Small MACs from Small Permutations

Maria Eichlseder, Ahmet Can Mert, Christian Rechberger, and Markus Schofnegger

IAIK, Graz University of Technology (Austria)
firstname.lastname@iaik.tugraz.at

Abstract. The concept of lightweight cryptography has gained in popularity recently, also due to various competitions and standardization efforts specifically targeting more efficient algorithms, which are also easier to implement.
One of the important properties of lightweight constructions is the area of a hardware implementation, or in other words, the size of the implementation in a particular environment. Reducing the area usually has multiple advantages like decreased production cost or lower power consumption.
In this paper, we focus on MAC functions and on ASIC implementations in hardware, and our goal is to minimize the area requirements in this setting. For this purpose, we design a new MAC scheme based on the well-known Pelican MAC function. However, in an effort to reduce the size of the implementation, we make use of smaller internal permutations. While this certainly leads to a higher internal collision probability, effectively reducing the allowed data, we show that the full security is still maintained with respect to other attacks, in particular forgery and key recovery attacks. This is useful in scenarios which do not require large amounts of data.
Our detailed estimates, comparisons, and concrete benchmark results show that our new MAC scheme has the lowest area requirements and offers competitive performance. Indeed, we observe an area advantage of up to 30% in our estimated comparisons, and an advantage of around 13% compared to the closest competitor in a concrete implementation.

Keywords: MAC, lightweight, symmetric cryptography, permutation

1 Introduction

Lightweight cryptography is a prominent topic in modern cryptography, and some of the recent competitions in this area have focused partially or even completely on the design and analysis of lightweight cryptographic primitives. This includes the now finished CAESAR competition [12] (with Ascon [17] having been selected as the primary choice in its lightweight category) and the ongoing NIST lightweight competition [28].

Indeed, while the computing power of modern desktop CPUs is constantly increasing, the design of algorithms which have a small area footprint is especially interesting when considering hardware implementations like ASICs. In this

scenario, the area of the resulting construction can often be directly related to the production cost of the device or its power consumption.

Maximizing the efficiency of cryptographic constructions should however not result in a security loss, which is the main factor contributing to the difficulty of designing efficient algorithms. While this security is usually achieved by keeping both the time and data costs of the best-known attacks as close to the claimed security level (e.g., key size) as possible, in this paper we explore quite a different direction: We show how to retain an otherwise high security level when *deliberately losing* security against attacks which use more than a certain limited amount of data. One existing example of such an approach is the low-latency design PRINCE [11], based on the FX construction [21]. In other words, when limiting the attacker to D data, the computational complexity of key-recovery attacks and other approaches should still be exponential in κ for a κ-bit key and independent of D. This is related to the fact that in many scenarios the possibility to encrypt or authenticate a (very) large amount of data is not needed, which can be the case for a device which is used only several times. Hence, protecting a certain algorithm against attackers having access to, for example, 2^{32} transcripts or even more may not be necessary.

This strategy is motivated by hardware efficiency reasons. Indeed, cryptographic primitives with a smaller state size tend to have a reduced area footprint on hardware, and have otherwise beneficial properties.

This strategy allows us to use cryptographic primitives which are more efficient for both software and hardware implementations. In our approach, we focus on a MAC construction built from smaller permutations (e.g., 64 or 32 bits), and we reach GE numbers of around 1000 for the entire construction, depending on the primitive being used, while still providing reasonable performance. Hence, our new construction is competitive against other currently used algorithms in terms of gate area and latency.

A comparison of our design `LDMAC`[1] and various other constructions is given in Fig. 1. In this presentation, we can see that for a fixed building block (in that case SKINNY), our mode allows for the smallest area and a slightly better number of cycles. This is also similar when instantiating the modes with different primitives, as will be shown in more detail in Section 5. We emphasize that these are estimations based on the logic gates of the underlying primitive and on the additional operations necessary for our design.

1.1 Related Work

MAC functions are widely used in the area of authentication, and there are multiple proposals which focus on specific optimizations. In this section, we briefly highlight some of the related work in this area.

[1] The name "`LDMAC`" suggests that this construction is used in a *low-data* scenario, i.e., when the total number of authenticated message blocks is small.

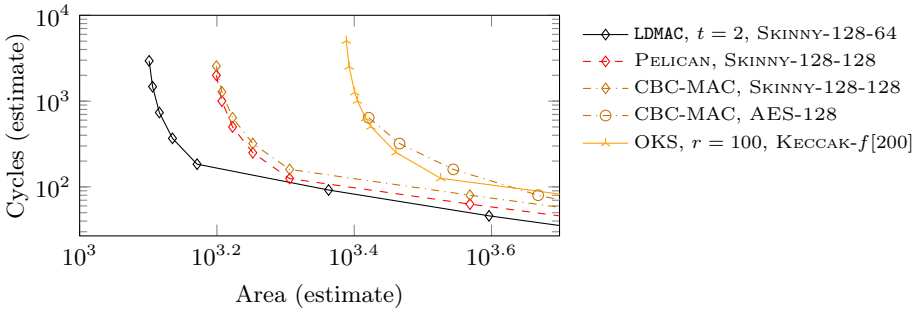

Fig. 1: LDMAC and other constructions when authenticating 512-bit messages.

Lightweight MAC Functions. Lightweight MAC functions can be built by combining established approaches, such as CBC-MAC and various sponge constructions, with lightweight primitives. However, besides these well-known combinations, which we mainly focus on later in our comparisons, there also exist designs which are specifically tailored towards certain use cases.

For example, MergeMAC [3] is a MAC design that aims to be used in scenarios with strict time requirements and on devices which only support a limited amount of bandwidth. Other functions like Chaskey [26] and SipHash [4] are more software-oriented and focus on devices with code size or speed limitations and on the authentication of small messages, respectively. Further, a MAC mode called LightMAC [23] was presented in 2016, to be used together with an (ideally lightweight) block cipher. It is built for quite a different target, namely to make the security bounds independent of the message length. The mode itself has only small requirements, making the whole construction not much larger than the used block cipher itself. As is the case in e.g. CBC-MAC, it uses one encryption call per message block. Another example is TuLP, a family of lightweight MACs presented in 2014 [18]. The authors of this paper design lightweight message authentication codes for hardware-constrained devices based on the Alred [15] construction and the lightweight Present block cipher [9]. They consider two versions, one producing a 64-bit tag and one producing a 128-bit tag for increased collision security. In particular TuLP-128 is similar to our proposal in that it also uses parallel evaluations of a block cipher. However, in order to decrease the collision probability, there are various mixing steps taking place in the construction, essentially increasing the cost.

Finally, there exist various designs recently submitted to the NIST lightweight cryptography competition. These are targeted towards authenticated encryption purposes, but of course they can also serve as (nonce-based) MAC functions by using only the authenticated data field. However, many of the NIST submissions make use of sponge functions, which naturally need larger state sizes (and hence more registers) in order to achieve their security properties.

Double-Block-Length Hash Functions. Our new construction bears similarities to double-block-length hash constructions. These functions use small-state components (for example block ciphers) in a construction providing a larger output. The goal of this approach is to achieve a security higher than that of the individual component. This can have advantages in hardware, since reusing existing building blocks while providing higher security levels becomes possible.

Double-block-length hash functions have been proposed already, and the general idea to use small-state building blocks for large-state outputs is well-known in the literature [25,19,27,20]. However, these proposals focus on offering a higher security against collisions w.r.t the small state of the building block. In our specific scenario, we do not require this property and can hence design a more efficient construction.

1.2 Contribution

We describe a new MAC construction called `LDMAC`, which we conjecture to provide a computational security level larger than the size of its underlying permutations. However, we limit the attacker to a small amount of queries, which allows us to be comparatively efficient in terms of hardware area and latency.

We do this by essentially splitting the full state into smaller substates, and applying the smaller permutation to each of these substates when processing the message. Additionally, we present a security analysis for this new construction and a comparison with similar approaches.

1.3 Organization

In the next section, we will give an overview of the properties of lightweight algorithms. We will also describe MAC functions in general and discuss the Pelican MAC function and its security, as our proposal will be closely related to it. Then, we will describe our new construction in detail, and also highlight both the similarities and differences with Pelican. After that, we will also analyze the security of our construction. Finally, we conclude with a performance comparison between our design and various other MAC functions.

2 Preliminaries

2.1 Algorithms for Constrained Devices

When evaluating a cryptographic primitive for use on a constrained device, multiple characteristics need to be considered. These include the security and the hardware performance of the algorithm. The latter is usually measured using properties like the resulting gate area, the power or energy consumption, and the latency and the throughput. Designing an algorithm which simultaneously optimizes all of these criteria is usually not possible. Therefore, tradeoffs have to be made in order to find a cryptographic primitive and mode of operation

suitable for a specific use case. For our scenario, we focus on an implementation which provides a good tradeoff between the gate area and the latency, emphasizing that our primary goal is a low gate area number.

2.2 Message Authentication Codes

A message authentication code (MAC), sometimes also referred to as a *tag*, is a numerical value used to authenticate the origin of the corresponding message. Assuming that the same symmetric key is known only to the sender and to the receiver, a MAC provides both authenticity and data integrity.

More formally, a MAC signing function $S(\cdot)$ takes as input a message m and a key k, and produces a tag T:

$$S(m, k) = T.$$

The MAC verification function $V(\cdot)$ takes as input a message m, a key k, and a tag T, and produces the result $R \in \{\text{True}, \text{False}\}$:

$$V(m, k, T) = R.$$

The key k has to be chosen randomly and shared between both parties. However, this process is out of the scope of this paper, and hence in our scenario we assume this issue has already been taken care of before.

A MAC function is considered secure if no attack is faster than exhaustively trying each possible key, and if the probability of generating a new valid tag for an arbitrary message (without requesting it) is not higher than that for just guessing the correct tag (assuming a uniform distribution of all tag values). These security notions are given in [24], and we adopt them for our proposal.

2.3 The Pelican MAC Function

The Pelican MAC function, proposed in 2005 [16] and based on Alred [15], can be seen as a predecessor of modern sponge constructions, with the rate (outer) part covering the whole state. It is built using two different permutations, a keyed one and an unkeyed one. The keyed permutation is essentially a block cipher applied right at the beginning and just before the tag output. The unkeyed permutation is used in the so-called *chaining phase*, which is reminiscent of the absorption step in sponge constructions. Note that both permutations may be based on the same round functions. Moreover, the permutation used in this chaining phase does not need to be indistinguishable from a random permutation. We will discuss this in more detail in the following.

More formally, let $\mathcal{E}_k(x)$ denote the encryption of x using the block cipher $\mathcal{E} : (\mathbb{F}_2^N \times \mathbb{F}_2^N) \to \mathbb{F}_2^N$ and the key k. Further, let $\mathcal{P} : \mathbb{F}_2^N \to \mathbb{F}_2^N$ denote the unkeyed permutation and IV the initialization input used for Pelican. Then, the tag $\text{Pelican}(m, k)$ for a u-block message m is defined as

$$\text{Pelican}(m, k) = \mathcal{E}_k(\mathcal{P}(\cdots(\mathcal{P}(\mathcal{E}_k(IV) \oplus m_1) \oplus m_2)\cdots) \oplus m_u).$$

A graphical representation of Pelican is given in Fig. 2. Pelican has also been used as a building block for other algorithms in the literature, such as ALE [10].

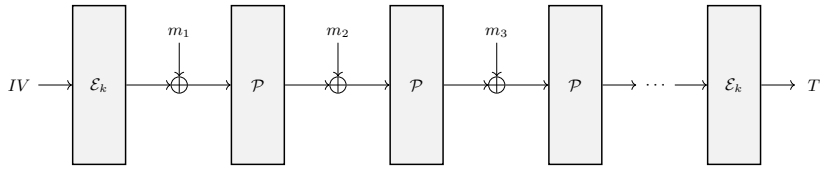

Fig. 2: The PELICAN MAC construction (slightly simplified).

Security Reasoning for PELICAN. The first component is a full block cipher evaluation. If we omit it, an attacker can locally prepare a forgery by just changing arbitrary message blocks, computing the corresponding permutations locally, and finally altering the last message block in order to cancel the difference. The resulting tag will then be equal to the tag of a previously queried message. The last component of PELICAN (ignoring the potential truncation) is another full evaluation of a block cipher. Without this last block cipher call, an attacker could simply compute backward and thus forge tags for arbitrarily chosen messages.

The main component of PELICAN is the chaining phase. Four AES [14] rounds are used in the original description, which is due to the maximum probability a differential characteristic can achieve over these four rounds. Indeed, the probability that an attacker guesses the correct difference in a block after inserting any difference into a previous block is about 2^{-128}, because in four AES rounds the 25 active S-boxes ensure that any differential characteristic has a maximum probability of $2^{-6 \cdot 25} = 2^{-150} < 2^{-128}$. This is the security property which has to be fulfilled by the unkeyed permutation, i.e., for an N-bit unkeyed permutation, the probability for any characteristic has to be at most 2^{-N}.[2]

2.4 Notation

Following the description of PELICAN in the previous section, we briefly give the notation used throughout the paper. First, N will specificy the *total* size of the state in bits (for example, $N = 128$ for PELICAN with AES). We will later use t smaller permutations with a size of n bits each, for a total state size of $N = nt$.

To denote keys, we will generally use k, and we will use $k^{(i)}$ to specify a key for some instance i. We use k_i to denote different parts of a key k (for example, the first part may be denoted by k_1). Message parts will be denoted by m_i and tag parts by T_i. Moreover, we use $\oplus$ to denote the `XOR` operation in $\mathbb{F}_2^n$, and we use $||$ to denote concatenation. We further use $[1, t]$ to denote the set of integers $\{1, 2, \ldots, t\}$. Finally, we denote the different instances of various primitives as Instance-κ-n, where κ denotes the key size and n denotes the block size.

[2] We note that in the specific case of PELICAN, this requirement is not sufficient, and indeed impossible differential attacks using less then 2^N data are possible [30]. However, the attacks presented in this work do not apply to our case, because we do not allow a data complexity higher than $2^{N/2}$.

3 Specification of LDMAC

Here we present an alternative construction, using similar ideas but providing better performance characteristics in certain scenarios. In particular, we will modify the chaining phase, and we will also make slight changes to the block cipher evaluations. The resulting mode will then be similar to a "parallelized" version of smaller PELICAN instances, with a few additional properties in order to retain the higher security level against various attacks.

The New Chaining Phase. Recall that the large-state permutation calls in PELICAN are needed in order to get a sufficiently low probability for any differential characteristic. However, in the scenario we focus on in this paper, we try to build a more efficient construction at the cost of reducing the amount of data that can be authenticated.

We hence propose a new chaining phase. Instead of using large-state permutation calls, we use several smaller-state permutation calls in parallel. This is similar to what is done in double-block-length constructions. However, compared to double-block-length constructions which try to maintain a certain (optimal) level of collision resistance, we do not require collision resistance of up to $2^{N/2}$ for a total state width of N bits. Hence, while double-block-length constructions introduce some form of mixing between the parallel evaluations, we omit these and argue that while losing collision resistance, the security is otherwise (e.g., key-recovery attacks) not affected.

More formally, let N be the total state size in bits (e.g., $N = 128$ for PELICAN with the AES), and let the total state size be split into t equally large parts of $N/t = n \in \mathbb{N}$ bits. Then, we use n-bit permutations in the chaining phase, each ensuring that any differential characteristic has a probability of 2^{-n}.

Let us denote the full N-bit state after the first block cipher call by v. This state is split into t equally sized parts $v_i \in (\mathbb{F}_2)^n$, where $i \in [1, t]$, i.e., $v = v_1 \,||\, v_2 \,||\, \cdots \,||\, v_t$. Let us further denote the full N-bit state after the chaining phase by w. Again, we split it into t equally sized parts and obtain $w = w_1 \,||\, w_2 \,||\, \cdots \,||\, w_t$. Next, we split the message m into n-bit parts and obtain $m = m_1 \,||\, m_2 \,||\, \cdots \,||\, m_u$. Now, with our modified chaining phase, we define

$$w_i = \mathcal{P}(\mathcal{P}(\cdots \mathcal{P}(\mathcal{P}(v_i \oplus m_1) \oplus m_2) \cdots) \oplus m_{u-1}),$$

where $i \in \{1, 2, \ldots, t\}$ and $\mathcal{P} : \mathbb{F}_2^n \to \mathbb{F}_2^n$ is a permutation. Finally, the N-bit state w after this step is again feeded to the last block cipher call. Note that this class of chaining phases contains the PELICAN chaining phase for $t = 1$. Hence, we will focus on constructions where $t > 1$.

Rate Reduction. A disadvantage of this method is the reduced rate for $t > 1$. Indeed, every bit of the message has to be processed t times, which is not the case when using PELICAN. This naturally increases the latency of the resulting construction. However, as we will show in our theoretical evaluation, small-state permutations tend to need fewer rounds and also less area.

Omitting the Full Block Cipher Evaluations. A similar approach can also be applied to the two block cipher evaluations. Indeed, since in our scenario we consider a device which is only used a few times before rekeying, we assume that the preparation of the initial state is part of this rekeying process. Hence, the first N-bit state consists of secret bits and is reinitialized when the number of authentications reaches its limit. Since this is similar to a keyed block cipher evaluation[3] of a known state from a security point of view, we can omit the first block cipher evaluation entirely and directly start with the chaining phase.

Moreover, similar to the chaining phase, we can split the N-bit state w after this step into t equally sized n-bit parts and use t different n-bit block cipher calls in parallel. It is crucial that each of these block cipher evaluations provides a computational security of N bits. In other words, it should take no less than 2^N local operations to "decrypt" an n-bit tag part and obtain the n-bit part w_i of the state w.[4] For example, when setting $N = 128$, $t = 2$, and $n = 64$, one may use SKINNY-128-64 with a block size of 64 bits and a key size of 128 bits.

3.1 Detailed Specification

In total, each n-bit part T_i of the tag is computed by

$$T_i = \mathcal{E}_{k^{(i)}}(\mathcal{P}((\cdots \mathcal{P}(\mathcal{P}(s_i \oplus m_1) \oplus m_2) \cdots) \oplus m_{u-1}) \oplus m_u),$$

where $k^{(i)}$ is the N-bit key used for the i-th block cipher call. We recommend all keys $k^{(1)}, k^{(2)}, \ldots, k^{(t)}$ to be pairwise distinct (see Section 4.2). A detailed graphical overview of the construction is shown in Fig. 3. Our MAC does not require any nonces, but we assume that the initial state is freshly initialized after a predefined fixed amount of authenticated blocks.

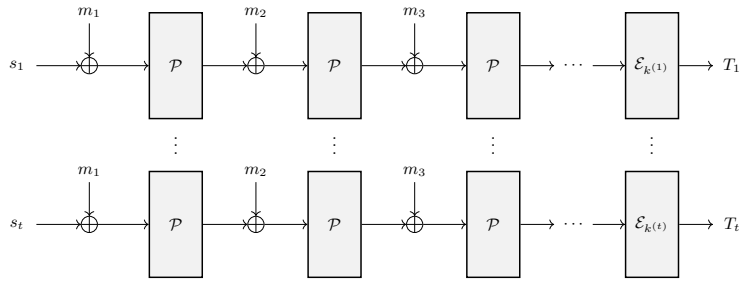

Fig. 3: The LDMAC construction.

[3] In our security analysis, we assume that the initial states are independent of the keys used for the final block cipher calls.

[4] Note that the attacker cannot easily obtain the state before the block cipher call, and hence building the full code book is not possible in a straightforward way.

Minimum Message Length and Padding Rule. We require a minimum message length of one block (i.e., n bits). Regarding the padding, we specify two versions, namely one without a padding rule (and only accepting messages whose sizes are multiples of n) and one with a simple padding rule. First, when using no padding, `LDMAC` only accepts zn-bit messages, where $z \in \mathbb{N}_{>0}$. This is the most efficient version in a scenario where only messages with complete message blocks are used. For example, it makes sense in protocols which only need to authenticate fixed-sized messages. On the other hand, for messages of arbitrary size, we suggest a simple one-zero padding rule. In particular, we first add `0b1` to the message (i.e., a single bit equal to 1), and then as many zero bits as needed to fill the last block.

For simplicity and ease of comparison, in this paper we only focus on message lengths which are multiples of the block size. Indeed, such a limitation makes sense in many real-world scenarios where only fixed-sized messages are authenticated, and we therefore omit any padding. However, also with a padding rule our MAC construction has an advantage compared to competitors, because on average fewer bits have to be added due to the smaller block size.

Key Sizes and an Alternative Version. Given our construction, both the initial state and the keys of the final block cipher calls are secret. In total, this results in $2N$ bits of secret material (e.g., 256 bits for a 128-bit security level). In most practical implementations, this will not be an issue.

However, we conjecture that the security does not decrease when fixing the block cipher keys and reusing them even after having exhausted the allowed number of authentications. In other words, after around $2^{n/2}$ authentications, we could potentially only change the initial N-bit state and keep the same block cipher keys. With this approach, only N secret bits would need to be changed regularly for an N-bit security level.

Reference Implementation and Test Vectors. Together with the theoretical specification, we also provide a reference implementation of `LDMAC` as supplementary material. To instantiate the MAC function, we used the 28-round GIFT-128-64 block cipher for the final block cipher evaluations, and the 16-round permutation (with round keys set to 0) for the permutation calls in the chaining phase (i.e., we set $N = 128$ and $t = 2$).[5] For GIFT-128-64, we used the reference implementation provided online and also used in [1].[6] In our implementation, both the internal permutation and final block cipher can easily be replaced by any other 64-bit primitive.

An overview of the files provided as supplementary material can be found in Supplementary Material A. Test vectors are given in Supplementary Material B.

[5] Note that 14 rounds of GIFT-128-64 are sufficient to meet the required security level [5], but the reference implementation of GIFT-128-64 encrypts in blocks of 4 rounds and hence we chose to increase the number of rounds to 16 for our reference implementation.

[6] https://github.com/aadomn/gift

4 Security of LDMAC

In this section, we evaluate the security of the resulting construction, in particular when compared to the original Pelican MAC function. We assume that the first N-bit state is pseudorandomly generated and unknown to an attacker, and that all of its n-bit parts are pairwise distinct (note that this is a very plausible assumption, especially for reasonably sized n).

We further emphasize that we do not focus on PRF security. As we will also make clear in our analysis below, our construction is always vulnerable to collisions in the smaller inner states. Since these states have a size of only $n < N$ bits, and since we do not introduce any further mixing between these states, we cannot reach an N-bit PRF security. Indeed, the probability of collisions P_{coll} in all of the t branches is bounded by

$$P_{\text{coll}} \leq \left(1 - e^{\frac{-q(q-1)}{2^{n+1}}}\right)^t$$

after q queries. Note that we count collisions whether they occur or not, independently of the number of authenticated blocks.

Hence, instead we focus on the MAC security of our mode. In particular, we will focus on the difficulty of key-recovery and forgery attacks which do not exploit the collision probability in the inner states. For these attacks, we argue that the mode provides a MAC security level of N bits (i.e., a security level equal to the total state size).

4.1 Security Arguments

Compared to Pelican, one may ask if the parallel version of it introduces otherwise nonexistent weaknesses. To answer this question, we first show that computing multiple secure MAC functions in parallel has no negative effect, i.e., it does not *decrease* the security when compared to a MAC computation. Then, we show that key-recovery attacks against LDMAC can be converted into key-recovery attacks against the set of t underlying block ciphers, and that forgery attacks against LDMAC not involving internal collisions can be converted into ciphertext recovery attacks against the set of t underlying block ciphers. Our proofs follow the strategy used for the proofs provided in [15], but are adapted to our specific construction, in particular to the chaining phase using t parallel evaluations. We also assume that all instances of the underlying block cipher are equally secure.[7]

Computing Multiple MAC Instances in Parallel. Our mode is similar to a construction consisting of multiple MAC computations. Indeed, consider

$$F(m, k^{(1)}, k^{(2)}, \ldots, k^{(t)}) = f^{(1)}(m, k^{(1)}) \mid\mid f^{(2)}(m, k^{(2)}) \mid\mid \cdots \mid\mid f^{(t)}(m, k^{(t)}),$$

[7] We emphasize that this is a reasonable requirement assuming that no weak-key instances exist.

where $f^{(i)}(m, k^{(i)}) \in \mathbb{F}_2^n$. We argue that the security level of F is not lower than the security level of any of the $f^{(i)}$. Indeed, this means that any attack against F can be transformed into an attack against any of the $f^{(i)}$. Focusing on an attack using ℓ chosen messages and breaking the unforgeability of F, the transformation works as follows.

1. Let $f^{(j)}$ denote the attacked instance.
2. Locally construct $t-1$ instances $f^{(1)}, \ldots, f^{(j-1)}, f^{(j+1)}, \ldots, f^{(t)}$.
3. Forge a tag $T^\star$ for the resulting MAC function F using ℓ oracle queries for $f^{(j)}$ and $\ell(t-1)$ local computations for $f^{(i)}$, where $i \in \{1, \ldots, t\} \setminus \{j\}$.
4. A forged tag for $f^{(j)}$ is part of $T^\star$.

Security of LDMAC Against Key-Recovery Attacks. Every key-recovery attack against LDMAC using ℓ messages and the corresponding tag values can be converted into key-recovery attacks against the t underlying block ciphers, requiring ℓ chosen plaintexts each. Indeed, let A be a key-recovery attack against LDMAC requiring ℓ messages and the corresponding tag values $(m^{(1)}, T^{(1)})$, $(m^{(2)}, T^{(2)})$, …, $(m^{(t)}, T^{(t)})$, and yielding the keys $k^{(1)}, k^{(2)}, \ldots, k^{(t)}$. The attack against the block ciphers proceeds as follows, where $i \in [1, t]$ and $j \in [1, \ell]$ in all steps.

1. Sample random initial states s_i.[8]
2. Compute $w_i^{(j)}$ locally by using $m^{(j)}$ and the unkeyed permutation $\mathcal{P}$.
3. Request $T_i^{(j)} = \mathcal{E}_{k^{(i)}}(w_i^{(j)})$, where $w_i^{(j)}$ are the chosen plaintexts.
4. Input the tag values $T^{(j)} = T_1^{(j)} \,||\, T_2^{(j)} \,||\, \cdots \,||\, T_t^{(j)}$ to A and obtain the secret key(s).

Security of LDMAC Against Forgery Attacks. Every forgery attack against LDMAC not involving internal collisions and requiring ℓ chosen messages can be converted into ciphertext guessing attacks with probability 1 against the underlying block ciphers, requiring ℓ chosen plaintexts each. Indeed, let B be a forgery attack against LDMAC not involving internal collisions, requiring ℓ messages and the corresponding tag values $(m^{(1)}, T^{(1)})$, $(m^{(2)}, T^{(2)})$, …, $(m^{(t)}, T^{(t)})$ and yielding the correct tag $T^\star$ for the message $m^\star$. The ciphertext-recovery attacks against the block ciphers proceeds as follows, where $i \in [1, t]$ and $j \in [1, \ell]$ in all steps.

1. Sample random initial states s_i.
2. Compute $w_i^{(j)}$ locally by using $m^{(j)}$ and the unkeyed permutation $\mathcal{P}$.
3. Request $T_i^{(j)} = \mathcal{E}_{k^{(i)}}(w_i^{(j)})$, where $w_i^{(j)}$ are the chosen plaintexts.
4. Input $(m^{(j)}, T^{(j)})$ to B and obtain $T^\star$.
5. Compute $w_i^\star$ locally by using $m^\star$ and the unkeyed permutation $\mathcal{P}$.

[8] Note that sampling random initial states is sufficient, since this will represent a valid instantiation of LDMAC.

6. If there is a j for which $w_i^{(j)} = w_i^\star$, there was an internal collision, which is a contradiction to the assumption on B. If no such collision exists, $T_i^\star$ is the ciphertext of $w_i^\star$ when using the key $k^{(i)}$, i.e., $T_i^\star = \mathcal{E}_{k^{(i)}}(w_i^\star)$.

Note that the above result is not applicable in the case of internal collisions. However, as further described later, we limit the data complexity such that with high probability no collisions take place. Moreover, if B is a universal forgery attack, then any message can be chosen for the attack, and hence the ciphertext recovery of any plaintext is possible by first choosing $w_a^{(j)}$ for a fixed a, computing backwards, and generating the last blocks of the messages accordingly. In this case, a determines the block cipher instance which is attacked.

4.2 Attacks against LDMAC

We refer to Supplementary Material C for an evaluation of concrete attack vectors on LDMAC, including differential characteristics, key-recovery attacks, attacks based on the generation of code books, collisions in the chaining phase, and generic guessing attacks. In all these approaches, we assume that the keys used for the final block cipher calls are not the same. For example, for an N-bit master key $k = k_1 \,||\, k_2 \,||\, \cdots \,||\, k_t$, where $k_i \in \mathbb{F}_2^n$, we suggest that the t final N-bit keys $k^{(1)}, k^{(2)}, \ldots, k^{(t)}$ fulfill $k^{(i)} = \mathrm{rot}_{(i-1)n}(k)$, where $\mathrm{rot}_s(\cdot)$ denotes the rotation to the left by s bits. Indeed, if the keys for the last block cipher evaluations are the same, the n-bit tag outputs have to be different since the permutation inputs are different. This would slightly reduce the entropy of the scheme.

4.3 Summary and Security Claims

We provide our security claims in Table 1. We recommend using a limit of $\ll \frac{2^{n/2}}{t}$ for the number of authenticated message blocks.

Attack	Success Probability	Time	Data
Key recovery	≈ 1	2^N	2
Tag forgery	> 0.3	$\frac{2^{n/2}}{t}$	$\frac{2^{n/2}}{t}$
Tag guessing	2^{-N}	1	0

Table 1: The security claims for our construction, where N is the full state size (and key size) and n is the state size of each of the t smaller evaluations.

5 Performance Evaluation

We now compare the performance of LDMAC with various other primitives by using detailed estimates and concrete hardware implementation numbers. We

emphasize that limiting the data complexity is beneficial in our setting, but other constructions in our comparison provide a higher security in that regard.

5.1 Implementation Details

Apart from the specification of the design, various additional properties are needed in a practical (hardware) implementation. The following properties are considered both in the theoretical estimation in Section 5.2 and in our concrete implementation in Section 5.3.

Registers for Received Message Parts. In the cell library we consider, we use 5.33 GE for each of the registers, which results in significant areas even when only taking into account the state size. It is therefore important to not increase the state or the number of temporary registers unnecessarily.

Therefore, we consider a model in which each of the message parts is thrown away after being processed. Given the representation in Fig. 3, we suggest to first finish the computation of each message block before moving to the next block. Formally, this means first computing $\mathcal{P}(s_1 \oplus m_1), \mathcal{P}(s_2 \oplus m_1), \ldots, \mathcal{P}(s_t \oplus m_1)$, and only then beginning to process m_2. The advantage of this approach is that all computations involving m_i can be finalized before receiving m_{i+1}, and therefore m_i can be discarded after each of these steps.

In this case, we add the received message part directly to our state registers and hence avoid temporarily storing it somewhere else. If this method is not applicable for some hardware or implementation reason, we note that the disadvantage with respect to e.g. CBC-MAC is still smaller. Indeed, larger message parts would have to be stored temporarily for constructions with a larger rate.

Storing the Initial State. Each MAC computation of our scheme starts from the same secret initial state. However, after computing a single MAC and using only N registers, the initial state is overwritten and thus a new MAC computation cannot start from it any more. Further, resuming from the "new" initial state would immediately result in attacks, since the observed tag reveals the new starting state and makes forgeries easily possible.

We therefore have to either store the initial state somewhere, or compute a new secret and pseudo-random state after every MAC computation. We take the first approach and note that storing this initial state does not need any additional registers. Indeed, it can be stored in slower memory on the device, since reading from it is sufficient for our design and we never need to write to this memory during an authentication.

5.2 Estimated Comparison with Various Primitives

We use the estimations given in Table 4 in Supplementary Material D, which are taken from the UMCL18G212T3 library [29]. We consider both round-based implementations (i.e., implementations where a full round is implemented in

hardware) and partially round-based implementations such as versions where the S-boxes are computed separately. To illustrate how we estimate the gate area, we provide an example using the KECCAK-f [8] permutation in Supplementary Material D.1. There we also show how we estimate the latency and the final performance numbers for a specific LDMAC instance.

We focus on the case of a 128-bit key and an initial (secret) state of 128 bits. Further, we use $t = 2$, which implies $n = 64$. We therefore use two 64-bit block cipher calls. To give concrete numbers, we chose various 64-bit block ciphers such as SKINNY-128-64 [7], a 64-bit block cipher using a 128-bit key. For the permutation in the chaining phase, we need a sufficiently low differential characteristic probability. In the case of SKINNY-128-64, this means using 8 rounds [7, Table 5]. Hence, for example, to authenticate a 128-bit message, we need $(2-1) \cdot 2 \cdot 8 = 16$ unkeyed SKINNY-128-64 rounds[9] and then two full 36-round evaluations of SKINNY-128-64.

We further focus on PELICAN since it is similar to our mode, and on CBC-MAC and OKS since they are well-known and can easily be generalized to be used together with different internal building blocks. In general, we focus on constructions where this generalization is easily possible – i.e., we focus more on a mode level rather than on specific instances.

The comparison of LDMAC with other approaches in the literature is shown in Table 2, where "OKS" denotes the outer-keyed sponge construction [2]. In this table (and the following tables), we count the cycles c of the resulting construction, each of which has a logic circuit depth d. In the last column, we multiply the area a by the number of cycles c to give an idea of the final efficiency (lower is better). Further, we denote the different instances as Instance-κ-n, where κ denotes the key size and n denotes the block size.

Note that we do not include the depth in the efficiency value. First, the critical path is usually shorter than what the clock frequency allows for power reasons. Therefore, the number of cycles is the dominating property. Secondly, the tendency in low-area constructions goes to round-based (or even smaller) implementations, where naturally the resulting depth is less crucial.

A graphical overview of the area-latency tradeoff when comparing LDMAC to other constructions is shown in Fig. 4 for short 128-bit messages and in Fig. 6 in Supplementary Material E for 512-bit messages. In this graph (and similar graphs in the paper), we connect concrete instances with lines to better illustrate the tradeoff graph for a specific combination.

Regarding the round numbers against differential attacks, we use 20 rounds for SIMON-128-64 [6,22] and 14 rounds for GIFT-128-64 [5]. We apply the same approach to the larger 128-bit versions of the primitives, and hence we always use a round number which provides security specifically against differential attacks.

Further, since we focus on small area requirements, we do not show instantiations which may result in a very low latency, but need significantly more area.

[9] Note that $u - 1$ unkeyed permutation calls are needed in each of the t evaluations when authenticating a u-block message.

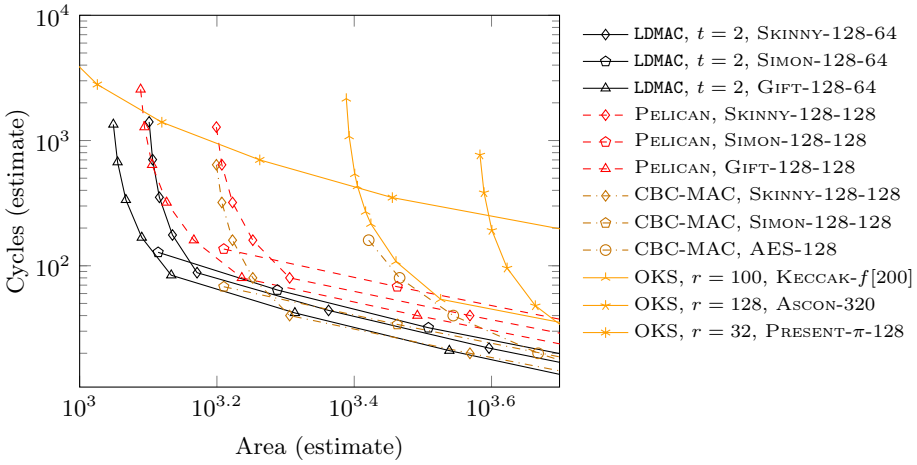

Fig. 4: Comparison of LDMAC with various other constructions when authenticating 128-bit messages.

For the sake of this comparison, we therefore ignore any instantations with an estimated area requirement of 10^4 or more.

Results of the Comparison. We see that LDMAC is competitive in terms of hardware area and latency. In Table 2, we have highlighted the lowest area number and also the best efficiency indicator. Both are achieved by LDMAC constructions using Gift-128-64 and Skinny-128-64, respectively.

In particular, we highlight the following observations. LDMAC offers the lowest area (1120 GE with Gift-128-64), and it also achieves the best efficiency value (200, also with Gift-128-64). Further, our design offers the lowest number of cycles when limiting the area requirements to various fixed values. This advantage is visible for various area limits, particularly when fosusing on low-area designs. A more detailed comparison of this behavior is shown in Fig. 5. In this graph, the black line illustrates the "best" LDMAC instance in terms of the lowest number of cycles up to a specific area limit, and the gray line illustrates the best such instance from all other instances in our comparison.

Hence, comparing on a mode level, we can see that LDMAC achieves its goal of reducing the total amount of area while at the same time offering a similar latency. Moreover, in the case of LDMAC with Gift-128-64 and Pelican with Gift-128-128, we can even observe both an area and a latency advantage when considering the low-area versions of these instances. These advantages are more clearly visible in Fig. 7, where we limit both the GE and the amount of cycles to 1500. In this graph, we have highlighted the points where LDMAC is more efficient than a comparable Pelican instance instantiated with the same primitive.

Algorithm	Config	Area a (GE)	Latency (512-bit message)		$\frac{a \times c}{10^3}$
			Cycles c	Depth d	
LDMAC, $t = 2$ SKINNY-128-64	1 S-box	1263	2944	7	3719
	2 S-boxes	1278	1472	7	1882
	16 S-boxes	1483	184	7	273
	2 rounds	2305	92	14	213
LDMAC, $t = 2$ SIMON-128-64	1 round	1302	368	4	480
	2 rounds	1942	184	8	358
	4 rounds	3222	92	16	297
LDMAC, $t = 2$ GIFT-128-64	1 S-box	**1120**	4032	8	4516
	8 S-boxes	1232	504	8	621
	16 S-boxes	1360	252	8	343
	2 rounds	2059	126	16	260
	7 rounds	5552	36	56	**200**
PELICAN SKINNY-128-128	1 S-box	1583	2000	10	3166
	4 S-boxes	1671	500	10	836
	8 S-boxes	1788	250	10	447
	16 S-boxes	2023	125	10	253
	2 rounds	3704	63	20	234
PELICAN SIMON-128-128	1 round	1622	250	4	406
	2 rounds	2902	125	8	363
	4 rounds	5462	63	16	345
PELICAN GIFT-128-128	1 S-box	1227	5056	8	6204
	8 S-boxes	1339	632	8	847
	16 S-boxes	1467	316	8	464
	32 S-boxes	1723	158	8	273
	2 rounds	3105	79	16	246
TULP SKINNY-128-64	1 S-box	1604	4096	7	6570
	2 S-boxes	1619	2048	7	3316
	16 S-boxes	1824	256	7	467
TULP SIMON-128-64	1 round	1557	456	4	710
	2 rounds	2112	228	8	482
	4 rounds	3221	114	16	368
TULP GIFT-128-64	1 S-box	1461	4928	8	7200
	8 S-boxes	1573	616	8	969
	16 S-boxes	1701	308	8	524
OKS, $c = 100$, $r = 100$ KECCAK-f[200]	1 S-box	2446	5040	6	12328
	40 S-boxes	3357	126	6	423
	2 rounds	6446	63	12	407
OKS, $c = 192$, $r = 128$ ASCON-320	1 S-box	3830	1920	15	7354
	8 S-boxes	4194	240	15	1007
	64 S-boxes	7109	30	15	214
OKS, $c = 96$, $r = 32$ PRESENT-π-128	32 S-boxes	2855	1190	8	3398
	2 rounds	5624	595	16	3347
CBC-MAC SKINNY-128-128	1 S-box	1583	2560	10	4053
	16 S-boxes	2023	160	10	324
	2 rounds	3704	80	20	297
CBC-MAC SIMON-128-128	1 round	1622	272	4	442
	4 rounds	5462	68	16	372
CBC-MAC AES-128	1 S-box	2637	640	41	1688
	16 S-boxes	6960	40	41	279

Table 2: Estimates for the area and latency of LDMAC and other modes. The last column gives an idea of the total efficiency including area and latency (lower is better). We also highlight the best numbers in terms of area and total efficiency. The different instances are denoted as Instance-κ-n, where κ denotes the key size and n denotes the block size.

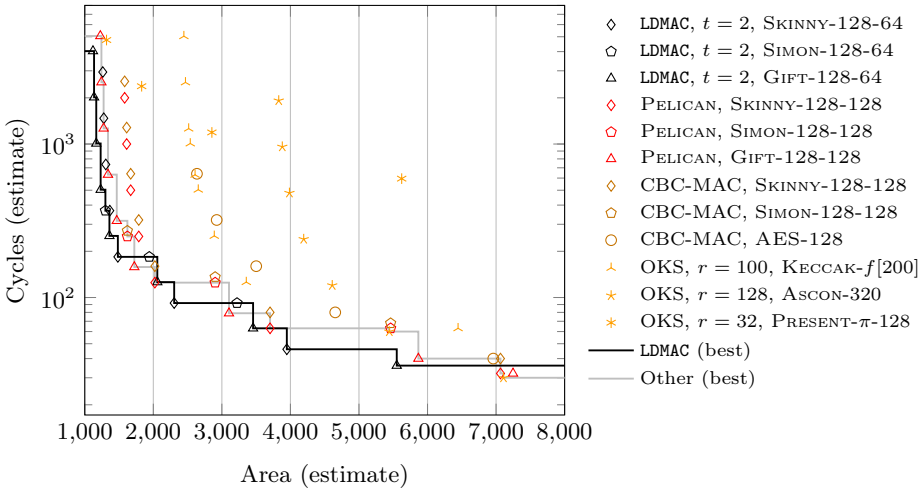

Fig. 5: Comparison of LDMAC with other constructions at various fixed points for the area when authenticating 512-bit messages. The black line illustrates the "best" LDMAC instance in terms of the lowest number of cycles up to a specific area limit, and the gray line illustrates the best such instance from all other instances in our comparison.

5.3 Hardware Implementation Results

Besides our theoretical estimation, we provide results for proof-of-concept ASIC implementations of the proposed LDMAC and Pelican MAC functions. Both implementations use Gift-128-64 and Gift-128-128, respectively, for keyed and unkeyed permutations. Specifically, the LDMAC implementation uses 28 rounds for cipher evaluation and 14 rounds for chaining phase. The Pelican MAC implementation uses 40 rounds for cipher evaluation and 26 rounds for chaining phase. Both MAC implementations follow a round-based approach and process 512-bit long message inputs. The proposed LDMAC and Pelican MAC implementations are synthesized for ASIC using a 65nm standard cell library. Area and performance results of ASIC implementations of the LDMAC and Pelican MAC functions are shown in Table 3. The LDMAC implementation uses around 13% less area compared to the Pelican MAC implementation.

5.4 Additional Considerations

LDMAC becomes increasingly efficient with longer messages, since the full block cipher calls at the end are only ever called t times, independently of the length of the message. The latency differences become more apparent when increasing the message size, since the effect of the reduced-round permutation calls in LDMAC and Pelican is then more obvious.

Design	Latency	Cell count	NAND2-based area
LDMAC	271	944	4.18
PELICAN	167	1218	4.79

Table 3: Area and performance results (ASIC) for a 512-bit message.

Further, LDMAC is sufficiently generic to be used together with smaller primitives. For example, in order to still allow around 2^{16} (block) authentications[10], one may want to use 32-bit permutations. We illustrate this idea briefly in Supplementary Material E.1, where we conclude that the area can further be decreased, albeit at the cost of an increased latency.

Finally, in order to highlight the differences between our construction and other modes using small permutations, let us have a look at the CBC-MAC mode using SKINNY-128-64. While a security against key-recovery attacks of 128 bits may still be provided, guessing the final tag is possible with a probability of 2^{-64}, while the probability for guessing the correct tag in LDMAC (even with intermediate guesses) is $2^{-128} \ll 2^{-64}$. The same can be said about sponge functions, where (depending on the specifics) guessing the full 64-bit intermediate state right before tag squeezing is sufficient in order to also derive the tag.

6 Conclusion

In this paper, we propose a MAC mode which allows the designer to reduce the size of internal permutations, while still providing a security level equal to the total state size w.r.t. attacks which do not focus on internal collisions. This is beneficial in settings where the needed number of authenticated blocks is lower than in more traditional scenarios.

Moreover, our MAC function motivates an new topic for future work, namely the design and analysis of permutations and block ciphers with very small block sizes. Indeed, this often immediately leads to attacks in all other modes we are aware of, but it may result in further performance advantages (in particular even lower area requirements) when used together with our proposal.

[10] The probability of collisions is already significant after 2^{16} block authentications, and hence a lot less (e.g., 2^{14}) should be the limit in practice.

References

1. Adomnicai, A., Najm, Z., Peyrin, T.: Fixslicing: A new GIFT representation fast constant-time implementations of GIFT and GIFT-COFB on ARM cortex-m. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(3), 402–427 (2020)
2. Andreeva, E., Daemen, J., Mennink, B., Assche, G.V.: Security of keyed sponge constructions using a modular proof approach. In: FSE. LNCS, vol. 9054, pp. 364–384. Springer (2015)
3. Ankele, R., Böhl, F., Friedberger, S.: Mergemac: A MAC for authentication with strict time constraints and limited bandwidth. In: ACNS. LNCS, vol. 10892, pp. 381–399. Springer (2018)
4. Aumasson, J., Bernstein, D.J.: Siphash: A fast short-input PRF. In: INDOCRYPT. LNCS, vol. 7668, pp. 489–508. Springer (2012)
5. Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT: A small present - towards reaching the limit of lightweight encryption. In: CHES. LNCS, vol. 10529, pp. 321–345. Springer (2017)
6. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK lightweight block ciphers. In: DAC. pp. 175:1–175:6. ACM (2015)
7. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY family of block ciphers and its low-latency variant MANTIS. In: CRYPTO (2). LNCS, vol. 9815, pp. 123–153. Springer (2016)
8. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Keccak. In: EUROCRYPT. LNCS, vol. 7881, pp. 313–314. Springer (2013)
9. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: CHES. LNCS, vol. 4727, pp. 450–466. Springer (2007)
10. Bogdanov, A., Mendel, F., Regazzoni, F., Rijmen, V., Tischhauser, E.: ALE: aes-based lightweight authenticated encryption. In: FSE. LNCS, vol. 8424, pp. 447–466. Springer (2013)
11. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçin, T.: PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In: ASIACRYPT. LNCS, vol. 7658, pp. 208–225. Springer (2012)
12. CAESAR Competition: Caesar: Competition for authenticated encryption: Security, applicability, and robustness (2018), http://competitions.cr.yp.to/caesar.html
13. Cannière, C.D., Dunkelman, O., Knezevic, M.: KATAN and KTANTAN - A family of small and efficient hardware-oriented block ciphers. In: CHES. LNCS, vol. 5747, pp. 272–288. Springer (2009)
14. Daemen, J., Rijmen, V.: The block cipher rijndael. In: CARDIS. LNCS, vol. 1820, pp. 277–284. Springer (1998)
15. Daemen, J., Rijmen, V.: A new MAC construction ALRED and a specific instance ALPHA-MAC. In: FSE. LNCS, vol. 3557, pp. 1–17. Springer (2005)
16. Daemen, J., Rijmen, V.: The pelican MAC function. IACR Cryptology ePrint Archive **2005**, 88 (2005)
17. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2. Submission to NIST: https://ascon.iaik.tugraz.at/files/asconv12-nist.pdf (2019), http://ascon.iaik.tugraz.at

18. Gong, Z., Hartel, P.H., Nikova, S., Tang, S., Zhu, B.: Tulp: A family of lightweight message authentication codes for body sensor networks. J. Comput. Sci. Technol. **29**(1), 53–68 (2014)
19. Hirose, S.: Provably secure double-block-length hash functions in a black-box model. In: ICISC. LNCS, vol. 3506, pp. 330–342. Springer (2004)
20. Hirose, S.: Some plausible constructions of double-block-length hash functions. In: FSE. LNCS, vol. 4047, pp. 210–225. Springer (2006)
21. Kilian, J., Rogaway, P.: How to protect DES against exhaustive key search. In: CRYPTO. LNCS, vol. 1109, pp. 252–267. Springer (1996)
22. Liu, Z., Li, Y., Wang, M.: Optimal differential trails in simon-like ciphers. IACR Trans. Symmetric Cryptol. **2017**(1), 358–379 (2017)
23. Luykx, A., Preneel, B., Tischhauser, E., Yasuda, K.: A MAC mode for lightweight block ciphers. In: FSE. LNCS, vol. 9783, pp. 43–59. Springer (2016)
24. Menezes, A., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (1996)
25. Merkle, R.C.: One way hash functions and DES. In: CRYPTO. LNCS, vol. 435, pp. 428–446. Springer (1989)
26. Mouha, N., Mennink, B., Herrewege, A.V., Watanabe, D., Preneel, B., Verbauwhede, I.: Chaskey: An efficient MAC algorithm for 32-bit microcontrollers. In: Selected Areas in Cryptography. LNCS, vol. 8781, pp. 306–323. Springer (2014)
27. Nandi, M.: Towards optimal double-length hash functions. In: INDOCRYPT. LNCS, vol. 3797, pp. 77–89. Springer (2005)
28. National Institute of Standards and Technology: Submission requirements and evaluation criteria for the lightweight cryptography standardization process (2018), https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf
29. Virtual Silicon Inc.: 0.18μm VIP Standard Cell Library Tape Out Ready, Part Number: UMCL18G212T3, Process: UMC Logic 0.18μm Generic II Technology: 0.18μm (2004)
30. Yuan, Z., Wang, W., Jia, K., Xu, G., Wang, X.: New birthday attacks on some macs based on block ciphers. In: CRYPTO. Lecture Notes in Computer Science, vol. 5677, pp. 209–230. Springer (2009)

A Supplementary Material

We provide the following files as supplementary material:

- `code_hw/*`
- `gift/*`
- `ldmac.cpp`

The folder `code_hw` contains the code used for the hardware implementation. The folder `gift` contains the reference implementation of GIFT-128-64, with some additions such as a 64-bit permutation of the primitive with a modified round number and the round keys set to 0. The file `ldmac.cpp` contains the reference implementation of `LDMAC` using GIFT-128-64, where the state N is 128 bits large. We provide all files online.[11]

B Test Vectors

In the following representation, the message and the tag value are denoted by $m^{(\cdot)}$ and $T^{(\cdot)}$, respectively. The initial state is set to `0x12...1234...34` $\in (\mathbb{F}_2)^{128}$ and the key is set to `0xAB...AB` $\in (\mathbb{F}_2)^{128}$ in all tests. Note that, for the sake of simplicity, we did not sample the initial state at random for our test vectors. The resulting test vectors are

$$
\begin{aligned}
m^{(1)} &= \texttt{0x00000000000000000000000000000000},\\
T^{(1)} &= \texttt{0x117c50b5bfb88c2384f1e778f6264d25},\\
m^{(2)} &= \texttt{0x42424242424242424242424242424242},\\
T^{(2)} &= \texttt{0xe8bc31ce41fe58b828322883e174e772},\\
m^{(3)} &= \texttt{0xffffffffffffffffffffffffffffffff},\\
T^{(3)} &= \texttt{0xe7019dd3ded4f28d709f077171431468}.
\end{aligned}
$$

C Attacks on `LDMAC`

Differential Characteristics. The "best" probability for any differential characteristic (from a designer's point of view) an n-bit permutation can achieve is 2^{-n}. However, observe that any difference injected into the message is simultaneously injected into every of the t branches due to the shared absorption phase. Hence, assuming a cryptographically strong permutation $\mathcal{P}$, the differential probability is $2^{-nt} = 2^{-N}$.

[11] https://github.com/mschof/ldmac

Key-Recovery Attacks. For simplicity, we first focus on $t = 1$. Note that the initial state is not known by the attacker in our scenario, hence a trivial approach using a single (message, tag) pair will not work. However, the attacker can use two different (message, tag) pairs and then, for each possible key, compute backwards by using both tags and messages. If the initial state is the same for both computations, the attacker can conclude that the current key candidate is correct with high probability. This approach needs 2^N operations and is essentially equal to exhaustive search.

Let us assume $t > 1$ now. Recall that we require each of the t final n-bit block ciphers to provide a security against key-recovery attacks of N bits. Focusing on one single evaluation, this means that key-recovery attacks have a complexity in $\Theta\left(2^N\right)$. Since all evaluations are independent of each other, and only the added constant part is shared, the complexity of recovering the key is still an element in $\Theta\left(2^N\right)$, or strictly speaking $\Theta\left(2^N/t\right)$ when making t trials in parallel.

Generating a Code Book for the Last Keyed Part. Given n-bit block cipher evaluations in the final step, we can make 2^n queries with altering last message blocks to obtain a lookup table with a size of $2^n \cdot 2n$ bits, containing the translations

$$c \oplus {m_u}^{(i)} \rightarrow o^{(i)},$$

where $i \in [1, 2^n]$. Now, we can modify the second to last message block and request a tag for this message. The output will be an entry $\overline{o^{(i)}}$ of the table, giving us the corresponding $\overline{{m_u}^{(i)}}$. We can now add an arbitrary difference to the last message block and forge the resulting tag offline using our lookup table. The same method can be applied to any of the t evaluations.

This simple approach needs at least $2^n + 1$ queries. However, note that this attack can be parallelized by using the t different evaluations. If the keys are the same for all t evaluations, we can exploit this fact. When applying this method, we only need $\frac{2^n}{t}$ queries in order to build the lookup table, which results in a data limit of $\frac{2^n}{t}$. Hence, reducing n and increasing t significantly may quickly result in data limits which are far too low. Again, we therefore recommend to use different keys for the final block cipher calls, or different permutations of the same key bits.

Note that a tradeoff version of this attack is also possible. In particular, consider using $2^c < 2^n$ data for the attack (and a table of respective size). Now, when following the same approach, the probability for a table entry to exist is $2^{-(n-c)}$, which reduces the probability of the attack while also reducing the needed amount of data. We emphasize that this tradeoff does not extend to all of the t branches. Indeed, while we can produce tables of size 2^c for all branches in parallel, first the final probability is $2^{-t(n-c)}$ for all entries, and secondly we need the single change in the last block to satisfy the needed outputs in all branches, which is again only the case with small probability ($2^{-n(t-1)}$, the same as simply guessing the remaining tag parts). In conclusion, if D denotes

the amount of data used and if P denotes the success probability of the attack, we claim that $\log_2(DP^{-1}) = N$.[12]

Collisions in the Chaining Phase. When using arbitrarily sized messages and n-bit permutations, collisions in the chaining phase are to be expected after around $2^{n/2}$ queries. An attacker can detect a collision when using different messages and observing the same output for two different messages in any of the tag parts.

Let us assume the attacker has found such a collision and knows where this collisions occurred (i.e., where precisely in the chaining phase). Let us further assume $t = 2$ for simplicity. Now, the attacker can modify the block in the first message where the collision took place (which is not necessarily the same in both of the messages) by adding a constant. The attacker now requests the tag for this modified message, knowing that adding the same constant to the same position in the second message will yield the same tag part with a probability of 1. They can now simply guess the second tag part, which leads to a successful forgery with a probability of 2^{-64} when knowing where exactly a collision has occurred.

In general, when knowing where the collision has occurred in t' evaluations, the success probability of the attack by guessing the remaining tag parts is $2^{-(N-(t'n))}$. Knowing where the collision happened (or guessing the position with high probability) is possible when using only a few message blocks. Moreover, a collision can happen in every of the t branches, and this can be evaluated in parallel. This can be (approximately) accounted for by dividing the number of allowed message blocks by t. We therefore further decrease the allowed data limit to $\ll \frac{2^{n/2}}{t}$ authenticated message blocks.

Guessing Attacks. Let us for simplicity assume $t = 1$. The construction is now similar to Pelican. Having a state size of $n = N$ bits, guessing the tag is successful with a probability of 2^{-N}.

Now, let us assume $t > 1$. The state is now $N = nt$ bits large, and each of the t evaluations has an n-bit state. Note that the t evaluations (and in particular the t n-bit initial states) are independent of each other. They only share the chaining phase with the same message blocks. Hence, guessing the output of one of the t evaluations (i.e., one single tag part) does not yield any information about the other $t-1$ evaluations. Therefore, guessing the entire tag is successful with a probability of $2^{-nt} = 2^{-N}$, and is equally efficient as in the classical (one-branch) Pelican construction.

For completeness, we also evaluate the possibility of guessing intermediate states. Assume we guess one of the intermediate states (or even one of the initial states) correctly. This happens with a probability of 2^{-n}, which may be non-negligible for small n. We can now easily adjust adjacent message parts in order to provoke a collision in the chaining phase, and the output of this single evaluation will then be equal to the output of a previously queried message. However,

[12] Note that this claim is reminiscent of claims regarding Even–Mansour constructions.

this also affects all other $t-1$ evaluations. We would therefore need to guess the states of the $t-1$ other evaluations as well, which again results in a success probability of $2^{-nt} = 2^{-N}$.

D Estimation Details

NOT	NAND	AND	NOR	OR	XOR	MUX	D flip-flop
0.67	1	1.33	1	1.33	2.67	2.33	5.33

Table 4: Gate equivalents for various logic gates and registers.

D.1 Concrete Calculation

KECCAK-f Gate Area. KECCAK-f uses five main operations: χ, θ, ι, π, and ρ. The latter two use no gates and can be accomplished with simple wiring. χ is the nonlinear 5-bit S-box operation using one `XOR` gate, one `AND` gate, and one `NOT` gate per state bit (i.e., five `XOR` gates, five `AND` gates, and five `NOT` gates per S-box). θ is the main diffusion operation using two `XOR` gates per state bit. Finally, ι denotes the round constant addition, needing one `XOR` gates for each lane of KECCAK-f. Hence, the estimated GE number for KECCAK-$f[N]$ when focusing on rounds implemented fully in parallel is given by

$$\underbrace{N \cdot 5.33}_{\text{Registers}} + \underbrace{\left(3N + \frac{N}{25}\right) \cdot 2.67}_{\texttt{XOR gates}} + \underbrace{N \cdot (0.67 + 1.33)}_{\texttt{NOT and AND gates}}.$$

Estimations for `LDMAC` Instance. To better illustrate how we arrive at the final numbers, let us consider 128-bit `LDMAC` instantiated with SIMON-128-64, i.e., we have $n = 64$ and $t = 2$. First, we compute the area requirements of SIMON-128-64. Following the method given above, we use three 32-bit adders, one for each branch, resulting in 64 `XOR` gates. We also need 32 `AND` gates for the nonlinar operation, and finally 64 registers. This results in an area (GE) of

$$\underbrace{64 \cdot 5.33}_{\text{Registers}} + \underbrace{96 \cdot 2.67}_{\texttt{XOR gates}} + \underbrace{32 \cdot 1.33}_{\texttt{AND gates}} = 640.$$

We further add 64 registers to meet the total state size of 128 bits, and we also add 64 `XOR` gates for the absorption. Finally, we add 64 multiplexers, and we note that this is done for every `LDMAC` instance in our comparison. We then arrive at

$$\lceil 640 + (64 \cdot 5.33) + (64 \cdot 2.67) + (64 \cdot 2.33) \rceil = 1302$$

GE for a round-based implementation, which can also be found in Table 2.

Regarding the number of cycles, we consider a round-based implementation for the sake of simplicity. Then, we use 20 rounds for SIMON-128-64 [6,22], and we need

$$t \cdot \left(\left\lceil \frac{|m|}{n} \right\rceil - 1 \right) \cdot 20 + (t \cdot 44)$$

rounds in total, where 44 is the number of rounds of the final block cipher calls. For $t = 2$ and $|m| = 512$, this leads to a total number of 368, as can also be found in Table 2.

In this estimation, we ignore the effect of control logic, since it is hard to derive a meaningful number without an actual hardware implementation. We apply the same approach to every other construction, noting that a constant overhead due to the control logic would get added similarly to all constructions.

E Additional Comparisons

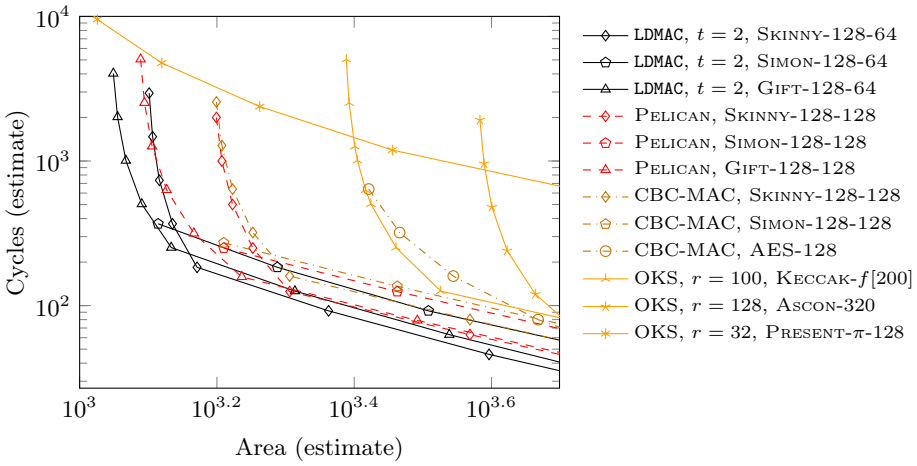

Fig. 6: LDMAC and other constructions when authenticating 512-bit messages.

E.1 Using 32-Bit KATAN32 with LDMAC

KATAN32 [13] provides a security level of 80 bits. This is below the recommended security level by NIST, and also below the security level of constructions in our comparison, but we use it to illustrate how LDMAC could scale to 32-bit primitives.

KATAN32 works by splitting the 32-bit state into two unbalanced parts, and then applying two nonlinear functions in each round. These use 8 XOR gates

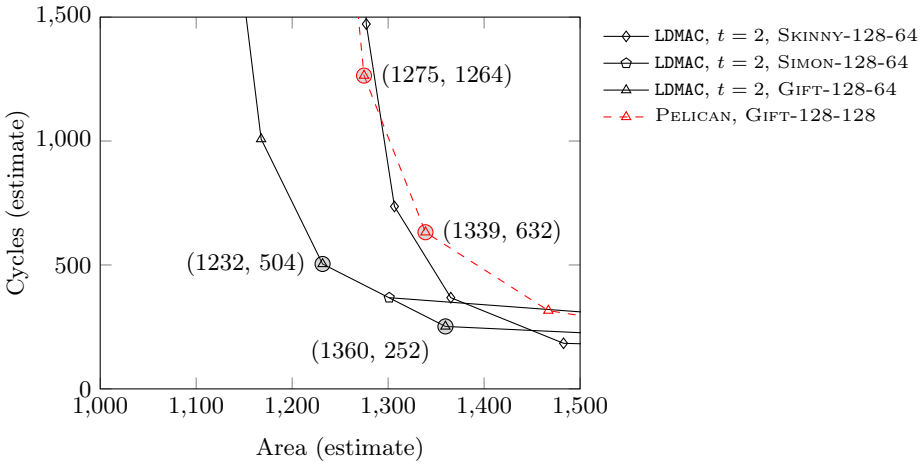

Fig. 7: LDMAC and other constructions when authenticating 512-bit messages and limiting both the area and the number of cycles to 1500.

and 4 AND gates. Following designers' analysis, 126 rounds are needed in order to achieve security against differential attacks. Hence, we use an 126-round unkeyed version of KATAN32 in the absorption phase, and we generate our final tag parts by using 4 calls to the full 254-round block cipher.

The results of this approach are shown in Table 5. We can see that the area can further be decreased, albeit at the cost of an increased latency. For an additional comparison, we also include our mode together with Simon-64-32. However, we strongly emphasize that the security level of this instance is only 64 bits (because Simon-64-32 itself provides only 64 bits of security against key-recovery attacks), and hence we do not encourage to instantiate LDMAC with Simon-64-32.

Algorithm	Config	Area a (GE)	Latency (512-bit message)		$\frac{a \times c}{10^3}$
			Cycles c	Depth d	
`LDMAC`, $t = 2$	1 S-box	**1120**	4032	8	4516
GIFT-128-64	7 rounds	5552	36	56	**200**
`LDMAC` KATAN32, $t = 4$	1 round	869	8576	3	7453
	2 rounds	1067	4288	6	4576
	10 rounds	2645	858	30	2270
	20 rounds	4617	429	60	1981
`LDMAC` SIMON-64-32, $t = 4$	1 round	992	1448	4	1437
	2 rounds	1312	724	8	950
	4 rounds	1952	362	16	707

Table 5: Estimates for the area and latency of `LDMAC` using 32-bit permutations and a 80-bit security level (KATAN32), a 64-bit security level (SIMON-64-32), and the previous instance using SKINNY-128-64.