Operation Count as a Performance Predictor: An Empirical Study of Lightweight AEAD Schemes

Raphaël Wintersdorff; Renaud Pacalet

COMMUNICATIONS ET ÉLECTRONIQUE [COMELEC]

Télécom Paris
Palaiseau, France

Laurent Sauvage

SECURE AND SAFE HARDWARE [SSH]

Télécom Paris
Palaiseau, France

Abstract—**This paper presents an analytical model for estimating the performance of cryptographic algorithms based on operation counting at the specification level. Aside from the requirement of having 32-bit operations available on the target, the proposed approach avoids platform-dependent assumptions by deriving a cost metric directly from the number of operations required by an algorithm description. This enables early-stage, target-independent performance comparison. The model is evaluated using the NIST lightweight cryptography competition benchmark results, where it is shown to accurately approximate cycle counts across a diverse set of algorithms. The results indicate that operation counting can provide a consistent relative ordering of computational cost, and even a good order of magnitude approximation, despite its simplicity compared to empirical measurement methods. In addition, the model is applied to algorithms protected against side-channel attacks using first-order Boolean masking schemes implemented with ISW gadgets. While the method has not yet been formally validated for masked implementations, it is used as an exploratory indicator of the additional overhead introduced by the masking countermeasure. Overall, this work suggests that specification-level operation counting can serve as a practical and low-cost tool for early performance estimation and comparative analysis of unprotected, and possibly masked cryptographic implementations.**

***Keywords*-performance estimation; lightweight cryptography; boolean masking; software**

I. INTRODUCTION

Performance estimation of cryptographic algorithms is a recurring challenge in both academic research and practical system design. Accurate evaluation is particularly important in the context of lightweight cryptography, where algorithms are expected to operate under strict constraints on computation, memory, and power consumption. Existing evaluation methodologies can broadly be divided into two categories: algorithmlevel analytical evaluation and implementation-based performance estimation. Analytical evaluations of lightweight cryptographic algorithms typically report counts of elementary operations, such as logical, arithmetic, or memory operations for software implementations, or hardware area and Gate Equivalents (GEs) for hardware implementations [7], [18]. While these metrics are useful for comparing algorithmic complexity, they are not generally intended to estimate execution time. Conversely, execution-time estimation is usually performed from an implementation using compiler analyses, instructionset simulators, or empirical benchmarking [6], [17], [20].

In this paper, we propose a performance estimation model that bridges these two levels of evaluation. The model is based on a simple yet systematic principle: counting the elementary operations derived directly from the algorithmic specification and translating these counts into an estimate of execution cost. The underlying premise is that, for a broad class of cryptographic primitives, the structure of the specification provides a sufficiently faithful proxy for computational cost. Unlike compiler-based estimation frameworks, instruction set simulators, or platform-specific benchmarking, the proposed approach requires neither source code nor executable implementations, making it suitable for use at the earliest stages of algorithm development.

We evaluate the validity of this model using the National Institute of Standards and Technology (NIST) lightweight cryptography competition benchmark results. In our analysis, we consider the primary AEAD members of Ascon [11], GIFTCOFB [1], Grain [14], ISAP [10], PHOTON-Beetle [2], Romulus [13], Sparkle [3], TinyJambu [21], and Xoodyak [9]. These algorithms provide a diverse and representative set of modern lightweight primitives, allowing us to compare specificationbased estimates against publicly available benchmark results. Our findings indicate that, despite its simplicity, the proposed model successfully captures the relative computational cost of different algorithms for a given input size and even provides a good approximation of the measured execution time of software implementations.

Beyond unprotected implementations, we further investigate the applicability of the model to implementations protected against side-channel attacks. In particular, we consider first-order Boolean masking schemes implemented using Ishai–

Manuscript received May 11, 2026; revised July 22, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.138

Sahai–Wagner (ISW) gadgets [15]. Such countermeasures introduce substantial computational overhead due to the additional operations and randomness generation, whose impact is generally difficult to assess without a complete implementation-level analysis. Although only a limited number of masked implementations are publicly available for experimental validation, we apply the proposed model as a preliminary indicator of performance for masked implementations.

The contributions of this work are threefold: (i) we propose a specification-level analytical model that estimates implementation performance directly from the algorithm description, without requiring any implementation, (ii) we validate the model against benchmark results from the NIST Lightweight Cryptography competition, demonstrating that it provides meaningful execution-time estimates despite its simplicity, and (iii) we extend the methodology to masked implementations as a preliminary exploratory study. While the proposed model is intentionally simple and is not intended to replace implementation-level benchmarking or cycle-accurate simulation, it provides an early-stage analytical tool for performance assessment, when implementation-based evaluation is not yet possible.

In this paper, we use the following symbols:

$\oplus$	xor operator, exclusive disjunction
$\wedge$	and operator, conjunction
$\neg$	not operator, negation
$A \ggg n$	rot operation, rotation of A n bits to the right
$A \gg n$	shift operation, bit shift of A n bits to the right
$A \ll n$	shift operation, bit shift of A n bits to the left
$A \| B$	concatenation of A and B
$a \leftarrow b$	assignment statement, where variable a takes the value of b

We also use other operators that do not require symbols, namely the inclusive disjunction or, the exclusive disjunction with a constant xor.c, the conjunction with a constant and.c, the modular addition add, and the table lookup lookup.

II. Approximation of performance

In our analysis, we considered 9 algorithms among the 10 finalists of the NIST lightweight cryptography competition. We only removed Elephant [5] from our analysis, since it has a performance dominated by bit permutation operations whose computational cost is highly sensitive to implementation-level optimization strategies. Since the proposed model relies on a static operation-counting abstraction at the specification level, such implementation-dependent effects cannot be consistently and fairly represented. Including this algorithm would therefore introduce a bias unrelated to the intended abstraction level of the analysis.

For every of these algorithms, we counted the number of arithmetic and logic operations that would be performed in the calculation of the outputs, for 32-bit software implementations, based on the specifications given when available. Since some specifications are not naturally amenable to 32-bit operations, we adapted them to only use 32-bit variables by strategically packing some variables in the specification.

We show hereafter an example of operation count on Ascon. While Ascon is natively specified using 64-bit words, its implementation on 32-bit platforms requires an appropriate representation of these state variables. A straightforward approach consists in splitting each 64-bit word into two 32-bit halves, storing the most and least significant bits separately. However, this representation introduces additional complexity for operations involving bit positions from both halves, particularly rotations, which may require multiple instructions and negatively impact performance. To enable more efficient 32-bit implementations, we consider the bit-interleaved representation originally introduced by Bertoni, Daemen, Peeters, and Van Assche for Keccak [4], and later used in reference implementations of Ascon. In this representation, even and odd bits of a 64-bit word are separated and stored in two 32-bit words. This arrangement simplifies the implementation of rotations and other bitwise operations on 32-bit architectures by avoiding costly cross-word manipulations. In this work, bit interleaving is considered as an existing implementation technique for adapting 64-bit cryptographic specifications to narrower architectures. Its use for Ascon is therefore not a contribution of this paper. However, describing this representation is necessary for our operation-count model, since the sequence and number of elementary operations depend on the selected representation, which is not explicitly defined in the Ascon specification.

The 320-bit state S_{BI} of Ascon in its interleaved form can be written as the concatenation of 10 32-bit words:

$$S_{BI} = S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \quad \textbf{(1)}$$

Each round, a constant is added to $S_{2,e}$ and $S_{2,o}$, which costs 2 xor.c.

The bit-interleaved version of the substitution layer can then be done with the following sequence of instructions:

$$\begin{cases} S_{0,*} \leftarrow S_{0,*} \oplus S_{4,*} \\ S_{2,*} \leftarrow S_{2,*} \oplus S_{1,*} \\ S_{4,*} \leftarrow S_{4,*} \oplus S_{3,*} \\ m_{0,*} \leftarrow (\neg S_{0,*}) \wedge S_{1,*} \\ m_{1,*} \leftarrow (\neg S_{1,*}) \wedge S_{2,*} \\ m_{2,*} \leftarrow (\neg S_{2,*}) \wedge S_{3,*} \\ m_{3,*} \leftarrow (\neg S_{3,*}) \wedge S_{4,*} \\ m_{4,*} \leftarrow (\neg S_{4,*}) \wedge S_{0,*} \\ S_{0,*} \leftarrow S_{0,*} \oplus m_{1,*} \\ S_{1,*} \leftarrow S_{1,*} \oplus m_{2,*} \\ S_{2,*} \leftarrow S_{2,*} \oplus m_{3,*} \\ S_{3,*} \leftarrow S_{3,*} \oplus m_{4,*} \\ S_{4,*} \leftarrow S_{4,*} \oplus m_{0,*} \\ S_{1,*} \leftarrow S_{1,*} \oplus S_{0,*} \\ S_{3,*} \leftarrow S_{3,*} \oplus S_{2,*} \\ S_{0,*} \leftarrow S_{0,*} \oplus S_{4,*} \\ S_{2,*} \leftarrow \neg S_{2,*} \end{cases} \quad \textbf{(2)}$$

for $* \in \{e, o\}$.

This substitution layer costs 10 and, 22 xor and 12 not.

Lastly, the linear diffusion layer uses rotations on the words of the state to produce the output. It relies on the use of linear functions $\Sigma_{i,e}$ and $\Sigma_{i,o}$, defined as follows:

$$\begin{aligned} \Sigma_{0,e}(S_{0,e}, S_{0,o}) &= S_{0,e} \oplus ((S_{0,o} \oplus (S_{0,e} \ggg 5)) \ggg 9 \\ \Sigma_{0,o}(S_{0,e}, S_{0,o}) &= S_{0,o} \oplus ((S_{0,e} \oplus (S_{0,o} \ggg 4)) \ggg 1 \\ \Sigma_{1,e}(S_{1,e}, S_{1,o}) &= S_{1,e} \oplus ((S_{1,o} \oplus (S_{1,o} \ggg 11)) \ggg 1 \\ \Sigma_{1,o}(S_{1,e}, S_{1,o}) &= S_{1,o} \oplus ((S_{1,e} \oplus (S_{1,e} \ggg 11)) \ggg 2 \\ \Sigma_{2,e}(S_{2,e}, S_{2,o}) &= S_{2,e} \oplus (S_{2,o} \oplus (S_{2,e} \ggg 3)) \\ \Sigma_{2,o}(S_{2,e}, S_{2,o}) &= S_{2,o} \oplus ((S_{2,e} \oplus (S_{2,o} \ggg 2)) \ggg 1 \\ \Sigma_{3,e}(S_{3,e}, S_{3,o}) &= S_{3,e} \oplus ((S_{3,e} \oplus (S_{3,o} \ggg 3)) \ggg 5 \\ \Sigma_{3,o}(S_{3,e}, S_{3,o}) &= S_{3,o} \oplus ((S_{3,o} \oplus (S_{3,e} \ggg 4)) \ggg 5 \\ \Sigma_{4,e}(S_{4,e}, S_{4,o}) &= S_{4,e} \oplus ((S_{4,o} \oplus (S_{4,o} \ggg 17)) \ggg \\ \Sigma_{4,o}(S_{4,e}, S_{4,o}) &= S_{4,o} \oplus ((S_{4,e} \oplus (S_{4,e} \ggg 17)) \ggg \end{aligned} \quad \textbf{(3)}$$

The state is then updated by using these calculations:

$$(S_{i,e}, S_{i,o}) \leftarrow (\Sigma_{i,e}(S_{i,e}, S_{i,o}), \Sigma_{i,o}(S_{i,e}, S_{i,o})), \quad \textbf{(4)}$$

for $i \in [0\,;4]$.

This linear diffusion layer costs 20 xor and 19 rot.

We thus have a total per-round cost of 10 and, 42 xor, 2 xor.c, 12 not, and 19 rot.

Using the same methodology, we counted the different operations used in the other schemes considered. Table **I** shows the number of operations for each of the main primitives used in the different schemes. There are other functions used in the AEAD modes whose operation count is not included in this paper, such as LFSRs or key schedules for example, but they were taken into account in our final model that we describe hereafter.

In order to perform a comparison between the respective modes of the 9 ciphers considered, we first expressed the number of operations of the modes in terms of the number of operations of the different functions that compose each cipher. For example, if we denote by A_b the number of b-bit words of associated data sent, P_b, the number of b'-bit words of plaintext sent, and $xor_{b''}$ the cost of a b''-bit xor, then the total number of operations of Ascon-AEAD128, the primary member of the Ascon family chosen to become the standard for lightweight cryptography, can be expressed as $A_{128} \cdot (p^b + xor_{128}) + (P_{128} - 1) \cdot p^b + P_{128} \cdot xor_{128} + 2 \cdot p^a + 3 \cdot xor_{128} + xor_1$(Figure 1).

We give the list of number of operations for all the modes considered hereafter:

$$\begin{aligned} C_{\text{Ascon}} = {} & A_{128} \cdot (p^b + xor_{128}) \\ & + (P_{128} - 1) \cdot p^b + P_{128} \cdot xor_{128} \\ & + 2 \cdot p^a + 3 \cdot xor_{128} + xor_1 \\ C_{\text{GIFT}} = {} & A_{128} \cdot (mult2 + G + E_k + 2 \cdot xor_{128}) \\ & + P_{128} \cdot (mult2 + G + E_k + 3 \cdot xor_{128}) \\ & + \text{KeySchedule} \\ C_{\text{Grain}} = {} & (A_{32} + P_{32}) \cdot (2(NFSR + LFSR + y_t) + a_t + r_t) \\ & + \text{Init} \\ C_{\text{ISAP}} = {} & P_{64} \cdot p^6 + (A_{64} + P_{64} + 2) \cdot p^{12} \\ & + (A_{64} + 2P_{64}) \cdot xor_{64} + 4 \cdot p^6 + 254 \cdot p^1 \\ & + 256 \cdot xor_1, \text{where p} = \text{Ascon-p} \\ C_{\text{PHOTON}} = {} & (A_{128} + P_{128} + 1) \cdot f + P_{128} \cdot \rho \\ & + (A_{128} + 2) \cdot xor_{128} \\ C_{\text{Romulus}} = {} & (\lfloor \frac{A_{128}}{2} + P_{128} + 1 \rfloor) \cdot (E_k + \rho) \\ & + \text{TweakeySchedule} \\ C_{\text{Sparkle}} = {} & (A_{256} - 1 + P_{256} - 1) \cdot (S_7 + \rho) \\ & + (A_{256} + P_{256}) \cdot xor_{256} \\ & + 3 \cdot xor_{128} + 2 \cdot S_{11} + 2 \cdot \rho \\ C_{\text{TinyJambu}} = {} & A_{32} \cdot (p_{384} + xor_{32}) \\ & + (P_{32} + 2) \cdot (p_{1024} + 2 \cdot xor_{32}) \\ & + 4 \cdot p_{384} + 6 \cdot xor_3 \\ C_{\text{Xoodyak}} = {} & (A_{352} + P_{192} + 1) \cdot p^{12} + A_{352} \cdot (xor_{352} \\ & + xor_{32}) + 2P_{192} \cdot xor_{192} \end{aligned}$$

Note that the use of the terms $(P_{128} - 1) \cdot p^b$ for Ascon-AEAD128 and $(A_{256} - 1 + P_{256} - 1) \cdot (S_7 + \rho)$ for Sparkle is

Table I: Operation counts for the main primitives

Operation	Bloc ciphers		Permutations						Stream cipher
	GIFT-128 (GIFT-COFB)	Skinny (Romulus)	p^r (Ascon)	p^r (ISAP)	PHOTON$_{256}$ (PHOTON-Beetle)	SPARKLE384 (Sparkle)	P_n (TinyJambu)	XOODOO (Xoodyak)	NFSR (Grain)
and	15	-	10	10	-	-	1	12	14
or	5	-	-	-	-	-	-	-	-
xor	56	44	42	42	56	42	8	36	42
and.c	28	28	-	-	56	-	-	-	-
xor.c	5	3	2	2	8	26	-	1	-
not	5	-	12	12	-	-	1	12	-
shift	28	24	-	-	56	2	8	-	54
rot	-	-	19	19	-	44	-	20	-
add	-	-	-	-	-	24	-	-	-
lookup	-	16	-	-	64	-	-	-	-
Sum	142	115	85	85	240	138	18	81	110
#Rounds	8	40	8/12	1/6/12	12	7/11	12/32	12	-
Total Ops	1136	4600	680/1020	85/510/1020	2880	966/1518	216/576	972	110

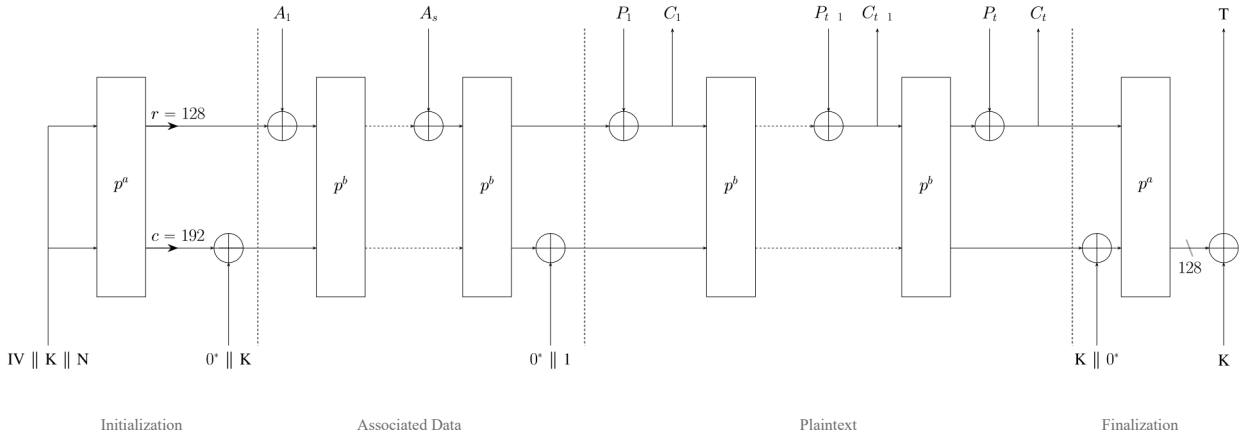

Fig. 1: Graphical representation of the Ascon-AEAD128 mode of operation.

The rate r being equal to 128, associated data and plaintext blocks are treated as 128-bit words. Associated-data absorption contributes A_{128} calls to p^b and A_{128} 128-bit xor, while plaintext processing contributes $P_{128} - 1$ additional calls to p^b and P_{128} 128-bit xor. There are also 2 calls to p^a, in the Initialization and Finalization phases. Finally, there are 3 exclusive or in the bottom branch, with a capacity c 192, but since one of the operands is padded with zeros, this can be considered as a 128-bit xor. Similarly, the addition with $0^* \| 1$ is modeled as a 1-bit xor.

an abuse of notation, used to improve readability: if no plaintext or associated data is provided, terms of the form $P_* - 1$ or $A_* - 1$ should be considered as zero, and not a negative number.

With these expressions, we have access to a platform-agnostic model that can estimate the execution time of the algorithms we considered for any input lengths.

In order to verify the soundness of our method, we made some statistical tests between a list of estimated number of cycles, given by our model for different lengths of associated data and plaintext, and a list of actual number of cycles given by the performance benchmarking results from the NIST's Github page (https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking/tree/main). The results shown in the paper are those from the Arduino Nano 33 BLE (nRF52840) target architecture, but we performed the same experiments on all platforms available for the benchmark. The benchmark consists of cycle counts for different implementations of the finalists of the lightweight cryptography competition organized by the NIST. The cycle counts depend on the number of bytes of associated data and of plaintext. We extracted the cycle counts for byte lengths in the set {0,8,16,32,64,128}, resulting in 36 values per implementation. We selected, for each mode, the minimal cycle count among the implementations of their primary member. For each of the 36 pairs of lengths, we calculated the Kendall rank correlation coefficient, also known

as Kendall's τ [16], which measures how similar the orderings of the implementations are in the sets of modeled and real cycle count. It is calculated as follows:

$$\tau = \frac{\#\text{correctly ordered pairs} - \#\text{incorrectly ordered pairs}}{\#\text{pairs}} \quad (5)$$

τ can be used in a statistical hypothesis test known as τ test, which yielded, in our case, a p-value below 0.01 for every input size. This value is below the usual threshold value of 0.05, allowing us to reject the null hypothesis that the two methods for finding the number of cycles are independent. However, for ease of interpretation, we also converted the results into a percentage of correct pairwise rankings, thanks to the following formula:

$$p = \frac{\tau + 1}{2} \quad (6)$$

We have a mean percentage of correct pairwise ranking of about 0.95, meaning that the model and the measurement agree about 95% of the time on the ranking of any two pairs of algorithm cycle count in the list of considered algorithms (see Figure 2 for the per-size results).

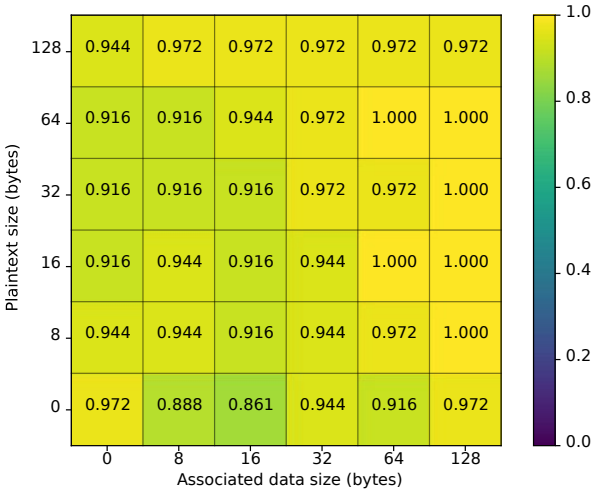

Fig. 2: Percentages of correct pairwise rankings.
Percentages were calculated per input size pair. For a given pair of input sizes, the probability of the method and the benchmark agreeing on the ordering of two algorithms in terms of cycle count is between 0.861 and 1, with an average of 0.951.

While conserving the order is a property that can be beneficial when choosing an algorithm in a list of candidates for a specific usecase, the order of magnitude of the two methods might be vastly different, which is why we also calculated the Pearson's correlation coefficient ρ [19] for every input size. The observed linear correlation across multiple algorithms suggests that the agreement between the proposed model and empirical benchmark results is not limited to a specific algorithm, since we obtained a p-value of less than 10^{-5} for every test. Given this linear relationship between real and modeled time, we made one linear regression per algorithm, on a dataset of points of the form (real cycle count, estimated cycle count), resulting in explained variance ratios ranging from 88\% to nearly 100\% (Table **II**).

Table II: Regression parameters for evaluated algorithms.

Algorithm	Slope	Intercept	R^2
Ascon	0.64232	-7.98728	0.97779
GIFT-COFB	0.81522	-1077.49161	0.98856
Grain	0.90720	-2047.76317	0.99968
ISAP	0.69126	8905.57643	0.88853
PHOTON-Beetle	0.90514	137.59719	0.99995
Romulus	0.81779	-520.36644	0.95275
Sparkle	0.73989	-1105.07928	0.96878
TinyJambu	1.07438	-857.44311	0.95319
Xoodyak	0.57563	612.48003	0.95518

The slopes of the per-algorithm regressions having a similar order of magnitude, we hypothesized that the whole set of points lies close to a single line, which we verified by making a linear regression on all the points of the form (real cycle count, *f*(estimated cycle count), with *f* being the affine transformation that results in a regression line representing the identity function. The resulting regression line has a ratio of explained variance of 99.5%, suggesting that the final model, after the affine transformation, could be a good proxy for cycle count (Figure 3). However, since the variance of the cycle counts of different algorithms could have different orders of magnitude, the ratio of explained variance alone is not sufficient to validate our method.

In order to numerically assess the accuracy of our method, we calculated the orthogonal root mean square error (RMSE) per algorithm, which is the square root of the average squared Euclidean distance between the points and their orthogonal projection on the considered line. The orthogonal RMSE of the different algorithms is relatively low compared to the spread of the evaluated data points, thus indicating that the points lie close to the line representing the identity function (Table **III**).

We tested the same methodology on all the platforms available in the NIST benchmarks. The ratio of explained variance of the resulting regression line is at least 95.3%, except for the Arduino nano every and the Arduino uno, which are the only two platforms based on 8-bit cores.

We thus deem the proposed model suitable for estimating the order of magnitude of the computational cost of algorithms This allows for a time saving when determining which algorithm has better performance in terms of execution speed, since it does not require implementing the algorithms under consideration and does not depend on the choice of a specific target, as long as it is capable of performing 32-bit operations.

However, it is important to point out that not every scheme

Table III: Per-algorithm Orthogonal RMSE and maximum distance of points in the scatter plot.

Algorithm	Orthogonal RMSE	Maximum point distance
Ascon	948	19667
GIFT-COFB	945	39849
Grain	473	95719
ISAP	3702	75756
PHOTON-Beetle	941	282089
Romulus	2201	66516
Sparkle	598	14396
TinyJambu	2370	36336
Xoodyak	1313	16603

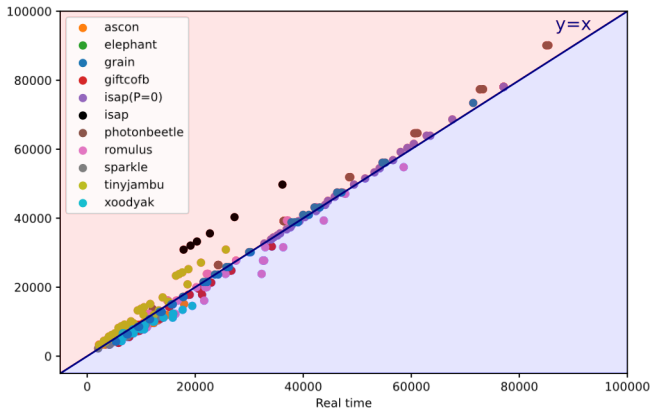

Fig. 3: Scatter plot of model time against real time. Points are mostly aligned along the regression line $y = x$, with a ratio of explained variance of 99.6%, when removing the outliers for ISAP when the plaintext is empty. This line, represented in blue, represents where points would lie if the model had perfect accuracy. The red half-plane indicates where the model suggests worse results than those in the benchmark. Conversely, the blue half-plane indicates where the model suggests better performance than the actual results.

lends itself to such an estimation. For instance, Elephant is the only AEAD scheme from the NIST lightweight cryptography competition that we did not consider in our analysis, due to the fact that the only implementation available in the NIST benchmark is in C and contains a bit permutation that is not optimized. If we were to model this implementation strategy, consisting of selecting each of the 160 bits and accumulating them in different registers, our method would yield a high estimated cost, while in reality the compiled executable runs faster than the model predicts.

It is also important to consider implementation choices that can significantly affect the operation counts. Notably, some algorithms can be implemented with the use of lookup tables, or can benefit from bitslicing. These are design choices that have a substantial impact on performance. Our model is capable of taking such choices into account, thereby enabling a rapid assessment of the potential performance of varied implementation choices.

The algorithms for which our model performs best are those that are primarily defined using arithmetic and logical operators on 32-bit variables, or that can be adapted to operate on 32-bit variables by packing bits or bytes together. Operations that are not straightforward to express in terms of 32-bit variables lead to a mismatch between the modeled cost and the actual cost of the algorithm.

While multiple implementations of the NIST lightweight cryptography competition finalists are available for the different platforms in the performance benchmarking Github repository of the NIST, there are few public implementations of these algorithms protected against side-channel attacks. We surmise that our model is applicable to protected implementations, although this claim has yet to be confirmed via practical experiments. With this assumption in mind, we explore in the rest of the paper the effect of boolean masking on the lightweight modes we considered, despite the lack of publicly available implementation.

III. Effects of Boolean Masking on the Operation Count for the Lightweight Competition Finalists

The method used in Section II for estimating the execution time of unprotected algorithms can be used for protected implementations, though it should be considered as an indicator, until further experiments are done to assess its suitability to masked schemes. Since masking introduces new variables and more computation, it might for instance affect register pressure in assembly implementations, which would in turn make calls to memory more frequent, and since these calls are not taken into account in the method, the difference between real and modeled times could deviate more than for unprotected implementations. Nevertheless, we used it to determine which masked schemes are the most promising without having to implement them all.

The extension to masked implementations is relatively straightforward: instead of having operations between variables accounting for one unit cost, we now have gadgets functionally equivalent to the unmasked operations, which themselves are made of multiple logic operations. We then just have to compute a weighted sum, where the weights are the number of operations for a given gadget (Table **IV**), and the values are the number of the different gadgets used, which is the same as the number of operations used in the unprotected implementations, assuming that no additional randomness in the form of refresh gadgets needs to be added to make the implementation secure. For simplicity, we restrict our analysis on first-order boolean masking. We chose to use the ISW-AND gadget [15] for the logical conjunction and to create the inclusive disjunction, since it allows for an easy generalization to higher-order settings, and it also allows for the creation of formal proofs of security against side-channel attacks in the probing model [15], if required. As for the affine gadgets, we use either the trivial implementation of a gadget when it is linear, and apply the affine part to the first share when it isn't.

TABLE IV: Operation counts for the gadgets considered for first-order ISW masking

Operation	ISW-AND	ISW-OR	XOR	AND.c	XOR.c	SHIFT	ROT	NOT	ADD
and	4	4	-	-	-	-	-	-	40
or	-	-	-	-	-	-	-	-	-
xor	4	4	2	-	-	-	-	-	84
and.c	-	-	-	2	-	-	-	-	-
xor.c	-	-	-	-	1	-	-	-	-
not	-	3	-	-	-	-	-	1	-
shift	-	-	-	-	-	2	-	-	20
rot	-	-	-	-	-	-	2	-	-
add	-	-	-	-	-	-	-	-	-
Sum	8	11	2	2	1	2	2	1	144
#random	1	1	-	-	-	-	-	-	10

Defining an ADD gadget for the arithmetic addition is not as straightforward, since this operation is not linear with respect to the exclusive disjunction operation xor. One way of doing that would be to use conversion algorithms to temporarily switch from boolean masking to arithmetic masking, perform the addition in the arithmetic domain, and then convert the result back to boolean masking [12]. It is also possible to avoid this back and forth by using a masked implementation of a full binary adder with the previously defined gadgets. This adder can be a ripple-carry adder, or a carry-lookahead adder such as the Kogge–Stone adder, for example [8]. However, both methods are costly in terms of both computational complexity and randomness consumption, which makes them unattractive when arithmetic additions occur frequently, as is the case in ARX-based permutations such as SPARKLE384[11]. In the rest of the paper, we consider the addition gadget to be a Kogge-Stone masked addition algorithm [8].

As for lookup tables, the conversion to a masked lookup table is less easy: for 2-sharings, if a table has 256 possible indices for instance, then its masked version would require $256^2 = 65536$ entries. The amount of computation and memory to use this kind of table would be prohibitive for lightweight applications, and scales poorly with the masking order. A solution is to fix the masks of the varibales that are used as inputs to the table, and add another mask at the output. More formally, for an input mask value $m \in E$, an output mask value $m' \in E$ and a table $T: E \to E$, we define a table $T_m: E \to E$ such that, for all $i \in E$, $T_m[i] = T[m \oplus i\,] \oplus m'$. This ensures that for a 2-sharing $[x] = (x_m, m)$, we have $T_m[x_m] = T[m \oplus x_m] \oplus m' = T[x] \oplus m'$, which is a masked value of the *secret* $T[x]$ that we aim to calculate. The use of such masked tables introduces new problems, however: since their definition depends on the values m and m′, they need to be recomputed each time the masks change. Since the reuse of mask is not advised for security purposes, the tables would need to be calculated for each encryption, negating the advantage of using them for fast calculations.

We removed the implementations that made use of tables, namely those of PHOTON-Beetle and Romulus, for which a tabulated S-box and other precomputations were used. After removal, we made regressions on the modeled costs of masked implementation. With the exception of Sparkle, whose use of arithmetic additions makes the cost overhead very large, the cost of the considered masked implementations ranges from about 2.7 to 3.7 times that of their unprotected counterpart, according to our model (Figure 4).

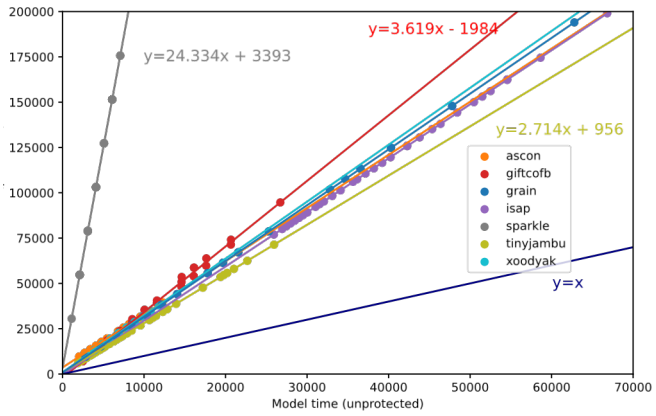

Fig. 4: Scatter plot of model time for masked implementation against model time for unprotected implementation, with removed table-based implementations. Except for Sparkle, regression lines for the different algorithms have a slope between 2.7 and 3.7. These similar values suggest that their ranking for any input size pair is

mostly unchanged compared to the ranking of the unprotected algorithms.

This indicates that for any given pair of input size, the ranking of these algorithms would not change much, except for Sparkle, which becomes quickly unattractive when the input sizes get larger, since a regression on the data points related to Sparkle yields a line $y = 24.334x + 3393$, indicating that a masked implementation for this algorithm would be approximately 24 times slower than the unprotected one.

IV. Conclusion

This paper presented a model for estimating the performance of cryptographic algorithms through operation counting derived directly from their specification. The proposed approach aims to provide a simple method for evaluating computational cost during early design and analysis stages, without relying on platform-specific benchmarking. The relevance of the model was investigated using algorithms from the NIST lightweight cryptography competition benchmark results. These include a wide range of cryptographic designs, such as block ciphers, permutation-based ciphers, and a stream cipher. Our results show that specification-level operation counting provides a useful indicator of algorithmic complexity across different lightweight cryptographic constructions.

In addition, the method was applied to implementations protected against side-channel attacks using first-order Boolean mmasking based on ISW gadgets. Although the model has not yet been experimentally validated for masked implementations, the obtained estimates suggest that it can still provide useful insight into the computational overhead introduced by masking countermeasures. This preliminary study highlights the potential of operation-based estimation techniques for comparing secure implementations at a high level of abstraction.

The simplicity of the proposed model constitutes both its main strength and its primary limitation. While it enables fast and portable evaluation, it does not capture target-dependent effects that could make the model results deviate from actual cycle counts. Future work could focus on refining the model, to take into account access to memory for instance. Another direction would be to extend the validation to additional implementation styles, algorithms, and masking strategies, and to investigate weighting strategies for different operation classes in order to improve estimation accuracy. Since our methodology does not necessarily need the operations to be between 32-bit variables, future work could be about adapting this method for 8-bit variables and verifying its applicability through benchmarks results on 8-bit platforms.

Overall, this work demonstrates that specification-level operation counting can serve as a practical and low-cost tool for performance analysis of lightweight cryptographic algorithms, and shows how it can be used to estimate the cost of their protected counterparts.

REFERENCES

[1] Subhadeep Banik, Avik Chakraborti, Akiko Inoue, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT-COFB. Cryptology ePrint Archive, Paper 2020/738, 2020.

[2] Zhenzhen Bao, Avik Chakraborti, Nilanjan Datta, Jian Guo, Mridul Nandi, Thomas Peyrin, and Kan Yasuda. Photon-beetle, 2021. Accessed: 2024-12-11.

[3] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschadl, Amir Moradi, L¨ eo Perrin, Aein Rezaei Shahmirzadi, Aleksei´ Udovenko, Vesselin Velichkov, and Qingju Wang. Schwaemm and esch: Lightweight authenticated encryption and hashing using the sparkle permutation family, 2021. Accessed: 2024-12-11.

[4] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche.¨ Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 313–314, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] Tim Beyne, Yu Long Chen, Christoph Dobraunig, and Bart Mennink. Elephant v2, 2021. Accessed: 2024-12-11.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[7] Alex Biryukov and Leo Perrin. State of the art in lightweight symmetric´ cryptography. *IACR Cryptol. ePrint Arch.*, 2017:511, 2017.

[8] Jean-Sebastien Coron, Johann Großsch´ adl, and Praveen Vadnala. Secure¨ conversion between boolean and arithmetic masking of any order. pages 188–205, 09 2014.

[9] Joan Daemen, Seth Hoffert, Silvia Mella, Michael Peeters, Gilles Van¨ Assche, and Ronny Van Keer. Xoodyak, a lightweight cryptographic scheme, 2021. Accessed: 2024-12-11.

[10] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. Isapv2.0, 2021. Accessed: 2024-12-11.

[11] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schlaffer. Ascon v1.2: Lightweight authenticated encryption and hash-¨ ing. *J. Cryptol.*, 34(3), July 2021.

[12] Louis Goubin. A sound method for switching between boolean and arithmetic masking. pages 3–15, 05 2001.

[13] Chun Guo, Tetsu Iwata, Mustafa Khairallah nad Kazuhiko Minematsu, and Thomas Peyrin. Romulus-v1.3, 2021. Accessed: 2024-12-11.

[14] Martin Hell, Thomas Johansson, Alexander Maximov, Willi Meier, Jonathan Sonnerup, and Hirotaka Yoshida.¨ Grain-128aeadv2, 2021. Accessed: 2024-12-11.

[15] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 463–481, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[16] M. G. Kendall. Rank correlation methods. 1949.

[17] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[18] Kerry McKay, Lawrence Bassham, Meltem Sonmez Turan, and Nicky Mouha. Report on lightweight cryptography, 2017-03-28 00:03:00 2017.

[19] Karl Pearson and Francis Galton. Vii. note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58(347-352):240–242, 1895.

[20] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstrom. The worst-case execution-¨ time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.

[21] Hongjun Wu and Tao Huang. Tinyjambu: A family of lightweight authenticated encryption algorithms(version 2), 2021. Accessed: 202412-11.