergari et al., 2020). Our constructed PSDD represents the uniform distribution over all $\mathbf{v}$ which are consistent with MIXCOLUMNS, while inconsistent $\mathbf{v}$ get probability 0.

PSDDs allow both factor multiplication and factor summation in polynomial time (Vergari et al., 2020; Choi et al., 2020), as required in (2). Furthermore, they also allow to compute a *most probable evidence* (MPE), i.e. a probability maximizing assignment. These tractable routines are the key routines we require for attacking AES. Before proceeding with implementation details, we review the required background on tractable circuits, in particular SDDs and PSDDs. At their core, SDDs are hierarchical and structured decompositions of Boolean functions, defined by so-called *compressed partitions*.

Definition 3.1 (Compressed Partition (Darwiche, 2011))**.** Let $f(\mathbf{x}, \mathbf{y})$ be a Boolean function over disjoint sets of binary variables $\mathbf{x}$ and $\mathbf{y}$. Any such f can be written as $f = (p_1(\mathbf{x}) \wedge s_1(\mathbf{y})) \vee \cdots \vee (p_k(\mathbf{x}) \wedge s_k(\mathbf{y}))$, where p_i (*primes*) and s_i (*subs*) are Boolean functions over $\mathbf{x}$ and $\mathbf{y}$, respectively. Moreover, this decomposition can be constructed such 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 $i \neq j$. Then we call $\{(p_1, s_1), \ldots, (p_k, s_k)\}$ a *compressed* $\mathbf{x}$*-partition* of f.

A structure-defining notion of SDDs are *vtrees*.

Definition 3.2 (Vtree (Pipatsrisawat & Darwiche, 2008))**.** A *vtree*, or *variable tree*, over a set of binary variables $\mathbf{z}$ is a full, rooted binary tree whose leaves are in a one-to-one correspondence to the variables in $\mathbf{z}$.

Definition 3.3 (Sentential Decision Diagram (Darwiche, 2011))**.** Let v be a vtree over binary variables $\mathbf{z}$. A *sentential decision diagram* (SDD) is a logical circuit where each internal node is either an *or*-node ($\vee$) or an *and*-node ($\wedge$), while leaf nodes correspond to variables, their negation, $\top$, or $\bot$. W.l.o.g., we demand that every *or*-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 *or*-node n in the SDD is *normalized* w.r.t. a vtree node in v.

Hence, an SDD is a representation of a Boolean function; due its structural constraints it allows a wide range of logical operations (Darwiche, 2011). As mentioned above, we can also think of an SDD as an unnormalized uniform distribution over all satisfying assignments $\mathbf{z}$. It is straightforward to turn an SDD into a proper probability distribution, a so-called PSDD (Kisa et al., 2014).

Definition 3.4 (Probabilistic SDD)**.** Given an SDD, the corresponding PSDD is obtained by replacing all *and*-nodes with *product* nodes and all *or*-nodes with *sum* nodes. A *product* node computes the product of its inputs while a *sum* node computes a convex combination of its inputs.

We denote the PSDD representation of a Boolean function f as $\mathcal{PC}(f)$. A PSDD is a type of *Probabilistic Circuit* (PC) (Vergari et al., 2020; Choi et al., 2020) with particular properties, allowing (i) *arbitrary factor summation*

(marginalization) in time linear in the circuit size; (ii) *circuit multiplication* of two circuits respecting the same vtree, yielding a PSDD whose size is in the worst-case the product of the sizes of the involved circuits; (iii) computing a *most-probable-explanation*, i.e. a probability maximizing variable assignment in time linear in the circuit size. Hence, PSDDs provide, among others, the operations required for exact message passing (2). The crucial first step is to compile $\mathcal{M}$.

3.1. Compiling MixColumn into an SDD

In order to compile MIXCOLUMN (Algorithm 2) to an SDD, we represent it as a Boolean function over 168 binary variables by replacing all variable assignments with equivalence constraints. Represented in conjunctive normal form (CNF), $\mathcal{M}$ consists of 648 clauses with an average of 3.09 literals within a clause. To compile the CNF into an SDD representation $SDD(\mathcal{M})$, we leverage the bottom-up SDD compiler introduced in (Choi & Darwiche, 2013). Initializing the vtree to be left-linear and using dynamic vtree minimization (Choi & Darwiche, 2013), we find that the resulting SDD contains $19k$ sums and products and takes ≈ 30 seconds to compile on a modern laptop CPU.

Further, to tractably compute the BP message in (2), we need to represent the product $\mathcal{M}(\mathbf{v}) \cdot \prod_{v_j \in \mathbf{v} \setminus x_i} p(v_j|\boldsymbol{\ell})$ as a circuit. We present two methods for this task, namely (i) an "out-of-the-box" implementation, compiling the mass functions $p(v_j|\boldsymbol{\ell})$ upon observing $\boldsymbol{\ell}$ into PSDDs and computing a sequence of circuit products, described in Section 3.2, and (ii) compiling the product symbolically and reducing the problem to *weighted model counting* (WMC) (Chavira & Darwiche, 2008), described in Section 3.3. The first techniques makes a slight simplifying sparseness assumption on the variable beliefs, while the second method is exact and can work with arbitrary messages.

3.2. PSDD Multiplication Chain

First, we convert the SDD representation of MIXCOLUMN into a PSDD $\mathcal{PC}(\mathcal{M})$. Given the vtree of $\mathcal{PC}(\mathcal{M})$ (ranging over 168 binary variables), we compute a vtree projection (Shen et al., 2016) for each byte v in $\mathbf{v}$, where each projection ranges over the 8 bits that make up v. It is then straightforward to compile $p(v|\boldsymbol{\ell})$ into a PSDD that respects the projected vtree. This technique ensures that all pairs of circuits in $\{\mathcal{PC}(\mathcal{M})\} \cup \{\mathcal{PC}(p(v_j|\boldsymbol{\ell}))\}_{j=1}^{21}$ are compatible and can be multiplied on the circuit level. Thus, the message in Equation 2 can be computed by a tractable marginal query in the circuit product $\mathcal{PC}(\mathcal{M}(\mathbf{v})) \cdot \prod_{v_j \in \mathbf{v} \setminus x_i} \mathcal{PC}(p(v_j|\boldsymbol{\ell}))$. In practice, computing this product is infeasible as the circuit size grows in the worst exponentially in the number of multiplied circuits (i.e. quadratic for two circuits, cubic for three, etc.), and the total number of involved circuits is 22.

However, when attacking vanilla AES, some of local beliefs $p(v|\boldsymbol{\ell}), v \in \mathbf{v}$ will typically have *low entropy*, i.e., the probability mass is concentrated on only a few values, often sharing the same hamming weight. Thus, we can work with *sparse* distributions by setting very small values in the mass functions to 0: Given $p(v|\boldsymbol{\ell})$ and a hyperparameter $0 \leq \varepsilon < 1$, we sort the probabilities in ascending order and find the largest k such that the sum of k smallest probabilities is $\leq \varepsilon$. We then clamp the k smallest probabilities to 0 and re-normalize the mass function. Empirically, we find that even for values of ε as small as 10^{-8}, the runtime of computing this circuit product is comparable to SASCA, while substantially outperforming it.

During compilation, this method can readily leverage sparse probability distributions. However, since we fix the vtree after the initial SDD compilation and we cannot perform dynamic vtree optimization (Choi & Darwiche, 2013) during the PSDD multiplication chain. Since dynamic compilation is key to achieve small circuits, this out-of-the-box method is computationally infeasible when dropping the low entropy assumption. This is in particular the case for protection techniques which add noise to the leakage signals.

3.3. SDD with Auxiliary Byte Indicators

Our second strategy is to perform *symbolic* circuit multiplication, which can take use of dynamic vtree optimization (Choi & Darwiche, 2013). Specifically, we take the SDD representation of $SDD(\mathcal{M})$ modelling *bit interactions* and, akin to previous work on SDD compilation (Choi et al., 2013), successively add auxiliary variables that represent the values of *bytes*. For example, let v be a byte that consists of the bits $v^{(1)}, \ldots, v^{(8)}$. For each byte variable v in MIXCOLUMN, we create 256 auxiliary boolean variables $b_{v=0}, \ldots, b_{v=255}$ that indicate the state of v. After adding these *byte indicator variables* to the vtree, we conjoin the following constraint to the current SDD:

$$\left(b_{v=0} \Leftrightarrow \left(\neg v^{(1)} \wedge \cdots \wedge \neg v^{(8)}\right)\right) \wedge \ldots$$
$$\left(b_{v=255} \Leftrightarrow \left(v^{(1)} \wedge \cdots \wedge v^{(8)}\right)\right)$$

To efficiently encode this into an SDD, we follow the method proposed in (Choi et al., 2013) and compute a base SDD α that represents $\neg b_{v=0} \wedge \cdots \wedge \neg b_{v=255}$ that we can re-use throughout the compilation.

When compared to the PSDD multiplication chain, the advantage of this approach is that we can utilize dynamic vtree optimization (Choi & Darwiche, 2013) *throughout the entire compilation procedure*. In the SDD with byte indicators, computing the BP messages defined in Equation 2 is effectively a series of weighted model counting (WMC) problems: Let $w(l)$ denote the weight of a literal l. To compute $\mu_{\mathcal{M} \to x_i}(x_i = x)$, we set $w(b_{x_i=x}) = 1$,

$w(\neg b_{x_i=x}) = 0$, and the weight of all other byte indicators to $w(b_{v=x}) = p(v = x|\boldsymbol{\ell})$ and $w(\neg b_{v=x}) = 1$ $\forall x \in \{0, \ldots, 255\}, \forall v \in \mathbf{v} \setminus x_i$, while we set $w(l) = 1$ for all non-auxiliary literals l. Naïvely, this entails that computing $\{\mu_{\mathcal{M}\to x_i}\}_{i=1}^{4}$ requires $4 \cdot 256$ WMC executions. However, it is known that all marginals (i.e., messages) can be computed using a *single WMC computation and its derivative* (backward pass) (Darwiche, 2000; Peharz et al., 2015). We thus assume that the number of operations needed to compute all messages is approximately twice the number of operations needed for a bottom-up circuit evaluation.

3.4. Merging Trick and Conditioning

Even when using modern knowledge compilers, constructing the SDD described above is prohibitively expensive. Hence, we aid the SDD compiler by performing a number of pre-processing steps. First, we apply a known technique in the SCA literature, called the *merging trick* (Guo et al., 2020): For example, in the SASCA factor graph (Figure 2), x_{12} and $\tilde{x}_{12}$ are connected via a deterministic bijection $f_{\text{XTIME}}(x_{12}) = \tilde{x}_{12}$. Thus, when computing key marginals, there is no need to model both variables in the factor graph—instead, we aggregate the leakage distributions of these variables by updating the local beliefs $p(x_{12} = x|\boldsymbol{\ell}) \leftarrow p(x_{12} = x|\boldsymbol{\ell}) \cdot p(\tilde{x}_{12} = f_{\text{XTIME}}(x)|\boldsymbol{\ell})$ for all $x \in \{0, \ldots, 255\}$. We remove the node $\tilde{x}_{12}$ from the factor graph and omit introducing byte indicator variables for it (or multiplying the PSDD that represents $p(\tilde{x}_{12}|\boldsymbol{\ell})$). We repeat this trick to remove $\tilde{x}_{23}, \tilde{x}_{34}$, and $\tilde{x}_{41}$.

Moreover, notice that if we condition on g, all XOR factors connected to g become indicator functions $\text{XOR}_g(x, x')$ that evaluate to 1 iff $x \oplus g = x'$. Thus, we can leverage the merging trick to remove $x'_{12}, x'_{23}, x'_{34}, x'_{41}$ by combining their local beliefs with the distributions of their corresponding left neighbors $\tilde{x}_{12}, \ldots, \tilde{x}_{41}$. In the same way, we remove x_{34}, x_{41} by updating the beliefs of x_{12} and x_{23}, respectively. As this simplifies the inference problem drastically, we condition $SDD(\mathcal{M})$ on a fixed $g \in \{0, \ldots, 255\}$ and only add byte indicator variables for 10 bytes in total (corresponding to 2560 auxiliary byte indicators). In the general case, we can compile a conditional SDD for each value of g (resulting in 256 SDDs) and easily combine them into a larger SDD by connecting them to an *or*-node.

However, due to the linearity of MIXCOLUMN, it suffices to compile a *single* conditional SDD (with arbitrary, fixed $g \in \{0, \ldots, 255\}$), as we can also run inference queries for different $g' \neq g$ by simply permuting some of the weight functions. This novel technique (1) reduces the time and space complexity of SDD compilation by a factor of $\approx$ 256, and (2) opens up the possibility for highly-parallel GPU-centric inference implementations. A more detailed exposition of this technique can be found in Appendix A.

Using these techniques, compiling $SDD(\mathcal{M})$ takes about 7 hours on a Intel Xeon E5-2670 CPU and the resulting circuit consists of $\approx$ 20M nodes and needs 17.4M sums and 37.2M products for a bottom-up evaluation. When evaluated 256 times, this amounts to 6 times less computations for MPE queries in the corresponding PSDD, and 3.5 times less computations when computing all marginals.

4. Experiments

To evaluate our method, we use asynchronously captured power traces from an SMT32F415 (ARM) microcontroller (Bursztein & Picod, 2019) that performs encryption using AES-128. The data $\mathcal{D} = \{(\boldsymbol{\ell}^{(i)}, \mathbf{k}^{(i)}, \mathbf{p}^{(i)})\}_{i=1}^{n}$ contains $n = 131{,}072$ samples, each of which consists of a high-dimensional power trace $\boldsymbol{\ell} \in \mathbb{R}^{20{,}000}$, and 16 byte key and plaintext vectors $\mathbf{k}, \mathbf{p}$, with 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.

4.1. Evaluation

As usual in literature, we model only the first round of AES, thus we can independently attack all 4 branches of MIXCOLUMN and reason about the corresponding 4-byte subkeys independently. For a given leakage $\boldsymbol{\ell}$, we utilize the template attack described in (Bronchain et al., 2021) to obtain a set of local beliefs $p(v \mid \boldsymbol{\ell})$ for all bytes $v \in \mathbf{v}$ and for each call to MIXCOLUMN. Using these beliefs, SASCA and ExSASCA + MAR compute marginal posterior distributions $p(k_1 \mid \boldsymbol{\ell}), \ldots, p(k_{16} \mid \boldsymbol{\ell})$ and then define the joint key posterior to be $p(\mathbf{k}|\boldsymbol{\ell}) = \prod_{i=1}^{16} p(k_i|\boldsymbol{\ell})$. On the other hand, ExSASCA + MPE does *not* make this conditional independence assumption and directly computes $\operatorname{argmax}_{\mathbf{k}} p(\mathbf{k} \mid \boldsymbol{\ell})$, i.e., the MPE in the *true joint key posterior* according to our probabilistic model $p(\mathbf{k} \mid \boldsymbol{\ell}) = p(\mathbf{k}_{1:4} \mid \boldsymbol{\ell}) \cdots p(\mathbf{k}_{13:16} \mid \boldsymbol{\ell})$, where $\mathbf{k}_{i:j} = (k_i, k_{i+1}, \ldots, k_j)$ denotes a subkey.

To simulate *protected* AES implementations, we experiment with corrupting the local belief distributions: For $\alpha \in [0, 1]$ and some byte $v \in \mathbf{v}$, we compute $\tilde{p}_\alpha(v = x \mid \boldsymbol{\ell}) = (1 - \alpha)p(v = x \mid \boldsymbol{\ell}) + \alpha\frac{1}{256}$ for all $x \in \{0, \ldots, 255\}$ and use the set of corrupted beliefs $\tilde{p}_\alpha(v \mid \boldsymbol{\ell})$ as inputs to different inference methods. In this scenario, we cannot effectively leverage sparse approximations and thus, choose $\varepsilon = 0$ in this experiment.

Table 1. Top-1 success rate of different inference methods, measured on the validation dataset $\mathcal{D}_{\text{val}}$. ε controls the level of sparsity in the approximated distributions, where $\varepsilon = 0$ means no approximation (utilizing the method in Section 3.3).

Inference Method	Top-1 Success Rate		
	$\varepsilon = 10^{-2}$	$\varepsilon = 10^{-8}$	$\varepsilon = 0$
Baseline	0.10%	0.10%	0.10%
SASCA (3 iters)	0.52%	0.52%	0.52%
SASCA (50 iters)	32.34%	33.81%	33.76%
SASCA (100 iters)	34.25%	36.04%	35.84%
ExSASCA + MAR	57.36%	67.37%	67.37%
ExSASCA + MPE	**57.51%**	**67.60%**	**67.61%**

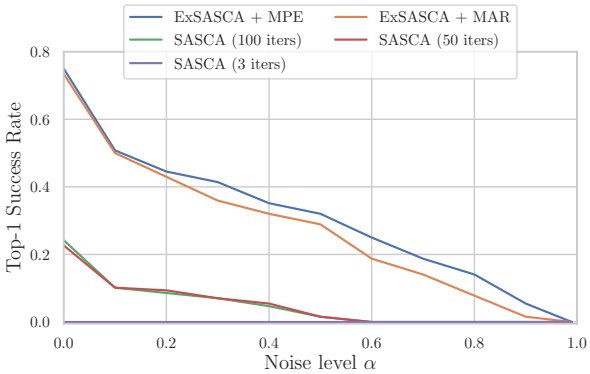

Figure 4. Top-1 success rate of different inference methods when using corrupted beliefs $\tilde{p}_\alpha(v \mid \boldsymbol{\ell})$, computed on a batch of 128 traces in $\mathcal{D}_{\text{val}}$ ($\varepsilon = 0$). As $\alpha \to 1$, the local beliefs become increasingly uninformative.

For evaluation we use *top-1 success rate*: Given a single leakage $\boldsymbol{\ell}$ and the corresponding 16-byte key $\mathbf{k}^*$ and plaintext $\mathbf{p}$, we define an attack to be *successful* if $\mathbf{k}^* = \operatorname{argmax}_{\mathbf{k}} p(\mathbf{k}|\boldsymbol{\ell})$, i.e., the true key has the highest probability in the joint key posterior. The *top-1 success rate* is the fraction of successful attacks.

4.2. Results

While SASCA always computes approximate marginals over key bytes, we can use our circuit to compute both (1) exact key marginals (ExSASCA + MAR) and (2) exact most probable evidence assignments directly in the joint key posterior (ExSASCA + MPE). As a baseline, we run BP only in the acyclic factor graph given by the AddRoundKey and SubBytes routines, i.e., we neglect all local beliefs over bytes involved in the computation of MixColumn.

The results in Table 1 show that (1) both ExSASCA variants systematically and substantially outperform all SASCA runs—even when ExSASCA uses sparse approximations and SASCA does not, and (2) directly computing the MPE in the joint key posterior yields a slightly higher success rate. Moreover, Figure 4 demonstrates empirically that as the local beliefs become less informative (e.g. due to noise), SASCA effectively fails to attack the system while the ExSASCA variants still show success rates of about 20%.

4.3. Computational Complexity

When $\varepsilon > 0$, the computational cost of ExSASCA depends on both ε and the entropy of the distributions $p(v \mid \boldsymbol{\ell})$. In our experiments, we find that the runtime of ExSASCA with $\varepsilon \geq 10^{-8}$ is still very competitive to a high-performance implementation of SASCA (Cassiers & Bronchain, 2023): For a single trace, a SASCA takes tens to hundreds of milliseconds on a modern laptop CPU, while our PSDD multiplication chain takes hundreds of milliseconds for $\varepsilon = 10^{-2}$ and seconds for $\varepsilon = 10^{-8}$. In this regime, both SASCA and ExSASCA need three orders of magnitudes less compute than the naïve exhaustive inference routine. Even without approximations ($\varepsilon = 0$), ExSASCA + MPE can perform exact inference with 6 times less operations than its exhaustive counterpart, while ExSASCA + MAR needs 3.5 times less operations.

5. Conclusion

In this work, we introduce ExSASCA, a fast and exact probabilistic inference algorithm for SASCA and demonstrate that ExSASCA substantially outperforms SASCA when attacking the Advanced Encryption Standard (AES). While SASCA uses loopy belief propagation to compute approximate marginal posteriors over the key bytes, we instead choose to represent the loopy part of the factor graph as a high-dimensional factor, which—conceptually—allows us to perform message passing on a tree. To efficiently compute messages which involve this factor, we compile it into a tractable probabilistic model that allows us to compute marginals and most probable evidence (MPE) queries in polynomial time of the circuit size. In particular, we develop two distinct approaches for compilation: (i) leveraging sparsity in the beliefs about intermediate computations during compilation, we frame the task as a combination of an off-line compilation phase and an on-line sequence of PSDD multiplications. (ii) for dense belief distributions, we present a novel dynamic compilation pipeline that produces an SDD that can perform exact inference with 6 times less operations than exhaustive enumeration. We posit that our method opens a new avenue to study both side-channel attacks and cryptanalysis using the framework of tractable (probabilisitic) models and may lead to stronger theoretical guarantees in cryptographic systems.

References

Ahmed, K., Teso, S., Chang, K., den Broeck, G. V., and Vergari, A. Semantic probabilistic layers for neuro-symbolic learning. In *NeurIPS*, 2022.

Bronchain, O., Cassiers, G., and Standaert, F. Give me 5 minutes: Attacking ASCAD with a single side-channel trace. *IACR Cryptol. ePrint Arch.*, pp. 817, 2021.

Bursztein, E. and Picod, J.-M. A hacker guide to deep learning based side channel attacks. In CON, D. (ed.), *DEF CON 27*, 2019.

Cassiers, G. and Bronchain, O. Scalib: A side-channel analysis library. *J. Open Source Softw.*, 8(86):5196, 2023.

Chari, S., Rao, J. R., and Rohatgi, P. Template attacks. In Jr., B. S. K., Koç, Ç. K., and Paar, C. (eds.), *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pp. 13–28. Springer, 2002.

Chavira, M. and Darwiche, A. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172 (6-7):772–799, 2008.

Choi, A. and Darwiche, A. Dynamic minimization of sentential decision diagrams. In desJardins, M. and Littman, M. L. (eds.), *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, pp. 187–194. AAAI Press, 2013.

Choi, A., Kisa, D., and Darwiche, A. Compiling probabilistic graphical models using sentential decision diagrams. In van der Gaag, L. C. (ed.), *Symbolic and Quantitative Approaches to Reasoning with Uncertainty - 12th European Conference, ECSQARU 2013, Utrecht, The Netherlands, July 8-10, 2013. Proceedings*, volume 7958 of *Lecture Notes in Computer Science*, pp. 121–132. Springer, 2013.

Choi, Y., Vergari, A., and Van den Broeck, G. Probabilistic circuits: A unifying framework for tractable probabilistic models. 10 2020.

Daemen, J. and Rijmen, V. The block cipher rijndael. In Quisquater, J. and Schneier, B. (eds.), *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*, volume 1820 of *Lecture Notes in Computer Science*, pp. 277–284. Springer, 1998.

Darwiche, A. A differential approach to inference in bayesian networks. In Boutilier, C. and Goldszmidt, M. (eds.), *UAI '00: Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence, Stanford University, Stanford, California, USA, June 30 - July 3, 2000*, pp. 123–132. Morgan Kaufmann, 2000.

Darwiche, A. SDD: A new canonical representation of propositional knowledge bases. In Walsh, T. (ed.), *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pp. 819–826. IJCAI/AAAI, 2011.

Darwiche, A. and Marquis, P. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002.

de Colnet, A. and Mengel, S. A compilation of succinctness results for arithmetic circuits. *arXiv preprint arXiv:2110.13014*, 2021.

Gandolfi, K., Mourtel, C., and Olivier, F. Electromagnetic analysis: Concrete results. In Koç, Ç. K., Naccache, D., and Paar, C. (eds.), *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pp. 251–261. Springer, 2001.

Guo, Q., Grosso, V., Standaert, F., and Bronchain, O. Modeling soft analytical side-channel attacks from a coding theory viewpoint. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(4):209–238, 2020.

Kisa, D., den Broeck, G. V., Choi, A., and Darwiche, A. Probabilistic sentential decision diagrams. In Baral, C., Giacomo, G. D., and Eiter, T. (eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*. AAAI Press, 2014.

Knoll, C. Understanding the behavior of belief propagation. *CoRR*, abs/2209.05464, 2022.

Kocher, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Koblitz, N. (ed.), *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pp. 104–113. Springer, 1996.

Kocher, P. C., Jaffe, J., and Jun, B. Differential power analysis. In Wiener, M. J. (ed.), *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pp. 388–397. Springer, 1999.

Koller, D. and Friedman, N. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

Kschischang, F. R., Frey, B. J., and Loeliger, H.-A. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.

Peharz, R., Tschiatschek, S., Pernkopf, F., and Domingos, P. On Theoretical Properties of Sum-Product Networks. In Lebanon, G. and Vishwanathan, S. V. N. (eds.), *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, pp. 744–752, San Diego, California, USA, 09–12 May 2015. PMLR.

Pipatsrisawat, K. and Darwiche, A. New compilation languages based on structured decomposability. In Fox, D. and Gomes, C. P. (eds.), *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pp. 517–522. AAAI Press, 2008.

Shen, Y., Choi, A., and Darwiche, A. Tractable operations for arithmetic circuits of probabilistic models. In Lee, D. D., Sugiyama, M., von Luxburg, U., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 3936–3944, 2016.

Shen, Y., Goyanka, A., Darwiche, A., and Choi, A. Structured bayesian networks: From inference to learning with routes. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pp. 7957–7965. AAAI Press, 2019.

Spreitzer, R., Moonsamy, V., Korak, T., and Mangard, S. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Commun. Surv. Tutorials*, 20(1):465–488, 2018.

Standaert, F. Introduction to side-channel attacks. In Verbauwhede, I. M. R. (ed.), *Secure Integrated Circuits and Systems*, Integrated Circuits and Systems, pp. 27–42. Springer, 2010.

Vergari, A., Choi, Y., Peharz, R., and Van den Broeck, G. Probabilistic circuits: Representations, inference, learning and applications. *AAAI Tutorial*, 2020.

Veyrat-Charvillon, N., Gérard, B., and Standaert, F. Soft analytical side-channel attacks. In Sarkar, P. and Iwata, T. (eds.), *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pp. 282–296. Springer, 2014.

Algorithm 1 Simplified Beginning of AES-128

Input: Key bytes $k_1, \ldots, k_{16}$, Plaintext bytes $p_1, \ldots, p_{16}$
for i **in** $1, \ldots, 16$ **do**
  $y_i \leftarrow k_i \oplus p_i$
  $x_i \leftarrow S(y_i)$
end for
for i **in** $0, 4, 8, 12$ **do**
  $x_{i+1}^{(m)}, \ldots, x_{i+4}^{(m)} \leftarrow \textsc{MixColumn}(x_{i+1}, \ldots, x_{i+4})$
end for

Algorithm 2 MIXCOLUMN

Input: Input bytes $x_1, \ldots, x_4$
$x_{12},\ x_{23},\ x_{34},\ x_{41} \leftarrow (x_1 \oplus x_2), (x_2 \oplus x_3), (x_3 \oplus x_4), (x_4 \oplus x_1)$
$g \leftarrow x_{12} \oplus x_{34}$
$\tilde{x}_{12},\ \tilde{x}_{23},\ \tilde{x}_{34},\ \tilde{x}_{41} \leftarrow \textsc{xtime}(x_{12}), \textsc{xtime}(x_{23}), \textsc{xtime}(x_{34}), \textsc{xtime}(x_{41})$
$x'_{12},\ x'_{23},\ x'_{34},\ x'_{41} \leftarrow (\tilde{x}_{12} \oplus g), (\tilde{x}_{23} \oplus g), (\tilde{x}_{34} \oplus g), (\tilde{x}_{41} \oplus g)$
$x_1^{(m)},\ x_2^{(m)},\ x_3^{(m)},\ x_4^{(m)} \leftarrow (x_1 \oplus x'_{12}), (x_2 \oplus x'_{23}), (x_3 \oplus x'_{34}), (x_4 \oplus x'_{41})$
Return: $x_1^{(m)},\ x_2^{(m)},\ x_3^{(m)},\ x_4^{(m)}$

A. Single-Circuit Compilation

As detailed in Section 3.4, we compile $\mathcal{M}$ conditioned on some arbitrary, but fixed $g \in \{0, \ldots, 255\}$, denoted $SDD(\mathcal{M} \mid g)$. Consider the set of models $\mathcal{X}_g = \{\mathbf{v} \text{ s.t. } (\mathcal{M} \mid g)(\mathbf{v}) = 1\}$. Since $g = x_1 \oplus x_2 \oplus x_3 \oplus x_4$, it is easy to see that g partitions the set of *all* models into equally sized sets, i.e., $|\mathcal{X}_g| = |\mathcal{X}_{g'}|$ for all byte-valued g, g'. Thus, there exists a *bijection* $\phi_{g \to g'} : \mathcal{X}_g \to \mathcal{X}_{g'}$ that maps a model from $\mathcal{X}_g$ to a model in $\mathcal{X}_{g'}$. Importantly, due to the linear structure of MIXCOLUMN, we can define such a bijection by *independently* mapping the individual bytes $v \in \mathbf{v}$:

$$(\phi_{g \to g'}(\mathbf{v}))_i = \phi_{g \to g'}^{v_i}(v_i) \tag{3}$$

Let $\mathbf{v}_g \in \mathcal{X}_g$. Since we can use the merging trick detailed in Section 3.4, it suffices to consider the subset of bytes in $\mathbf{v}_g$, namely $\hat{\mathbf{v}}_g = (x_1, x_2, x_3, x_4, x_{12}, x_{23}, x_1^{(m)}, x_2^{(m)}, x_3^{(m)}, x_4^{(m)})^\top$. To map this to some $\hat{\mathbf{v}}_{g'}$, we set

- $\phi_{g \to g'}^{x_4}(x_4) = x_4 \oplus g \oplus g'$
- $\phi_{g \to g'}^{x_1^{(m)}}(x_1^{(m)}) = x_1^{(m)} \oplus g \oplus g'$
- $\phi_{g \to g'}^{x_2^{(m)}}(x_2^{(m)}) = x_2^{(m)} \oplus g \oplus g'$
- $\phi_{g \to g'}^{x_3^{(m)}}(x_3^{(m)}) = x_3^{(m)} \oplus f_{\textsc{xtime}}(g \oplus g')$
- $\phi_{g \to g'}^{x_4^{(m)}}(x_3^{(m)}) = x_3^{(m)} \oplus f_{\textsc{xtime}}(g \oplus g') \oplus g \oplus g'$

and $\phi_{g \to g'}^{v}(v) = v$ for all remaining bytes $v \in \hat{\mathbf{v}}_g$. Consequently, we can exploit this fact to compute a weighted model count over $\mathcal{X}_{g'}$ using $SDD(\mathcal{M} \mid g)$ with $g' \neq g$ by (1) performing the merge trick with g' instead of g, and (2) by constructing a new weight function w' that first applies the corresponding byte-wise bijection before invoking the original weight function w, i.e., $w'(v) = w\left(\phi_{g \to g'}^{v}(v)\right)$. Thus, we keep the circuit $SDD(\mathcal{M} \mid g)$ fixed and compute the weighted model count 256 times, where each computation is performed with a different weight function.