A new stand-alone MAC construct called SMAC

Dachao Wang[1], Alexander Maximov[2], Patrik Ekdahl[2], and Thomas Johansson[1]

[1] Dept. of Electrical and Information Technology, Lund University, Lund, Sweden
dachao.wang,thomas.johansson@eit.lth.se
[2] Ericsson Research, Lund, Sweden
alexander.maximov,patrik.ekdahl@ericsson.com

Abstract. In this paper, we present a new efficient stand-alone MAC construct based on processing using the FSM part of the stream cipher family SNOW, which in turn uses the AES round function. It offers a combination of very high speed in software and hardware with a truncatable tag. Two concrete versions of SMAC are proposed with different security levels, although other use cases are also possible. For example, SMAC can be combined with an external ciphering engine in AEAD mode. Every design choice is justified and supported by the results of our analysis and simulations. A novelty of the proposal is that it meets future performance requirements but is still not directly vulnerable to attacks using repeated nonce when the tag size is short, as is the case for other very fast MACs (MACs based on polynomial hashing). This can be an important aspect in practical applications.

Keywords: MAC · SNOW · AES

1 Introduction

A Message Authentication Code (MAC) is a standard symmetric primitive for two parties that share a secret key to verify that a received message originates from the sending party and that it was not modified by an attacker when sent on a possibly insecure channel. Traditionally, most existing MACs are built either from a block cipher or from cryptographic hash functions. Common examples of constructs based on block ciphers are CBC-MAC and CMAC [IK03]. A common hash-based construct is HMAC [BCK96]. Another common direction is to use universal hash functions (or equivalently unconditionally secure authentication codes [BJKS94]) as a basis for constructs, resulting in schemes like UMAC [BHK$^+$99], Poly1305-AES [Ber05] and GMAC [Dwo07].

The speed of our communication systems is increasing rapidly. 6G (sixth-generation mobile) is the successor to the current 5G cellular technology, expected to be in use beyond 2030. 6G networks will use higher frequencies than 5G networks and provide substantially higher capacity together with a much lower latency. 5G is delivering up to 20 Gbps peak data rates and more than 100 Mbps average data rates. 6G is expected to be a factor up to 100 faster than 5G [Eri24]. The protection of data in 5G and 6G requires very fast cryptographic primitives and the possibility to implement them in a low-latency manner both in software and hardware.

Along with virtualisation trends and faster performance requirements, there is also a need for truncatable secure tags, which is important for many practical use cases. For example, the 4G/5G radio link layers mostly use 32-bit tags, transport and application layers commonly use 64-bit tags, and 80-bit tags are used in media applications [CMM23], etc.

CMAC and HMAC. Well-known constructs such as HMAC and CMAC are both slow in software (see e.g. our own measurements in Table 8). Consider, for example, AES-CMAC [SLI05]. In terms of hardware reuse and sharing components this seems like a good option, but performance wise it is a slow solution as it needs at least 10 sequential calls to AES round function to process one message block with a 128-bit key, and even more rounds with a 256-bit key. AEAD schemes that use AES-CMAC as the MAC engine are considered slow for virtualisation needs. Hash-based constructs appear to be too slow as well.

Polynomial-based MACs. The known fast algorithms are based on polynomial evaluations such as GCM/GMAC, Poly1305, etc. Consider AES-GCM based on GHASH [Dwo07], which can in turn be parallelised for a faster performance. However, GCM produces a 128-bit tag but its security level is upper bounded by $2^{\tau-k-1}$, where τ is the length of the tag in bits, and the combined plaintext and associated data are of maximum length 2^k words. Thus, the security of GCM is only around 96 bits while the tag cannot be truncated [MW16] and should be kept as 128-bit long. Moreover, the GHASH core in hardware can be large in terms of the number of gates.

As we describe later, most polynomial-based MACs seem not suitable to produce short tags as they become directly vulnerable to nonce-misuse types of attacks even if the nonce is respected. A possible solution to make them truncatable is to adopt e.g. Hash-then-Encrypt, such as the CWC [KVW04] mode of operation for block ciphers that relies on CTR mode for encryption with polynomial Carter–Wegman MAC. CWC-AES uses a polynomial evaluation at a secret point, like in GCM, then encrypts that result with AES, the output of which is finally truncated and produced as the MAC tag. The main downside of CWC mode is that it requires a block cipher for the tag encryption, whereas a stream cipher cannot be used for this step. Such a construct may be seen as a too heavy solution to serve as a stand-alone MAC engine.

A general weakness of polynomial-based MACs (including GCM and CWC) is that the algebraic structure opens up for a variety of attacks such as weak keys, summarised in [PC14]. There also exist other MAC schemes that are based on polynomial hashing to go beyond the birthday bound while many of them are also vulnerable to forgery attacks, see [SSW23] for a summary.

It seems that one of the advantages of polynomial-based MACs is the possibility to parallelise the computation and thus achieve a faster speed. However, the Accordion mode discussed by NIST [NIS24b] turns non-parallelisable primitives to be parallelisable, thus that advantage diminishes.

AES-NI[3] based primitives. In order to meet fast performance in virtualisation environments, there have been a number of efficient cryptographic algorithms constructed by making use of the AES round function so that the AES-NI instruction set can be used for speedups[BÖS11]. The first such attempt may have been the stream cipher LEX [Bir07], and another was the AEGIS family [WP14], which offers authenticated encryption. The stream ciphers SNOW-V [EJMY19] and Rocca [SLN+21] are two other examples.

MAC algorithms based on the AES round function were also developed, for example Alpha-MAC [DR05]. The state size of Alpha-MAC is only the block width and further analysis showed that the construct is not sufficiently secure [BDF11]. Another example of using AES-NI is AEZ [HKR15], one of the CAESAR candidates, claimed to be fast and resistant to nonce-misuse attacks. It has however been shown that all 5 versions of AEZ are indeed vulnerable to various collision and key recovery attacks [FLS15,CG16,VV18] through nonce-misuse scenarios.

AEAD. The development of authenticated encryption with associated data (AEAD) schemes provides joint encryption and authentication in a single primitive. This results in a highly efficient primitive. The sponge duplex is a popular construct of AEAD schemes. It can be noted that almost half of the 56 round-1 submissions to the NIST lightweight cryptography standardisation process were based on this construct, and the winner Ascon [DEMS21b] is one of them.

Although AEAD schemes are efficient solutions, some drawbacks can be found. The security can be more difficult to analyse because the adversary has more attack options compared to pure encryption or pure authentication. Take for example Rocca [SLN+21], where encryption and authentication happen in parallel by reusing the same registers of the internal state. The update function of Rocca is based on AES round, and since the AES round is already "heavy", the update function cannot have the circuit depth of more than a single call to the AES round in order to meet high performance.

By design, Rocca exposes information about its internal state in the form of the ciphertext. The final tag can be truncated, and such dual use of the internal state in combination with a short tag is challenging. It is shown in [HII+22] that Rocca is highly vulnerable to a nonce-misuse attack, especially for truncated tags, resulting in a key recovery even when the nonce usage is respected. Furthermore, various forgery attacks were also found in the AEAD candidates in the 3rd round of CAESAR [VV18], which indicates that the design of a MAC scheme can be a challenging task in AEAD constructs.

On the other hand, Ascon-MAC [DEMS21a] adopts lightweight rounds but the number of clocks between two message compressions is 12, which seems sufficient for a good shuffling of the internal state, but due to these 12 clocks the

[3] An AES instruction set is a set of processor instructions specifically designed to perform AES encryption and decryption operations fast. These instructions are found in modern processors and accelerate AES operations significantly compared to pure software implementations. AES-NI was the first such implementation, an extension to the x86 instruction set for processors from Intel and AMD.

software performance is not very high and comparable to CMAC and HMAC. The authors report at minimum 6.2 cycles per byte on Intel Core i5-6300U for long messages which corresponds to around 3.9 Gigabits per second (Gbps).

It is possible to take some of the AEAD schemes as a MAC function for an external cipher but, in reality, such a combination could be a too big solution. For example, adding GMAC to a SNOW-family cipher basically means two ciphering algorithms and the GHASH core in hardware.

Having all these complexities in mind, in this work we would like to focus on an efficient non-polynomial generic stand-alone MAC engine that can be combined with an external block or stream cipher independently.

MAC forgeries. In the context of MAC cryptanalysis, a *forgery attack* is when the attacker obtains the correct MAC value for a chosen message, with some success probability. Another attack is a *universal forgery attack* where the attacker can retrieve the internal state of the MAC engine, or even worse, the complete secret key; in this case, the attacker can create an infinite number of valid messages with corresponding tags in time $O(1)$. A good MAC engine should preferably have the forgery success probability around $2^{-\tau}$, where τ is the size of the tag in bits, and the complexity to obtain a universal forgery to be as difficult as an exhaustive key search. This is not always the case as, for example, the tag size in GCM is 128 bits while only providing 96 bits of security.

Nonce-misuse with short tags. Many MAC algorithms require a nonce to be guaranteed/checked for uniqueness by both the sender and the receiver sides. In case of imperfect nonce-tracking implementations, nonce-misuse attack scenarios become more likely, which affects the robustness. In practice, although it is easy for the sender to ensure the nonce uniqueness, it is much harder for the receiver side to track nonces that have already been used, especially if nonces are taken as a pseudo-random (even non-repetitive) sequence, or when there are multiple receivers of the same message, each of which can be used as a verification oracle.

In a nonce-misuse scenario, universal forgery attacks on GCM [Jou06] and many CAESAR candidates [VV18] only require a few nonce-reuses (merely 2 for GCM and OCB). There is a strong recommendation that a check for the nonce uniqueness is done on both sides [HP08].

Government organisations have demonstrated a particular interest in robustness and highlight misuse-resistant AE (MRAE security) as a high-priority requirement [Cam23]. Citing [Cam23] on GCM, "nonce misuse leads to a catastrophic loss of security". Another such high-priority requirement is protection against release of unverified plaintext. New schemes GLEVIAN and VIGORNIAN addressing these concerns are proposed, but they lead to a degradation in performance.

The desire to have short (truncatable) tags leads us to a new situation when it comes to the nonce-misuse resistance property of a MAC scheme. If a MAC algorithm produces a short tag of size τ bits (e.g., $\tau = 32$), the attacker could simply collect many pairs (Message, Tag) by using the verification oracle at most 2^{τ} times per one valid pair. This is equivalent to a nonce-misuse attack with the sender oracle being virtually available, up to a multiplier 2^τ for the overall attack complexity, which is small for short tags, even though the MAC algorithm requires a unique nonce and it is indeed respected on the sender side.

The problem with weak or non-perfect nonce tracking on the receiver side becomes more severe when a polynomial-based MAC scheme is used to generate a short truncated tag. For example, AES-GCM-SST [CMM23] scheme appears to be vulnerable to the universal forgery attack in case the attacker has access to the verification oracle and makes just $O(2^{32})$ queries to the oracle (for the tag size of 32 bits) [IET24]. As a result, the attacker can create an unlimited number of messages under the same security context (e.g., same key and IV).

A similar issue was identified in the NIST SP 800-38D publication [NIS07], when using GCM-based MAC constructs with truncated tags [MW16]. Nowadays, NIST plans to revise that special publication by removing the support for tags shorter than 96 bits [NIS24a].

Our contribution. In this paper, we present a new efficient stand-alone MAC construct based on processing using the FSM part of the stream cipher family SNOW, which in turn uses the AES round function. It offers a combination of very *high speed in software*, *efficiency in hardware*, *truncatable MAC*, and a decent *robustness in nonce-misuse scenario.*

The design and security analysis consider Maximum Degree Monomial (MDM) tests, Time-Memory Trade-Off (TMTO), Guess-and-Determine (GnD), nonce-misuse, Key/IV/message differential, MAC forgery and cube attacks, and study clustering effects of differential trails as well as the number of active Sboxes for different choices of a particular permutation through CP/SAT/MILP modelling tools. Every design choice is justified and supported by the results of our analysis and simulations.

2 The SMAC construct

We propose a new MAC engine design framework, called SMAC. It is derived from the finite state machine part of the SNOW family of stream ciphers [EJMY19], as depicted in Figure 1.

The SMAC construct has three 128-bit internal state registers $(A1, A2, A3)$ and their values at time t are denoted by $(A1^t, A2^t, A3^t)$. Given a 128-bit wide message word M^t at time t, the internal registers are updated by the compression function $\varPi$ as follows:

$$\begin{aligned}(A1^{t+1}, A2^{t+1}, A3^{t+1}) \leftarrow \varPi(A1^t, A2^t, A3^t, M^t) \ :=& \\ A1^{t+1} &= \sigma(A2^t \oplus A3^t \oplus M^t) \\ A2^{t+1} &= AES_R(A1^t, M^t) \\ A3^{t+1} &= AES_R(A2^t, M^t)\end{aligned}$$

where σ is a fixed 16-byte permutation, and $AES_R(X, K)$ is the AES round function. A general way of usage of the compression function $\varPi$ in the SMAC framework is shown in Figure 2, and consists of three phases:

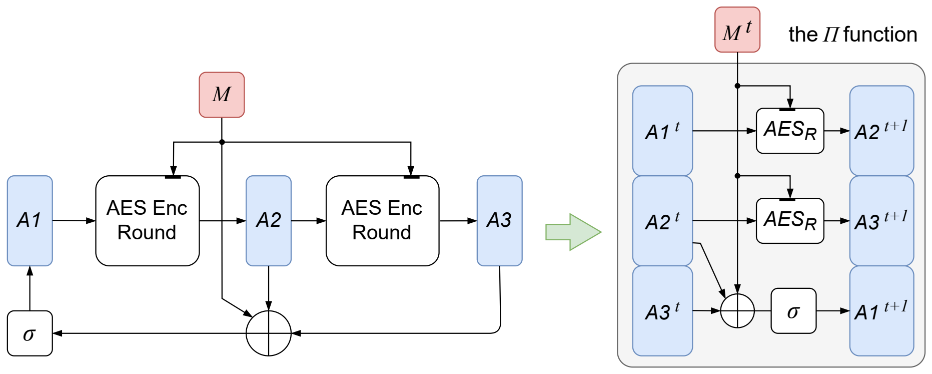

Fig. 1. SMAC compression function Π.

– **Initialisation phase.** In the initialisation phase, the three 128-bit registers $A1, A2, A3$ are loaded with the key material and other possible domain separation parameters, for example a nonce and the MAC tag size. The compression function Π is then clocked d times with a fixed $M = \mathbf{1}^\star$ (to be defined further), in order to bring the initial state to pseudo-random. The internal state after the initialisation phase is denoted by $(A1^t, A2^t, A3^t), t = 0$.
– **Compression phase.** The next n clocks are used to compress the sequence of n 128-bit message blocks $M^0, \ldots, M^{n-1}$, where the last message block should contain the actual length of the message in bits, which would resist attacks based on e.g. insertion and deletion of message words. The internal state after the compression phase is denoted by $(A1^t, A2^t, A3^t), t = n$.
– **Finalisation phase.** Before the MAC tag is produced, the SMAC engine does d dummy calls to the compression function with $M = \mathbf{1}^\star$, similarly to the initialisation phase. The output MAC tag is extracted from the state $(A1^t, A2^t, A3^t), t = n + d$, by simply taking the required number of bits.

2.1 Detailed description

The description is byte oriented and we will denote an array of bytes of length l by $\{\mathbb{N}_8\}^l$, where $\mathbb{N}_k$ denotes the natural numbers representable using k bits. The elements in an array $A \in \{\mathbb{N}_8\}^l$ are referenced by

$$A[0], A[1], \ldots, A[l-1],$$

where $A[0]$ is the first element in the array and $A[l-1]$ is the last. An assignment of an array $A = B$ is done element by element, as is the XOR (also denoted by $\oplus$) of two arrays $C = A \oplus B$, where then $C[0] = A[0] \oplus B[0]$, et cetera. The registers are considered byte arrays $A1, A2, A3 \in \{\mathbb{N}_8\}^{16}$.

The concatenation of two arrays A, B is denoted by $C = A \parallel B$ and the result C will carry the elements of A in its first positions and the elements of B in its last positions.

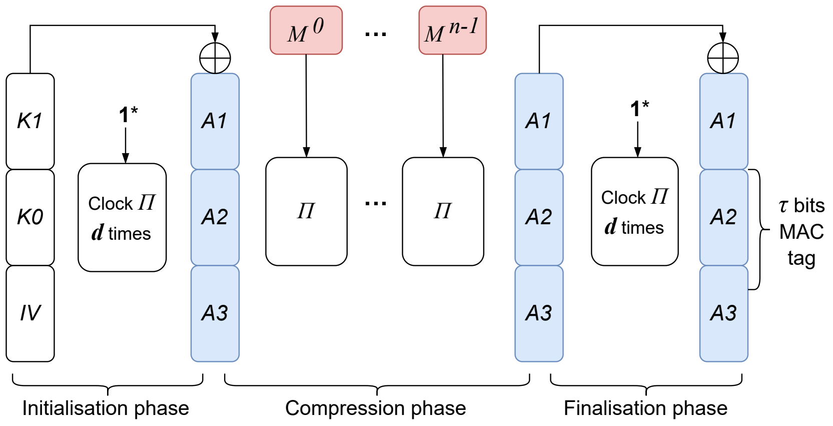

Fig. 2. A general usage model of the SMAC framework.

Let $\sigma(\cdot) : \{\mathbb{N}_8\}^{16} \rightarrow \{\mathbb{N}_8\}^{16}$ denote a byte permutation of an array of length 16. A specific permutation is defined as

$$\sigma = [\pi_0, \pi_1, \ldots, \pi_{15}], \pi_k \in [0 \ldots 15], \pi_i \neq \pi_j \ \forall (i \neq j).$$

This should be interpreted as the element at index π_0 is moved to position 0, the element at index π_1 is moved to position 1, and so on. For example, $\sigma(A)$ will result in the permuted array

$$B = \sigma(A) = \{A[\pi_0], A[\pi_1], \ldots, A[\pi_{15}]\},$$

where $B[0] = A[\pi_0], B[1] = A[\pi_1], \ldots, B[15] = A[\pi_{15}]$.

Furthermore, let

$$\mathbf{1}^{\star} = \{1, 0, 0, \ldots, 0\}$$

denote the array of 16 bytes with a single one in the first position and zeros in the rest. $M = \mathbf{1}^{\star}$ is the fixed constant value fed into Π during initialisation and finalisation phases. A single instance of the AES round function is denoted by

$$AES_R(X, K) : (\{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16}) \rightarrow \{\mathbb{N}_8\}^{16},$$

and defined as $AES_R(X, K) := \texttt{MixColumns}(\texttt{ShiftRows}(\texttt{SubBytes}(X))) \oplus K$. In subsequent sections of this paper, we will also use the notation $L \cdot X = \texttt{MixColumn}(X)$, $\pi \cdot X = \texttt{ShiftRows}(X)$, and $S(X) = \texttt{SubBytes}(X)$, so that the AES round can be rewritten in a shorter form as

$$AES_R(X, K) = L\pi S(X) \oplus K.$$

The mapping between a byte array and the AES state X is done in the usual way as defined in [oST01]. We can now formally define the compression function

Algorithm 1 $\Pi : (\{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16}) \to (\{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16})$

```
1: function $\Pi(A1, A2, A3, M) \to (A1', A2', A3')$
2:     $A1' = \sigma(A2 \oplus A3 \oplus M)$
3:     $A2' = AES_R(A1, M)$
4:     $A3' = AES_R(A2, M)$
5: end function
```

Algorithm 2 `InitFinal` :
$(\{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16}) \to (\{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16})$

```
1: function InitFinal$(A1, A2, A3) \to (A1', A2', A3')$
2:     $(X1, X2, X3) = (A1, A2, A3)$
3:     for $d$ times do                              ▷ In this specification $d = 9$
4:         $(A1, A2, A3) = \Pi(A1, A2, A3, \mathbf{1}^\star)$
5:     end for
6:     $(A1', A2', A3') = (A1, A2, A3) \oplus (X1, X2, X3)$
7: end function
```

Π in Algorithm 1. The initialisation and finalisation phases are identical in the proposed framework and is defined in Algorithm 2.

Since Π given M is an invertible function, we prevent back-tracking of the state from the exposed MAC tag by adding the starting state of the function to the ending state. This is also done during the initialisation phase to achieve an instantiation of the FP(1) property introduced in [HK18].

2.2 Two instances SMAC-1 and SMAC-3/4

While the previous subsection provides a detailed description of the framework, we need some additional specifications to instantiate an implementable algorithm. In this paper we provide two concrete instances called SMAC-1 and SMAC-3/4. They both work as stand-alone integrity algorithms that provide truncatable tags of size τ bits, where the upper limit is $\tau \leq 128$ and $\tau \leq 160$ bits for these two instances, respectively.

The first instance, SMAC-1, has a lower security level but processes a new message block for each round during the compression phase. The second variant, SMAC-3/4, has a higher security level but only processes three message blocks every 3 out of 4 compression round. It does so by running the compression function Π with $M = \mathbf{1}^\star$ every fourth round of the compression phase. The resulting rate is 3/4 of the rate of SMAC-1, neglecting the identical initialisation and finalisation phases.

The two instances SMAC-1 and SMAC-3/4 use distinct permutations σ. The different permutations are given by[4]

$$\sigma_1 = \{0, 7, 14, 11, 4, 13, 10, 1, 8, 15, 6, 3, 12, 5, 2, 9\} \quad \text{for SMAC-1}$$
$$\sigma_{42} = \{7, 14, 15, 10, 12, 13, 3, 0, 4, 6, 1, 5, 8, 11, 2, 9\} \quad \text{for SMAC-3/4}$$

[4] The rational and notation of these σ will be explained in Section 3.

Both of the instances take a 256-bit key $K \in \{\mathbb{N}_8\}^{32}$ and a 128-bit domain separation value $IV \in \{\mathbb{N}_8\}^{16}$ as inputs. Two 128-bit halves of the key are referred by $K = (K0||K1)$. If the original key is shorter then the 256-bit K is constructed from the original shorter key by extending it to 256 bits with zeroes. Exactly how the domain separation is to be done is left for the user of the algorithms, but separating different key and tag sizes together with a nonce should probably be considered. The K and IV are loaded into the registers according to Algorithm 3, before invoking the initialisation phase.

Algorithm 3 `KeyIVLoad` : $(\{\mathbb{N}_8\}^{32}, \{\mathbb{N}_8\}^{16}) \rightarrow (\{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{16})$

```
1: function KeyIVLoad($K, IV$) → $(A1, A2, A3)$
2:     $A1 = K[16 \dots 31]$    ▷ This part is zeroes if the original key is $\leq$ 128 bits long
3:     $A2 = K[0 \dots 15]$
4:     $A3 = IV$
5: end function
```

Algorithm 4 `MessagePrep` : $(\{\mathbb{N}_8\}^{L_A}, \{\mathbb{N}_8\}^{L_C}) \rightarrow \{\mathbb{N}_8\}^{L_A+L_C+P+16}$

```
1: function MessagePrep($\mathcal{A}, \mathcal{C}$) → $\mathcal{M}$
2:     $P_A : \mathbb{N}_4 = 15 - ((L_A + 15) \mod 16)$
3:     $P_C : \mathbb{N}_4 = 15 - ((L_C + 15) \mod 16)$
4:     $tmp_A : \{\mathbb{N}_8\}^{L_A+P_A} = A \parallel \{0\}^{P_A}$
5:     $tmp_C : \{\mathbb{N}_8\}^{L_C+P_C} = C \parallel \{0\}^{P_C}$
6:     $S : \{\mathbb{N}_8\}^{16} = \texttt{LittleEndian64}(8 \cdot L_A) \parallel \texttt{LittleEndian64}(8 \cdot L_C)$
7:     $P : \mathbb{N}_5 = P_A + P_C$
8:     $\mathcal{M} = tmp_A \parallel tmp_C \parallel S$
9: end function
```

The output tag of SMAC-1 and SMAC-3/4 is confined to a maximum of 16 or 20 bytes, and corresponds to the value of the $A2$ (and up to 4 additional bytes of $A3$ in case of SMAC-3/4) register after the finalisation phase.

The last thing to specify is how the message to be integrity protected is processed. Let us assume that there are two parts of the message that need to be integrity protected; one part with plaintext data, and one part with ciphertext data. This is a natural assumption in an AEAD scenario.

Let $\mathcal{A} \in \{\mathbb{N}_8\}^{L_A}$ be the associated plaintext data of length L_A bytes, and let $\mathcal{C} \in \{\mathbb{N}_8\}^{L_C}$ be the ciphertext data of length L_C bytes. We form the input message to SMAC-1 or SMAC-3/4 by firstly pad $\mathcal{A}$ and $\mathcal{C}$ to 16 bytes boundaries by inserting 0:s. Then the ciphertext array is concatenated to the end of the plaintext array. We now append a 16 byte block consisting of the lengths in bits of the messages. The procedure is described in Algorithm 4.

In these instances, we only admit byte oriented inputs and hence we multiply the array length by 8. If bit oriented inputs are needed, simply provide the total number of bits as input and use those values in line 6. The input data arrays need to be byte aligned in any case. The conversion function `LittleEndian64`(n) takes a 64 bit integer n and converts it to a byte array with the least significant byte of n in the first array element. If we write n as $n = n_7 n_6 \dots n_0$ with n_0 being the least significant byte, $N =$ `LittleEndian64`(n) will result in an array

$$N \in \{\mathbb{N}_8\}^8 = \{n_0, n_1, \dots, n_7\}.$$

This puts a restriction of the length of the plaintext and ciphertext messages to be maximum $2^{64} - 1$ bits each.

We can now split the array $\mathcal{M}$ of length $L_{\mathcal{M}}$ into blocks of 16 bytes each. Let us denote those blocks by M^i according to

$$\mathcal{M} = M^0 \parallel M^1 \parallel \dots \parallel M^{L_{\mathcal{M}}/16-1}.$$

Finally, we fix the number of rounds during the initialisation and finalisation phases to $d = 9$ and in Algorithm 5 we provide the complete description of SMAC-1 and SMAC-3/4. Here we assume τ as a constant parameter representing the size of the Tag in bits; recall that the tag size τ is at most 128 or 160 bits, depending on the SMAC variant. The extraction from the state is primarily done from register $A2$ where the bytes of Tag are assigned by the corresponding indices of $A2$. If $\tau > 128$ the Tag array is appended by bytes from register $A3$, starting with the first position.

Algorithm 5 SMAC-1(3/4) : $(\{\mathbb{N}_8\}^{32}, \{\mathbb{N}_8\}^{16}, \{\mathbb{N}_8\}^{L_A}, \{\mathbb{N}_8\}^{L_C}) \rightarrow \{\mathbb{N}_8\}^{16/20}$

```
1: function SMAC-1(3/4)($K, IV, \mathcal{A}, \mathcal{C}$) → $Tag$
2:     $\mathcal{M}$ = MessagePrep($\mathcal{A}, \mathcal{C}$)
3:     Divide $\mathcal{M}$ into $L_{\mathcal{M}}/16$ sub-blocks $M^i$ of size 16 bytes
4:     $(A1, A2, A3)$ = KeyIVLoad($K, IV$)
5:     $(A1, A2, A3)$ = InitFinal($A1, A2, A3$)
6:     for all sub-block $M^i$ in $\mathcal{M}$ do
7:         $(A1, A2, A3) = \Pi(A1, A2, A3, M^i)$
8:         if (variant is SMAC-3/4) and ($i \bmod 3 == 2$) then
9:             $(A1, A2, A3) = \Pi(A1, A2, A3, \mathbf{1}^\star)$
10:        end if
11:    end for
12:    $(A1, A2, A3)$ = InitFinal($A1, A2, A3$)
13:    $Tag = (A2 \parallel A3)_\tau$        ▷ First pick tag bits from $A2$, then from $A3$ if $\tau > 128$
14: end function
```

2.3 Altered instantiations of SMAC

Depending on the use case, the model can be altered. For example, when SMAC is used in AEAD mode paired with an encryption algorithm, the initialisation phase can be skipped by assigning the internal state in time $t = 0$ with three pseudo-random secret values produced by the accompanying cipher.

There could be various other use cases for the SMAC framework, and the exact instances we have provided here is partly to give concrete security bounds and advice on sufficient number of dummy clocks in the initialisation and finalisation phases. If an application needs a different security/performance trade-off, the number of clocks during the initialisation/finalisation phase may be changed.

Minimum d. Depending on the use case, required performance, security demands, the tag size etc., we advise the minimum number of rounds for the initialisation phase must be $d_{\texttt{Init}} \geq 6$ (if not combined with an external cipher), and for the finalisation phase it must be $d_{\texttt{Final}} \geq 4$ rounds (the first time when the tag is influenced by all registers). These absolute minimums are supported by the results of our analyses in Section 3.

2.4 Example of using SMAC-1 with AES in AEAD mode

Assume we are using AES-256-CTR with an IV value consisting of 12 bytes of nonce and domain separation and 4 bytes of counter value, starting at zero with IV_0. The subscript of IV indicates the counter value. The first three keystream

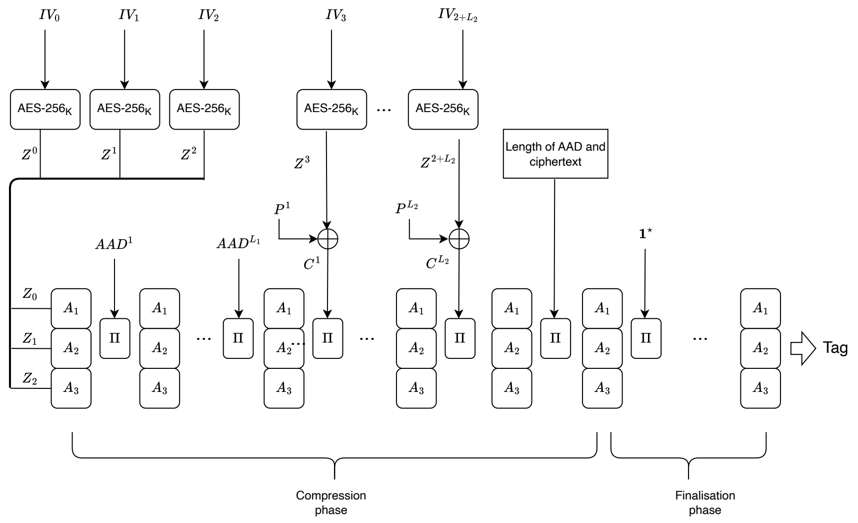

Fig. 3. Example of SMAC usage as AEAD integrity protection together with AES-256-CTR.

symbols are:

$$Z^0 = \text{AES-256}_K(IV_0)$$
$$Z^1 = \text{AES-256}_K(IV_1)$$
$$Z^2 = \text{AES-256}_K(IV_2)$$

where AES-256$_K(P)$ denotes the application of AES-256 on the 16 byte plaintext array P using the key K. These values can be directly loaded into the SMAC registers $(A1, A2, A3)$ and compression of the messages (AAD and ciphertext) can start immediately. This scheme is depicted in Figure 3.

2.5 Security claims and limitations

In the context of the current specification of SMAC-1 and SMAC-3/4, with $d = 9$ for both the initialisation and finalisation phases, we summarise security claims and limitations as given in Table 1.

Security and limitation aspects	SMAC-1	SMAC-3/4
Original key size, κ	$\kappa \leq 256$	$\kappa \leq 256$
Truncated tag size, τ	$\tau \in [12..128]$	$\tau \in [12..160]$
Maximum length of AAD and ciphertext	$2^{64} - 1$	$2^{64} - 1$
Maximum number of messages with the same key	2^{64}	2^{64}
Attack scenarios and complexities		
Security level against a single message forgery in a nonce-respecting setting, in bits (Sec.3.3)	$\geq \min\{\kappa, \tau, 118\}$	$\geq \min\{\kappa, \tau, 152\}$
Security level against input/output differentials in `InitFinal`, in bits (Sec.3.10)	$\geq \min\{\kappa, 56 \cdot 6\}$	$\geq \min\{\kappa, 49 \cdot 6\}$
GnD to recover the key from a known single state and IV (Sec.3.8)	$T = O(2^{\kappa})$	$T = O(2^{\kappa})$
GnD to recover the state from a known tag(s) when (K, IV) is fixed (Sec.3.8)	$T = O(2^{384})$	$T = O(2^{384})$
TMTO to recover the state with multiple known tags (Sec.3.6)	$TMD = O(2^{192})$	$TMD = O(2^{192})$
TMTO to recover the key with a fixed IV and multiple known tags (Sec.3.6)	$TMD = O(2^{\kappa/2)}$	$TMD = O(2^{\kappa/2})$
State recovery with nonce-misuse queries to the sender oracle (Sec.3.12)	$Q \geq O(2^{59})$	$Q \geq O(2^{76})$
State recovery with nonce-misuse queries to the receiver oracle (Sec.3.12)	$Q \geq O(2^{59+\tau})$	$Q \geq O(2^{76+\tau})$

Table 1. Security claims and limitations. In the table we use T=Time, M=Memory, D=Data, Q=Queries, TMD=Time, Memory, and Data.

The nonce-misuse using a sender oracle scenario discussed in Section 3.12 is highly theoretical since if the goal is to send some selected malicious messages to the receiver, the direct solution would be to ask the oracle for the correct tag, instead of using 2^{59} queries in order to recover the state. As will be discussed in Sections 3.6 and 3.8, a state recovery does not directly translate into a key recovery so the state is only valid for that particular (K, IV). The receiver oracle is more plausible in a practical scenario, since if the protocol does not include some replay protection, the receiver can indeed be used to verify correctly guessed state collisions. From the table we conclude that even if the smallest allowed tag size $\tau = 12$ is used, the complexity of a receiver nonce-misuse attack is well above the allowed number of messages to be MACed for that key.

As a good security practice, we advise that each pair (K, IV) should be used only once by the sender side. In order to prevent receiver side nonce-misuse attacks when the protocol does not include replay protection, we advise to introduce a counter of the number of verification requests per a key. If that counter reaches the maximum allowed value (see the last row of Table 1), then the receiver side invalidates the key and e.g. requests a key renegotiation. If there are multiple independent receivers, say m, then each receiver should have the upper limit for the counter been divided by m.

Note that the whole initial state of 384 bits may be initialised with a secret initial state when e.g. SMAC is combined with an external cipher.

3 Security analysis and simulations

3.1 Differential forgery attack

For the sake of notations, some shorter or simpler forms are used in this and further subsections. We use '+' to denote the XOR operation, where applicable depending on the context. Also, for bit-level differential trails, we call them *bit-trails* which depict concrete differential trails in practical attacks. We refer a byte-level trail to a *byte-trail* that indicates whether each byte value is zero or not. This type of trail is good for e.g. counting the minimum number of active Sboxes. A bit-differential message is denoted by ΔM, and its corresponding byte-differential is denoted by $\mu(\Delta M)$.

One of the main security concerns in this construct comes from the second preimage resistance, where an attacker given the first message M tries to construct a new message M' such that the resulting tag would coincide. In this scenario, the attacker may query the verification oracle to check if the tags for M and M' collide.

Here we have two possibilities: either two tags of size τ bits would collide by chance with probability $2^{-\tau}$, or the attacker carefully selects $\Delta M = M + M'$ such that the internal state would collide with probability larger than $2^{-\tau}$. In further subsections, we analyse this scenario in more details and study the differential MAC forgery attack.

In the SMAC compression function, the time frame where the state could collide from a differential input is $\Delta t \geq 3$, because there are three registers that shuffle in a nearly circular fashion, thus at least 3 clocks are neededto recover the internal state (to make the internal state difference become zero) given a differential input.

As the first step of our analysis, we find a differential byte-trail with the minimum number of active Sboxes. The minimum number s of active differential Rijndael Sboxes in a possible differential trail gives an upper bound for the probability of the internal state collision to happen (without clustering effect, which will be analysed in a later subsection), that probability is 2^{-6s}, since a differential trail of a single 8-bit Rijndael Sbox has probability at most 2^{-6}:

$$\forall \delta_x, \delta_y : \quad \Pr\{S(x) + S(x+\delta_x) = \delta_y\} \leq 2^{-6}.$$

As a simplified example, let us derive the state expressions for the first 3 clocks, i.e., where $\Delta t = 3$.

$$\text{1st clock:} \quad \begin{cases} A1^{t+1} = \sigma(A2^t + A3^t + M^t) \\ A2^{t+1} = L\pi S(A1^t) + M^t \\ A3^{t+1} = L\pi S(A2^t) + M^t \end{cases}$$

$$\text{2nd clock:} \quad \begin{cases} A1^{t+2} = \sigma(L\pi S(A1^t) + L\pi S(A2^t) + M^{t+1}) = x + \sigma M^{t+1} \\ A2^{t+2} = L\pi S(\sigma(A2^t + A3^t + M^t)) + M^{t+1} = L\pi S(y + \sigma M^t) + M^{t+1} \\ A3^{t+2} = L\pi S(L\pi S(A1^t) + M^t) + M^{t+1} = L\pi S(z + M^t) + M^{t+1} \\ \text{where: } x = \sigma(L\pi S(A1^t) + L\pi S(A2^t)),\ y = \sigma(A2^t + A3^t),\ z = L\pi S(A1^t) \end{cases}$$

$$\text{3rd clock:} \quad \begin{cases} A1^{t+3} = \sigma(L\pi S(y + \sigma M^t) + L\pi S(z + M^t) + M^{t+2}) \\ A2^{t+3} = L\pi S(x + \sigma M^{t+1}) + M^{t+2} \\ A3^{t+3} = L\pi S(L\pi S(y + \sigma M^t) + M^{t+1}) + M^{t+2} \end{cases} \qquad (1)$$

We want to find a differential $\Delta(M^t, M^{t+1}, M^{t+2})$ such that $\Delta(A1, A2, A3)^{t+3} = 0$, then we have the following differential system:

$$\begin{cases} 0 = \sigma(L\pi S(y + \sigma\Delta M^t) + L\pi S(z + \Delta M^t) + \Delta M^{t+2}) \\ 0 = L\pi S(x + \sigma\Delta M^{t+1}) + \Delta M^{t+2} \\ 0 = L\pi S(L\pi S(y + \sigma\Delta M^t) + \Delta M^{t+1}) + \Delta M^{t+2} \end{cases}$$

and after the substitutions $\Delta M^{t+2} \to L\pi\Delta M^{t+2}$, $\Delta M^{t+1} \to \sigma'\Delta M^{t+1}$, $\Delta M^t \to \sigma'\Delta M^t$, where $\sigma' = \sigma^{-1}$, the system is simplified to

$$\begin{cases} \Delta M^{t+2} = S(y + \Delta M^t) + S(z + \sigma'\Delta M^t) \\ \Delta M^{t+2} = S(x + \Delta M^{t+1}) \\ \Delta M^{t+2} = S(L\pi S(y + \Delta M^t) + \sigma'\Delta M^{t+1}) \end{cases}$$

The attacker starts introducing the first differential in time t, and can introduce up to 3 consecutive differentials, hoping that the state will recover in time $t+3$. Note that, since the compression function Π is reversible, the internal state may only collide in that certain time $t+3$, and not later.

From the second equation it is clear that $\mu(\Delta M^{t+2}) = \mu(\Delta M^{t+1})$, and thus the second equation can be virtually removed from consideration as follows:

$$\begin{cases} \Delta M^{t+2} = S(y + \Delta M^t) + S(z + \sigma'\Delta M^t) \\ \Delta M^{t+2} = S(L\pi S(y + \Delta M^t) + \sigma' S^{-1}(x + \Delta M^{t+2})) \end{cases}$$

In a naïve approach it is obviously possible just to loop over all 16-bit masks $\mu(\Delta M^t)$ and $\mu(\Delta M^{t+2})$ in time 2^{32} and check for feasibility of the above system to have a solution. If a solution to the system is feasible, then the total number of active Sboxes can be computed as [5]:

$$\begin{aligned}\#\ \text{Sboxes} &= HW(\mu(S(y+\Delta M^t))) + HW(\mu(S(z+\sigma'\Delta M^t))) \\ &+ HW(\mu(\sigma' S^{-1}(x+\Delta M^{t+2}))) + HW(\mu(S^{-1}(\Delta M^{t+2}))) \\ &= 2\cdot[HW(\mu(\Delta M^t)) + HW(\mu(\Delta M^{t+2}))].\end{aligned}$$

For SMAC-1 with σ_1 and $\Delta t = 3$ we found the following byte differential whose trail has 22 active Sboxes, which is the minimum in this scenario. Here, $\mu(\Delta M^{t+2})$ is the value before the aforementioned substitution.

$$\begin{aligned}\mu(\Delta M^t) &= (\mathtt{1,0,0,0,0,0,1,0,0,1,0,0,0,1,0,0}) \\ \mu(\Delta M^{t+1}) &= (\mathtt{1,0,0,0,0,1,1,0,0,1,1,0,0,1,0,1}) \\ \mu(\Delta M^{t+2}) &= (\mathtt{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1})\end{aligned}$$

One of many possible *bit-level* differential vectors that correspond to the above *byte* differential is given below. More details on the trail can be found in Appendix D.

$$\begin{aligned}\Delta M^t &= (\mathtt{80,00,00,00,00,00,59,00,00,0e,00,00,00,84,00,00}) \\ \Delta M^{t+1} &= (\mathtt{ed,00,00,00,00,44,40,00,00,32,24,00,00,ac,00,10}) \\ \Delta M^{t+2} &= (\mathtt{c3,11,11,33,c0,80,40,40,03,02,01,01,64,ac,c8,64})\end{aligned}$$

3.2 Simulations to find a strong permutation σ, generic tool, and byte-trails for $\Delta t \geq 3$ in SMAC-1 and -3/4

From the previous subsection where we have derived expressions for a differential forgery attack in the time frame $\Delta t = 3$, it becomes clear that the minimum number of active Sboxes depends heavily on the exact permutation σ. For example, there are weak permutations where the best differential trail may involve only a few active Sboxes, and thus the success probability of the forgery attack is high. Therefore, one of the goals in this work was to find strong permutation candidates, which can provide as high level of security as possible for the considered SMAC framework.

SMAC-1 vs. -3/4. Searching for a strong permutation for the instance SMAC-3/4 is similar to searching for the instance SMAC-1. We simply insert artificial message blocks with a fixed value $\mathbf{1}^\star$, representing the dummy clocks that happen in SMAC-3/4 after each three blocks of messages. Thus, the analysis for SMAC-1 and SMAC-3/4 can be carried out in the same way, with the

[5] Note that for $\Delta t = 3$ the number of active Sboxes is always an even number. I.e., if we filter for 26 active Sboxes and do not find permutations giving at least 25 in other $\Delta t > 3$, then the next step down would be $\Delta t = 3$ with 24 active Sboxes.

additional constraint that every 4th difference for SMAC-3/4 will be zero, i.e. $\Delta M^t = 0$ due to $M^t = \mathbf{1}^\star$ in every fourth time instance t. However, when analysing SMAC-3/4 and depending on Δt there may be distinct cases where those zero differences occur. For example in $\Delta t = 5$ we can have three sub-cases of a differential message, depending on t or where the dummy clocks actually happen:

$$\Delta M = (\Delta M^t \neq 0, \Delta M^{t+1} = 0, \Delta M^{t+2} = *, \Delta M^{t+3} = *, \Delta M^{t+4} \neq 0)$$
$$\Delta M = (\Delta M^t \neq 0, \Delta M^{t+1} = *, \Delta M^{t+2} = 0, \Delta M^{t+3} = *, \Delta M^{t+4} \neq 0)$$
$$\Delta M = (\Delta M^t \neq 0, \Delta M^{t+1} = *, \Delta M^{t+2} = *, \Delta M^{t+3} = 0, \Delta M^{t+4} \neq 0)$$

where the first and last must be nonzero differences (if one of them is a zero difference then the case falls into another scenario with shorter Δt due to the reversible compression function). The dummy clock can occur at position $t+1, t+2$, or $t+3$. The other positions can have any differential value.

There are $16! \approx 2^{44}$ possible permutations and in this work we aim at performing cryptanalysis on all 2^{44} instances of SMAC, each having a distinct permutation, to extract the strongest candidates. Testing 2^{44} permutations is a challenging task. We were able to narrow it down using several steps of fast filtering and pattern matching. The search was performed in two data centers[6] with thousands of compute nodes to test these permutation candidates in parallel.

First round of filtering. We have written highly optimised filters for two time frames $\Delta t = 3$ and $\Delta t = 4$ in C/C++. Given a permutation candidate σ, these filters test whether there exists a byte-trail with less number of minimum Sboxes than some preselected threshold. If the number of active Sboxes is below the threshold, the candidate is removed. After the first round of filtering, we obtained the following number of remaining candidates:

- SMAC-1. Filters for $\Delta t = 3$ and $\Delta t = 4$ both with the threshold to have at least 20 active Sboxes in the optimal differential trail resulted in 73073 remaining candidates out of 2^{44}.
- SMAC-3/4. Only one filter was applied in the first round – the case $\Delta t = 3$ with at least 26 active Sboxes, resulted in 1.6 billion candidates[7].

Second round of filtering. With a larger time frame $\Delta t > 4$ it became quite difficult to code specialised filtering functions. We instead used the CP-SAT optimisation solver from OR-Tools [PD] to perform cryptanalysis of the

[6] We used LUNARC at Lund University and E2C at Ericsson.

[7] We did not include the filter for $\Delta t = 4$ in the SMAC-3/4 search since the performance for this filter at high thresholds wasn't fast enough. Also, with the threshold of 26 we initially hoped to get a minimum of 25 active Sboxes overall (also for $\Delta t > 3$), but additional analysis and simulations showed that there are no permutations that have a minimum of 25 active Sboxes. Then we settled for targeting 24 active Sboxes, while already having 1.6B candidates from the first round, and we received the short list of promising candidates among these candidates having at least 24 active Sboxes in $\Delta t > 3$. Thus, a separate round of first-phase heavy simulations for $\Delta t = 3$ and threshold 24 could be done, but we assessed it would not be advantageous.

remaining candidates in larger time frames up to $\Delta t = 9$. A generic tool was constructed by utilising Constraint Programming models to test one or a set of permutation candidates, searching for a differential byte-trail with the minimum number of active Sboxes. Our generic tool supports different parameters such as the time frame Δt, position(s) of the dummy message block(s), threshold for the minimum number of Sboxes, et cetera. This tool can also be used to verify and/or test permutations and experiment with SMAC rates other than 1 and 3/4 as given in this paper [8].

Patterns. An additional technique to filter out candidates is to collect patterns of weak permutations. If we test a concrete permutation and a filter finds a byte-trail that has less active Sboxes than the desired threshold, we can further examine the permutation indices to spot which indices are contributing to that weak behaviour. It turns out that in many cases there is a particular set of indices that give rise to the low number of active Sboxes, and those permutation indices form a group of candidates that instantly can be skipped during filtering. By saving such groups as patterns we can significantly reduce the number of permutations we need to test. We collected ~50 million of unique patterns [9] that helped to truncate the search space by a lot.

Simulation results. In Appendix A, we give a short list of the strongest permutation candidates found for SMAC-1 and SMAC-3/4, analysed in the time frame $\Delta t = 3 \ldots 9$. Where the simulation running time was too long to find the optimal byte-trail, we provide a lower bound for the minimum number of active Sboxes reported by the tool. Notably, for SMAC-1, there are only two permutations out of 2^{44} that have at least 20 active Sboxes in all tested Δt, and for SMAC-3/4 we found candidates with at least 24 active Sboxes.

3.3 Clustering effects and forgery success probability

A differential trail typically contains an input difference, output difference, and differences of intermediate variables. Usually, only the input and output differences are known to an attacker. In this case, trails that have the same input and output differences but distinct intermediate differences can cluster together to form a differential with a higher probability, which is called *clustering effect*. Several previous works [BdSF+22,SII23,LPS21] have shown the power of this effect on various ciphers. Therefore, we need to study the clustering effect for the permutation candidates obtained in the previous step of filtering. We have again utilised the CP-SAT solver [PD] from OR-Tools for this part of analysis.

Adaptation of byte-trails for analysis of bit-level clusters. For SMAC, an attacker can prepare a new message that differs from another known message by ΔM, and hope the internal state of SMAC would be the same after a number of clocks, thus resulting in the same MAC value. All intermediate differential

[8] The tool, along with a reference implementation and test vectors, is available at GitHub: `https://github.com/ONG/smac-tools`

[9] As a side note, we have also developed optimised algorithms for handling patterns, such as collecting, sorting, merging, and checking for uniqueness in time $O(N \log N)$.

bit-trails that have the same ΔM are in the same cluster. However, testing all possible clusters and identifying the strongest one is difficult, since there is an exponential number of concrete input bit-differentials ΔM, and for each ΔM the analysis of its cluster requires enumeration of all intermediate bit-trails. To make this analysis feasible, we study the clustering effect on the level of byte-trails, instead. In particular, we aim to find upper bounds on probabilities of clusters corresponding to only those message differentials that contain optimal byte-trails with the minimum number of active Sboxes. As the result, these bounds apply to all bit-trails that follow the same byte-trails.

Analysis of a single byte-level cluster. A single cluster C on the byte level is identified by a concrete fixed byte-differential $\mu(\Delta M)$. Given $\mu(\Delta M)$, we enumerate all intermediate byte-trails which match that certain $\mu(\Delta M)$. Let that set of byte-trails be denoted by S_C. Then the probability of the cluster C is upper bounded by

$$p(C) \leq \sum_{\Psi \in S_C} n_\Psi \cdot max_p(\Psi)$$

where n_Ψ is the number of possible bit-trails matching a certain intermediate byte-trail $\Psi \in S_C$, and $max_p(\Psi)$ is the maximum probability of a single bit-trail that follows the byte-trail Ψ.

In our analysis, we assume that all byte-trails from S_C can be mapped to the same ΔM on the bit-level as well. Therefore, the resulting cluster C, in the way we construct it and taking into account n_Ψs, cannot be smaller than a corresponding valid bit-level cluster, and thus our bound of $p(C)$ provides a theoretical upper bound for the forgery success probability. This also means that in reality the forgery success probability on SMAC is not greater, but likely smaller than what we have derived.

Method to compute n_Ψ. Let us for the moment assume that, given the byte-trail Ψ, its corresponding byte-differential $\mu(\Delta M)$ is assigned with some (unknown to us) fixed bit-level difference ΔM. We propose a simple method to determine whether all other intermediate differential bytes of the byte-trail Ψ can be uniquely derived *on the bit-level*, given a hypothetical fixed bit-differential value of ΔM. Recall that in each round the differential propagates as

$$\begin{cases} \Delta A1^{t+1} = \sigma(\Delta A2^t + \Delta A3^t + \Delta M^t) \\ \Delta A2^{t+1} = L\pi S(\Delta A1^t) + \Delta M^t \\ \Delta A3^{t+1} = L\pi S(\Delta A2^t) + \Delta M^t \\ \Delta A1^{t+2} = \sigma(\Delta A2^{t+1} + \Delta A3^{t+1} + \Delta M^{t+1}) \end{cases} . \tag{2}$$

To simplify the analysis, the differential distribution table (DDT) of Rijndael Sbox is ignored, and we only consider whether involved Sboxes are active or not. Furthermore, we assume that $\Delta A1^t$, $\Delta A2^t$, and $\Delta A3^t$ are known, and thus $\Delta A1^{t+1}$ is uniquely determined by ΔM on the bit-level due to Eq. (2). Following the given byte-trail Ψ, some bytes can only take the value 0, and for non-zero bytes the following rules are applied repeatedly to determine (most of) the remaining bytes of Ψ on the *bit-level* given Eq. (2), where $\Delta A2^{t+1}[i]$ means the i-th byte of $\Delta A2^{t+1}$ and so on for other values.

1. If $\exists i : \Delta A2^{t+1}[i]$ is known, then $L\pi S(\Delta A1^t)[i] = \Delta A2^{t+1}[i] + \Delta M^t[i]$.
2. If $\exists i : \Delta A3^{t+1}[i]$ is known, then $L\pi S(\Delta A2^t)[i] = \Delta A3^{t+1}[i] + \Delta M^t[i]$.
3. If $\exists i : \sigma^{-1}(\Delta A1^{t+2})[i] = 0$ and $\Delta A2^{t+1}[i]$ are known, then $\Delta A3^{t+1}[i] = \Delta A2^{t+1}[i] + \Delta M^{t+1}[i]$.
4. If $\exists i : \sigma^{-1}(\Delta A1^{t+2})[i] = 0$ and $\Delta A3^{t+1}[i]$ are known, then $\Delta A2^{t+1}[i] = \Delta A3^{t+1}[i] + \Delta M^{t+1}[i]$.
5. If the number of unknown byte values in $S(\Delta A1^t)$ (resp. $S(\Delta A2^t)$) is less than or equal to the number of known byte values in $L\pi S(\Delta A1^t)$ (resp. $L\pi S(\Delta A2^t)$), then $S(\Delta A1^t)$ and $L\pi S(\Delta A1^t)$ (resp. $S(\Delta A2^t)$ and $L\pi S(\Delta A2^t)$) are all determined.

Note that $\Delta A1^0 = \Delta A2^0 = \Delta A3^0 = 0$. This way, this method propagates the differential knowledge to intermediate bytes of Ψ round by round, and stops when no new differential bytes can be determined *on the bit-level*. Although the above rules are described on the bit-level when a hypothetical ΔM is known, in our simulations we perform the same steps but on the byte-level where each variable is a binary known/unknown flag.

It is surprising that for almost every Ψ we have tested all intermediate bytes can be uniquely determined after applying these rules, except for a few cases. I.e., given a byte-trail Ψ, there is in most cases only one corresponding bit-trail, which means $n_\Psi = 1$.

In other cases where this method could not determine all values, we have noticed that the remaining (undetermined) bytes are all in the last round and all output from active Sboxes. By checking the number of free variables in the linear system during the 5-th rule, the number of possible bit-trails can be upper bounded. Due to the DDT of the Sbox, the number of possible output differences is 2^7 given the input difference. Hence, every free variable in the linear system can only take up to 2^7 possible values. If there are x free variables, then n_Ψ is set to 2^{7x}.

Simulations and results. We now turn to the short list of promising permutations in Table 9 derived in the previous filtering stage. Due to an extremely high complexity of cluster analysis, we picked 5 permutations for SMAC-1 and 9 for SMAC-3/4, such that they cover all distinct characteristics (the vector of minimum number of Sboxes for various Δt-scenarios).

For every SMAC variant, permutation, and attack scenario Δt and $(\Delta t, k)$ (where k defines dummy clocks), we first enumerate all clusters identified by distinct byte-differentials $\mu(\Delta M)$ that include at least one optimal byte-trail with the minimum number of active Sboxes. Then for each such cluster we compute the upper bound of the forgery attack success probability by using the above method, where the enumeration of intermediate byte-trails as well as computation of $max_p(\Psi)$ was done with OR-Tools. In the end, we get the maximum probability over all attack scenarios for each SMAC variant and permutation, from where the most secure permutations are determined.

The results of cluster analysis[10] for SMAC-1 are given in Table 2, and a similar table for SMAC-3/4 is given in Table 10 in Appendix B.

Case		A few selected permutations from Table 9 for SMAC-1				
		σ_1	σ_3	σ_{11}	σ_{13}	σ_{17}
$\Delta t = 3$	$b:c$	4:4	2:2	16:16	2:2	16:16
	p	2^{-134}	2^{-121}	2^{-152}	2^{-121}	2^{-153}
Sboxes	#	22	20	24	20	24
$\Delta t = 4$	$b:c$	2496:192	534:192	60:44	382:80	2:2
	p	$2^{-121.21}$	$2^{-120.44}$	$2^{-124.41}$	$2^{-120.83}$	2^{-122}
Sboxes	#	20	20	20	20	19
$\Delta t = 5$	$b:c$	4032:288	256:68	900:140	248:64	866:130
	p	$2^{-121.25}$	$2^{-115.67}$	$2^{-119.41}$	$2^{-115.88}$	$2^{-118.40}$
Sboxes	#	20	19	19	19	19
$\Delta t = 6$	$b:c$	18321:96*		2352:232		
	p	$2^{-118.95}$	—	$2^{-123.56}$	—	—
Sboxes	#	20	19	20	19	20
$\Delta t = 7$	$b:c$	30264:46**				
	p	$2^{-129.27}$	—	—	—	—
Sboxes	#	21	19	22	19	22
Max.	p	$\mathbf{2^{-118.95}}$	$2^{-115.67}$	$\mathbf{2^{-119.41}}$	$2^{-115.88}$	$\mathbf{2^{-118.40}}$

* Only these clusters are found in practical time.

** Only trails that have up to 24 active Sboxes were enumerated.

Table 2. Clusters of differential trails of SMAC-1. In this table, b is the number of byte-trails, c is the number of clusters, and p is the probability of the cluster.

For SMAC-1, we found that σ_1, σ_{11}, and σ_{17} are the three strongest candidates with similar security levels. Our preference goes to σ_1 for the reason that it is one of only two permutations out of 2^{44} that has minimum 20 active Sboxes in all Δt scenarios; also (though it is not relevant for the SMAC construct), σ_1 could as well be utilised for a SNOW-like stream cipher since it has a higher level of resistance against correlation attacks. We then claim that our final choice is σ_1 and it provides at least 118 bits of security level against forgery attacks.

[10] We skipped testing some scenarios for certain permutations since the maximum probability was already larger than another permutation candidates had at the time of simulations, and thus unnecessary simulations on the already worse case would only take resources with no influence on the final result. Also, some heavy cases such as ($\Delta t = 6, k = 2$) and ($\Delta t = 7, k = 1, 5$) for SMAC-3/4 were also skipped, since all remained "good" permutations have at least 27 and 31 active Sboxes in their best byte-trails for these scenarios, respectively, which is already much larger than the minimum 24 and therefore would not cross the maximum probability already detected by testing other cases of the same permutations. Finally, for a few cases where $\Delta t \geq 6$, the models in OR-Tools became too big and the number of byte-trails was huge, so only those clusters with the highest chance to break through the security level were checked thoroughly.

For SMAC-3/4, we found that σ_{37} and σ_{42} are the strongest candidates with a similar level of security. However, in this case, we would prefer σ_{42} with a slightly better security, due to all other secondary considerations of these two permutations are the same. Our final choice for SMAC-3/4 is σ_{42} that provides at least 152 bits of security level against forgery attacks.

As the number of active Sboxes grows rapidly with larger Δt, we believe that the existence of a forgery differential attack, with complexity much lower than the claimed security level and time frame $\Delta t > 9$, is not likely.

3.4 The constant $\mathbf{1}^\star$

The MixColumn linear transformation has a specific property such as if the input bytes are all equal, $X = \{x\}^{16}$, then the result $Y = \texttt{MixColumn}(X)$ preserves the same property and $Y = \{y\}^{16}$. All other operations, such as XOR, σ, ShiftRows, and SubBytes, also preserve this property. I.e., if all three registers $A1, A2, A3$ have that property in some certain time, then that property preserves over rounds *if* the message block $M = 0$, which may, in particular, affect the randomness of the initialisation and finalisation phases. In the initialisation phase, this property can further generate a weak key class. In order to remove this property, we add $\mathbf{1}^\star$ as the round key to the state during the InitFinal function, as well as for dummy clocks in SMAC-3/4. Moreover, the implementation of the constant is "cheap" in both software and hardware.

3.5 The PRP-PRF switch

The ending XOR with the input in the function InitFinal converts the scheme from a pseudo-random permutation (PRP) to a non-invertible pseudo-random function (PRF), similarly to the FP(1) mode of operation in stream ciphers [HK15]. This protects both the secret key and the state sequence. For example, suppose the state in some time instance is recovered, say, through a side-channel attack. In that case, it is not possible to revert the state back to the start of the initialisation phase and recover the secret key, the highest asset to be protected. Moreover, since t bits of the internal state become the final MAC tag value, it also makes sense to make the end of the finalisation phase more protected.

The InitFinal procedure can be simplified as

$$Y = \Pi^d(X) \oplus X,$$

where X, Y are 384-bit variables and Π is the SMAC round function with $\mathbf{1}^\star$as the message. The ending $\oplus X$ converts the PRP Π into a PRF. This is a standard technique and is used in many designs, for example, in MILENAGE for computing OP_c from OP [3GP], or in Grøstl [GKM+09] for the output transformation. The theoretical security of the finalisation function may be derived from e.g. the security proof of Davies-Meyer construct where $g(k, m) = E_k(m) \oplus m$ is proved to be a collision-resistant one-way function, given that E_k is an ideal block cipher [BRS02] and the same applies when the key k is fixed.

3.6 Internal state size and TMTO attacks

Assume that an attacker can observe the full 384-bit output Y (and not just the tag that is of maximum size 160 bits). The question is: what is the complexity of reverting Y into X? In case of PRP that would be a 1-to-1 mapping and the reverting algorithm is trivial – just clock backwards d times. However, in case of a PRF that mapping would in most cases have between 0 to 2 solutions and it is not trivial how to revert it as $\oplus X$ may be viewed as a masking of $\Pi(X)$. To revert that PRF, one may try a TMTO trade-off attack of complexity $T = M = D = O(2^{192})$ by just building a table of $M = O(2^{192})$ (X, Y) pairs and then ask for $D = O(2^{192})$ different Y's. This is a state-recovery attack. The full Y is not available from the SMAC tag, but if the nonce is misused in the verification oracle one can combine a few accepted tags from the same IV and fixed messages to form an output Y unique for each X. If the tag size is 32 bits then 2^{40} calls to the verification oracle are sufficient to get on average 2^8 accepted tags. But the complexity and data grow by at least a factor 2^{40}.

Generic key-recovery TMTO attacks are similarly also valid for our construct. For example, one can create a large table that maps a subset of the key and IV space to MAC tags for a predefined set of messages, and when the attacker observes tags also found in the table the full key is recovered. This TMTO attack would have a complexity around $T = M \approx O(2^{192})$ with data $D = O(2^{192})$ tags generated from different keys and IV pairs. Better is to fix the IV and have the same TMTO attack on the key only, requiring $T = M = D \approx O(2^{128})$. Allowing a large precomputation cost, one can reduce the memory and data cost by Hellman's approach [Hel80] and building Rainbow tables. It does not, however, offer better performance than the generic case of a search for any 256-bit key.

The state size 3×128 bits ensures a high enough resistance against internal state collision attacks in birthday paradox and TMTO settings. A single-state collision may happen naturally among 2^{192} collected pseudo-random states, but it does not impact the claimed forgery security levels.

3.7 The avalanche effect on full registers

A brief analysis of the initialisation/finalisation phases can be given by the avalanche effect on the level of registers depending on the number of clocks d. The results are given in Table 3 where, for the sake of notation, by k^x we denote that the initial value of Ak has been involved x times in a nonlinear expression for the resulting register after d clocks. As the result, after $d = 3$ clocks the register $A1$ already involves all three input registers, and after $d = 5$ clocks each of the three registers involves the whole input state.

3.8 Guess and determine attacks

In this section, we consider the `InitFinal` function

$$(A1', A2', A3') = \Pi^d(A1, A2, A3, \mathbf{1}^\star) \oplus (A1, A2, A3),$$

clocks, d	0	1	2	3	4	5	6	7	8	9
$A1$	1^1	$2^1 3^1$	$1^1 2^1$	$1^1 2^1 3^1$	$1^1 2^2 3^1$	$1^2 2^2 3^1$	$1^2 2^3 3^2$	$1^3 2^4 3^2$	$1^4 2^5 3^3$	$1^5 2^7 3^4$
$A2$	2^1	1^1	$2^1 3^1$	$1^1 2^1$	$1^1 2^1 3^1$	$1^1 2^2 3^1$	$1^2 2^2 3^1$	$1^2 2^3 3^2$	$1^3 2^4 3^2$	$1^4 2^5 3^3$
$A3$	3^1	2^1	1^1	$2^1 3^1$	$1^1 2^1$	$1^1 2^1 3^1$	$1^1 2^2 3^1$	$1^2 2^2 3^1$	$1^2 2^3 3^2$	$1^3 2^4 3^2$

Table 3. The avalanche effect on registers depending on the number of clocks.

and study the complexity of a generic guess-and-determine attack for various scenarios where some of the input/output register values are known, and we want to derive the remaining values through guessing the smallest number of other unknown bytes. We will model relations on the byte level, and while all operations are simple, it is only `MixColumn` that is more complex to model which includes 56 relations per a single 4-to-4 byte `MixColumn`. In order to find a (almost) smallest guess base for our GnD attack scenarios, we utilise the tool Autoguess from [HE22], as well as our own developed tool that is described in Appendix C in brief.

We have three sets of GnD scenarios. First of all, in G1 we would like to understand how good that PRF function is, i.e., given the complete output, how many bytes need to be guessed to revert that PRF back to the input. In G2 we consider the case when two input states are related through, e.g., a known differential, and both output states are fully available to the attacker. We see that the additional knowledge of an extra output state in G2 does not much help in a GnD attack as the complexity to recover the initial state is similar to G1.

The next set of scenarios G3-5 addresses the security of the initialisation phase where, as a hypothetical assumption, we let the whole state after initialisation to be known to the attacker, as well as some values of the input registers. In these scenarios we are interested in the minimum guess base to recover the missing input register (or even a single byte[11]), where the secret key may actually be settled. These attack vectors may become realistic if e.g. one device performs the initialisation and bypasses the computed output state to the second device for actual MACing, but that second device may be compromised.

In the third set of scenarios G6-8, we analyse the finalisation part where, given the knowledge of one or more output bytes (e.g. through the MAC tag), we wonder about the complexity to recover the internal state before the finalisation phase.

The absolute security level for all these scenarios is that guessing at most 1/16/32/48 bytes of the unknown input registers is enough to recover all other variables – we call it as a *trivial guess.*

All GnD scenarios and the smallest size of the guess base that we managed to derive and observe by using heuristic tools are given in Table 4. Since these tools are heuristic, a smaller guess base may still exist. However, the results that

[11] G5 simulates a scenario when all key bytes except one are guessed and the complete output is also known; it demonstrates that with $d \geq 7$ clocks that single byte is still an unknown variable and cannot be determined through all other 95 known bytes.

Scen-ario	Trivial guess	Known input/output registers and bytes	Number of rounds, d									
			1	2	3	4	5	6	7	8	9	10
General: How strong the stand-alone PRF function is.												
G1	48	All output regs. $A1', A2', A3'$	8	16	16	28	31	32	42	47	48	48
General: How difficult to recover the state given two related input states and the two output states are fully known.												
G2	48	All 2×48 output bytes	8	16	16	28	31	32	40	48	48	48
Initialisation: How difficult to recover the Key registers (or even a single byte) given the complete state after initialisation and some values at loading time.												
G3	32	One of $A1, A2, A3$ and $A1', A2', A3'$	0	0	0	10	15	16	24	28	32	32
G4	16	Two of $A1, A2, A3$ and $A1', A2', A3'$	0	0	0	0	0	0	8	12	16	16
G5	1	Any 47 input bytes and $A1', A2', A3'$	0	0	0	0	0	0	1	1	1	1
Finalisation: How difficult to recover the state given output tags taken from various registers.												
G6	48	One of output regs. $A1', A2', A3'$	32	32	32	32	32	32	42	47	48	48
G7	48	20 output bytes $A1', A2'_{[0..3]}$	28	28	28	31	32	34	45	48	48	48
G8	48	20 output bytes $A2', A3'_{[0..3]}$	28	28	28	30	33	32	46	48	48	48

Table 4. The observed minimum sizes of guess bases for both σ_1 and σ_{42} under various scenarios received from heuristic tools, in terms of the number of bytes to be guessed.

we received are still good indications on what the size of the guess base can be, and how it grows with the number of rounds d.

Notably, the results of these simulations demonstrate that with $d = 9$ we seem getting the absolute maximum possible security level in all GnD scenarios, although many of them are only theoretical. To note, the highest security level of 384 bits is not really needed as it is already much larger than we claimed for SMAC, thus the number of rounds d could actually be lower for certain applications, and may also be different for the initialisation and finalisation phases.

3.9 MDM and cube tests

In this section, we perform the MDM test and cube attack on `InitFinal` to check how many rounds are needed to fully mix the input bits. The initial state in time $t = 0$ is supposed to be pseudo-random by the initial d clocks, which shuffles the input parameters and the secret key. The first preimage resistance should be ensured by the ending d clocks, which makes it hard to find a message that results in a particular hash value. Note that practical distinguishers based on these two methods require fixing some input bits to known values and enumerating another subset of input bits which is called a *cube*. Meanwhile, both methods have to compute the summation of the outputs. However, these requirements cannot be satisfied simultaneously for either the initialisation or finalisation phase. Because the output of the initialisation phase and the input of the finalisation phase are secret to attackers. To analyse both phases, we view them as the same standalone function, `InitFinal`, and assume that all the inputs and outputs can be obtained.

In MDM test for a Boolean function, one fixes values to the input bits outside the cube, enumerating all possible values of the bits in the cube, and then sums

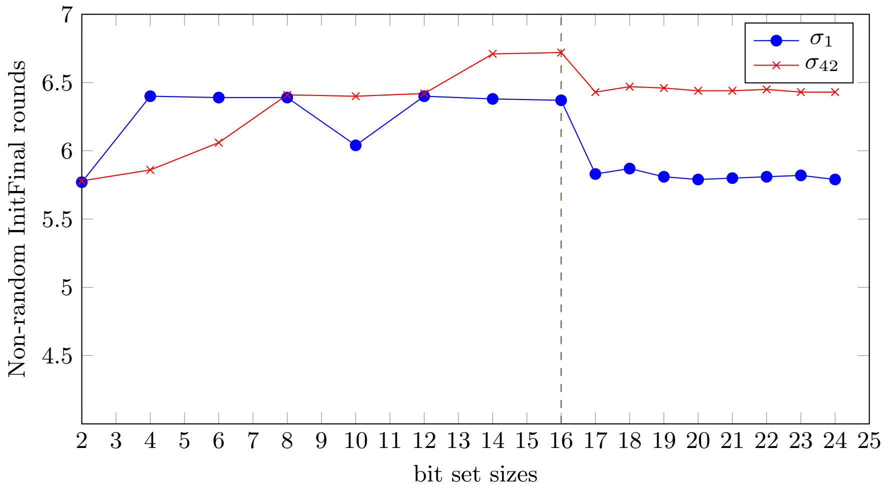

Fig. 4. MDM tests of `InitFinal`.

the output values of the function. For a random Boolean function, the result will be 1 with probability 1/2 while there is a bias for a non-random one. [Sta10] provides a greedy algorithm to find a good cube that detects non-randomness through ciphers. We regard the 384 output bits of `InitFinal` as 384 Boolean functions and take this algorithm to check their non-randomness. Our test starts with the worst 2-bit set that shows the longest non-random rounds. In each step, we add two new bits that give the worst randomness. When the size of the bit set reaches 16, the time complexity of finding the next two bits is too high, so we have to switch to adding one new bit in the next steps until the bit set has 24 bits. To test with a larger bit set, a more powerful computer is needed. Our results are shown in Figure 4. It can be seen that the first 7 rounds fail the MDM test, ensuring a good mixing effect.

Rounds, d	3	4	5	6	7	8	9
cube size $\|I\|$	7	7	7	103	103	128	128
degree d	21	126	231	231	255	254	255
involved key size $\|J\|$	24	152	256	256	256	256	256
time complexity	$2^{30.99}$	2^{159}	$> 2^{256}$	$> 2^{256}$	$> 2^{256}$	$> 2^{256}$	$> 2^{256}$

Table 5. Cube attacks on reduced-round of the initialisation phase (the results are the same for both σ_1 and σ_{42}).

In cube attack, by computing the summation of the outputs, attackers aim to recover the superpoly of the chosen cube. After the division property was proposed by Todo [Tod15], it was further used in [TIHM17] to find the set of key bits J that are involved in the superpoly of the given cube I. This method was improved by Wang *et al.* [WHT+18], which reduces the time complexity of the attack to $2^{|I|} \cdot \sum_{i=0}^{d} \binom{|J|}{i}$ where d is the degree of the superpoly. We evaluate the security of `InitFinal` against cube attacks by using the method described in these papers. Our model is a MILP model. The linear operations can be described by the models of XOR and COPY from [XZBL16] while the model of the Rijndael Sbox is given by [Tod15]. Several different cubes were tested and Table 5 shows the best results found by our model. One can see that, after 8 rounds, the degree is almost full which matches the result in Table 7. Meanwhile, all key bits are involved in this superpoly and the time complexity of the cube attack is larger than 2^{256}.

3.10 (Key, IV, Msg) differential attack and loading registers

Another test that we performed is a differential analysis where the initial state after loading $(A1, A2, A3)$ may have a difference $\Delta(A1, A2, A3)$, then we check the minimum number of active Sboxes after d initialisation rounds that bring the state to any other difference $\Delta(A1', A2', A3')$ (which may be zero or nonzero). One may argue that since IV and IV$'$ for the pair of messages can be selected or even be fixed to certain values, it might help the Key-differential to propagate through the initialisation phase more efficiently. However, since the IV is loaded into $A3$, we see that the very first clock would compute $A3 \oplus A2$ where $A2$ is the part of the secret key, thus making the intermediate result unknown. This way, considering a differential over both IV and Key parts would be a generic differential attack on the initialisation phase. This observation motivates to reserve the lower part of the key for $A2$, then if the original key is larger than 128 bits, the remaining bits to be placed into $A1$.

Rounds, d		3	4	5	6	7	8	9	10
Number of active Sboxes	σ_1	5	6	13	30	32	40	56	70
	σ_{42}	5	6	13	30	32	40	49	64

Table 6. Minimum number of active Sboxes of different rounds.

In our simulation, we find the optimal trails that activate the minimum number of Sboxes in the initialisation phase for each value of d and the results are given in Table 6.

Let us take $d = 6$ as an example to explain these results. In order to bring the difference of the internal state to zero either before the compression phase or utilising the first difference of the message, the attacker would still have to deal with at least 30 active Sboxes, which makes the collision probability to be upper bounded by 2^{-180} (though without clustering effect). This probability is much smaller than the target 2^{-160} in SMAC-3/4 and does not violate the claimed security level.

3.11 Output MAC tag registers

Finally, in order to determine which registers should serve as the source of the final tag, we performed yet another MILP-aided analysis which resulted in the degree of the Boolean functions of the registers' bits. This method comes from [WHT$^+$18] where the authors used it to determine degrees of superpolies. It is also suitable to evaluate the degree of an arbitrary Boolean function. For different rounds, our model gave the degree of the Boolean function corresponding to each output bit. We notice that the degrees are the same for all the bits within each register. The results are provided in Table 7 where some of them are ranges since the MILP model cannot be completely solved in practical time. We emphasise that the maximum degree is 383. This is because, without the PRP-PRF switch, `InitFinal` is a permutation whose degree is upper bounded by 383 and the switch does not change the final degree.

Rounds, d	3	4	5	6	7	8	9	10
A1	28	49	196	280	[352, 356]	376	382	383
A2	49	133	280	[352, 356]	376	382	383	383
A3	28	196	232	[352, 356]	372	382	383	383

Table 7. Degrees of Boolean functions of the registers.

From the results given in Table 7, we see that the degree of Boolean functions of the register $A1$ is always slightly behind the other two registers $A2$ and $A3$, for different ds. An obvious reason is that $A1$ is the linear combination of the previous $A2$ and $A3$ which does not increase the degree. Although for $d = 9$ the output choices $(A1 \parallel A2)_\tau$ and $(A2 \parallel A3)_\tau$ do not differ much, in case of shorter finalisation $d < 9$ the latter becomes more preferable. This motivates us to produce the output tag from $A2$ first, and if more bits are needed (in SMAC-3/4 with $\tau > 128$) then we take them from $A3$.

3.12 State recovery attack using nonce-misuse queries

Atomic step. As a simplified scenario, let us demonstrate feasibility of a state recovery using nonce-misuse queries when (K, IV) is fixed. Recall a differential forgery success probability as discussed in Sections 3.1 and 3.3. Let us take the case SMAC-1, $\Delta t = 3$, and assume we get two messages M and $M' = M + \Delta M$ where the message difference ΔM only happens during the time width $[t..t+2]$ and results in $\Delta(A1, A2, A3)^{t+3} = 0$ after these 3 clocks. In this case we have a state collision. Now recall the middle equation from Eq. (1) which is $A2^{t+3} = L\pi S(x + \sigma M^{t+1}) + M^{t+2}$, where x, y, z are one-to-one substitutions from $(A1, A2, A3)^t$. Since $\Delta A2^{t+3} = 0$, and both M and M' are known to us, we derive:

$$\begin{aligned}
L\pi S(x+\sigma M^{t+1})+M^{t+2} &= L\pi S(x+\sigma(M^{t+1}+\Delta M^{t+1}))+(M^{t+2}+\Delta M^{t+2})\\
\Rightarrow L\pi S(x') &= L\pi S(x'+\sigma\Delta M^{t+1})+\Delta M^{t+2},\ \text{where } x'=x+\sigma M^{t+1}\\
\Rightarrow S(x') &= S(x'+a)+b,\ \text{where } a=\sigma\Delta M^{t+1}, b=(L\pi)^{-1}\Delta M^{t+2}
\end{aligned}$$

Due to the state collision, for any $i \in [0..15]$ we get either $a_i = b_i = 0$ or $a_i \neq 0 \wedge b_i \neq 0$, and these byte values are derived from the (M, M') pair and thus known. For the latter case where a_i and b_i are both nonzero, the x'_i may have only 2 (in most cases) or 4 possible values. In this way we learn about the internal state, represented by the substitution triple (x, y, z). We can then take the other two equations from Eq. (1) and analyse these in a similar way to learn about the unknown y, z. The idea of using a differential trail for the state recovery is not new, see e.g. [HII$^+$22].

Repeat the atomic step several times. In SMAC-1, the optimal byte trail involves at least 20 active Sboxes, which means that from a single atomic step we learn around 20 bytes of the internal state. We can repeat the atomic step by using a different byte trail with a different $\mu(\Delta M)$ (for the same time instance t in all repeated atomic steps, but may involve different Δt and distinct trails), and recover new bytes of the internal state. After 3-4 such atomic steps, the complete 48-byte internal state can be derived.

State collision detection. To detect a true state collision, and not a random tag collision, we can append a single block to both M and M' and query the oracle whether the sequences $(M||1), (M||2)\dots$ and $(M'||1), (M'||2)\dots$ still produce the same tag. In case of a random tag collision, this will not be the case.

Complexity to get a related pair (M, M'). Assume for the moment that we can make queries to the sender oracle (although this scenario is not realistic since in that case the attacker already has access to the universal oracle). In a naïve approach, we choose ΔM, pick a random M and derive $M' = M + \Delta M$, then call the oracle to get the sequence of tags, and thereby determine whether it is a state collision or not. If the success probability of the state collision for the chosen ΔM is 2^{-s}, then we need to make $O(2^s)$ queries.

However, that complexity may be improved as follows. We pick a byte-differential $\mu(\Delta M)$, then make around $O(N = \sqrt{2^s})$ queries to the oracle with different messages $M_1, \dots, M_N$, and get relevant tag-sequences $T_1, \dots, T_N$ (each sequence of tags should cumulatively have size at least s bits) for each message. Each message is constructed to follow the chosen byte-differential $\mu(\Delta M)$ such that if for any byte index i we have $\mu(\Delta M)[i] = 0$ then in every message that byte $M_k[i]$ is a constant value for all $k \in [1..N]$ (we can choose that constant byte at random); and when $\mu(\Delta M)[i] = 1$ then $M_k[i]$ is picked at random in every message independently. In the end, we would get $2^{s/2}$ pairs $\{k \in [1..N] : (M_k, T_k)\}$, and due to the birthday paradox there should be a pair (M_a, T_a) and (M_b, T_b) in the list such that $T_a = T_b$, meaning that we fall into the state collision. The actual bit-differential is $\Delta M = M_a + M_b$, and the probability that this concrete bit-differential value follows the chosen byte-differential $\mu(\Delta M)$ is high (we skip further details of this point).

In order to find a related pair of messages among N collected, we can sort the list after the tag-sequences in time $O(N \log N)$, then find a matching $T_a = T_b$ in linear time. The sorting complexity can be decreased by also using hash tables, thus we should conservatively regard this step as the minimum $O(2^{s/2})$.

The overall complexity of the state recovery attack by using nonce-misuse queries to the sender oracle is at least $O(2^{s/2})$ queries, or more. If the attacker can only use a verification oracle, then the complexity is at least $O(2^{s/2+\tau})$. For SMAC-1 and SMAC-3/4, the minimum values for s are 118 and 152, respectively, thus the absolute lower bounds for this attack is at least $O(2^{59+\tau})$ and $O(2^{76+\tau})$ queries. These are very conservative estimates.

Note also that this attack does not lead to a key recovery, and the universal forgery can create messages only for a certain pair of (K, IV). Also, the above "birthday paradox" improvement may not work if the space of valid bit-differentials ΔM that follow the selected byte-differential $\mu(\Delta M)$ contains additional constraints on the bit-trails (e.g., not all ΔM are possible for the state collision to happen), thus the attack complexity may actually be much larger, up to $O(2^{s+\tau})$ queries. We leave this study as an open question to refine the nonce-misuse attack complexity in the future.

3.13 Other considerations and overall design justification

Yet another attack approach would be to perform some algebraic analysis. For example, in a nonce misuse scenario where several truncated correct tags are given for related messages, one may try to target the finalisation. But, with a very high degree of output Boolean functions (Table 7) that seems not feasible.

Adding the last block with the AAD and ciphertext lengths makes it harder to perform attacks based on insertions or deletions of message blocks and it might also be useful for reasons beyond security.

To conclude this section, we have demonstrated that every design choice in the SMAC framework has justified reasoning, and concrete analyses in various security models, attack scenarios, and simulations support these.

4 Software evaluation

SMAC compression function can be implemented with only a few SIMD instructions on modern CPUs, and our assessment is that SMAC is fast and competitive design in both software and hardware.

```
void SMAC_Compress(__m128i& A1, __m128i& A2, __m128i& A3, __m128i* msg)
{       __m128i M = msg ? *msg : const1;  // if msg=NULL then dummy clock
        __m128i T = Sigma(Xor3(A2, A3, M));
        A3 = AesRound(A2, M);
        A2 = AesRound(A1, M);
        A1 = T;
}
```

Listing 1. SMAC compression function (implementation sketch).

Performance comparison (in Gbps)	Length of the Message, in bytes					
of Algorithms...	65536	16384	4096	1024	256	64
Platform P1: Intel Core i5-1145G7 @2.6/4.4GHz						
GHASH (OpenSSL 3.3.0-dev)	212	202	153	81	78	22
CMAC-AES-128 (OpenSSL 3.3.0-dev)	16.7	16.6	16.1	13.9	9.3	3.6
HMAC-SHA3-256 (OpenSSL 3.3.0-dev)	3.9	3.8	3.6	2.9	1.7	0.6
Poly1305 (estimate)	[36, 92]					
SMAC-1 (C++, SIMD)	150	148	138	112	64	18.7
SMAC-3/4 (C++, SIMD)	111	110	105	90	56	18.7
Platform P2: AMD EPYC 9554P @3.1/3.75GHz						
GHASH (OpenSSL 3.3.0-dev)	191	178	138	72	67	21
CMAC-AES128 (OpenSSL 3.3.0-dev)	11.6	11.6	11.3	10.2	7.4	3.6
Poly1305 (estimate)	[27, 71]					
SMAC-1 (C++, SIMD)	124	124	114	90	49	14.5
SMAC-3/4 (C++, SIMD)	93	92	88	72	43	14

Table 8. Software performance evaluation results.

We, however, made a slightly optimised implementation of SMAC-1 and -3/4 leveraging SIMD instructions, partial unrolling, a better utilisation of registers, etc. All measurements were performed in a single threaded setup, averaging over 3 seconds. The results of our performance measurements are given in Table 8.

SMAC vs. GHASH. From the performance measurements we see that for large messages GHASH can outperform SMAC (212Gbps vs. 150Gbps on P1), which can be explained by parallelisation techniques of GHASH (e.g., in boundary cases SMAC is even faster than GHASH, see the results for 1024 bytes). However, we believe that a form of Accordion mode discussed by NIST [NIS24b] may be used if an even faster performance will be needed.

SMAC vs. Poly1305. Our current build of the latest OpenSSL-3 does not have an isolated speed measurement of MACing based on Poly1305, but we can roughly estimate the performance of Poly1305 from two other measurements of the algorithms `chacha20-poly1305` (36Gbps, 2^{16} bytes on P1) and `chacha20` (59Gbps). It is easy to derive that a stand-alone Poly1305 has the performance *at most* $1/(1/36 - 1/59) \approx 92$Gbps on P1. It is also most likely that `chacha20-poly1305` benefits from the instructions iterleaving effect in CPU (like AES in GCM mode), therefore, the performance of a stand-alone Poly1305 is actually somewhere in between 36-92Gbps, i.e., Poly1305 is significantly slower than SMAC. Similar measurements on P2 for 2^{16} bytes message are: 27.2Gbps for `chacha20-poly1305`, 44.2Gbps for `chacha20`, resulting in the performance of Poly1305 in between 27-71Gbps.

5 Conclusions

In this paper, we presented a new efficient stand-alone MAC scheme based on the processing in the FSM part of the stream cipher family SNOW. The proposal offers a combination of very high speed in software and hardware, a truncatable tag and resistance to nonce misuse. Two concrete versions of SMAC are proposed with different security levels. SMAC can be combined with an encryption scheme in an AEAD mode, with high performance and robust security. Every design choice has been argued for through analysis and simulations.

A direction for future work could be to examine the possibility of meaningful security proofs for the construct. For example, one might investigate to what extent the `InitFinal` algorithm with $d = 9$ is indistinguishable from a PRF. If so, this might be extended to proofs for the full construct.

Acknowledgements

We thank the cloud teams of E2C and LUNARC for their patience and help in computing resources for our simulations that made these results possible. We also thank John Preuß Mattsson and Erik Thormarker for providing constructive comments.

This work was supported by the Swedish Foundation for Strategic Research (Grants No. RIT17-0005 and SM22-0050) and the ELLIIT program.

Appendix A: Permutation candidates

SMAC-1. Minimum number of active Sboxes of a differential trail $(\Delta M^t, ..., \Delta M^{t+\Delta t-1})$ where the first and the last Δs are nonzero.							If used in FSM, min. number of active Sboxes
$\Delta t = 3$	$\Delta t = 4$	$\Delta t = 5$	$\Delta t = 6$	$\Delta t = 7$	$\Delta t = 8$	$\Delta t = 9$	
22	20	20	20	21	[22,23]	[23,25]	18
σ_1 ={0,7,14,11,4,13,10,1,8,15,6,3,12,5,2,9}					σ_2 ={0,9,6,13,4,11,2,15,8,1,14,5,12,3,10,7}		
20	20	19	19	19	20	≥22	18
σ_3 ={4,9,2,13,0,11,6,15,12,1,14,5,8,3,10,7}					σ_4 ={4,9,6,13,0,11,2,15,12,1,10,5,8,3,14,7}		
σ_5 ={8,1,6,13,12,11,2,15,0,9,14,5,4,3,10,7}					σ_6 ={8,7,14,3,12,13,10,1,0,15,6,11,4,5,2,9}		
σ_7 ={8,7,14,11,12,5,10,1,0,15,6,3,4,13,2,9}					σ_8 ={8,9,6,13,12,11,2,7,0,1,14,5,4,3,10,15}		
σ_9 ={12,7,2,11,8,13,10,1,4,15,6,3,0,5,14,9}					σ_{10} ={12,7,14,11,8,13,6,1,4,15,10,3,0,5,2,9}		
24	20	19	20	22	24	≥25	17
σ_{11} ={7,10,5,8,11,14,9,12,15,2,13,0,3,6,1,4}					σ_{12} ={13,8,15,10,1,12,3,14,5,0,7,2,9,4,11,6}		
20	20	19	19	19	20	22	16
σ_{13} ={4,1,14,11,0,13,10,7,12,15,6,3,8,5,2,9}					σ_{14} ={4,7,14,11,0,13,10,1,12,9,6,3,8,5,2,15}		
σ_{15} ={12,9,6,3,8,11,2,15,4,1,14,5,0,13,10,7}					σ_{16} ={12,9,6,13,8,5,2,15,4,1,14,11,0,3,10,7}		
24	19	19	20	22	24	≥25	15
σ_{17} ={6,5,15,12,13,9,8,14,3,2,4,7,10,0,1,11}					σ_{18} ={9,5,4,10,15,14,0,3,6,12,13,7,2,1,11,8}		
σ_{19} ={11,10,12,15,2,8,9,3,14,13,7,4,5,1,0,6}					σ_{20} ={14,4,5,15,10,9,3,0,1,13,12,2,7,6,8,11}		

SMAC-3/4. Minimum number of active Sboxes of a differential trail $(\Delta M^t, ..., \Delta M^{t+\Delta t-1})$ where the first and the last Δs are nonzero, encountering cases with dummy middle clock(s), i.e. where $\Delta M^{t+k} = 0$.							If used in FSM, minimum number of active Sboxes
$\Delta t = 3$	$\Delta t = 4$ $k = 1/2$	$\Delta t = 5$ $k = 1/2/3$	$\Delta t = 6$ $k = 2/3$	$\Delta t = 7$ $k = 1,5$ $/3$	$\Delta t = 8$ $k = 1,5$ $/2,6$	$\Delta t = 9$ $k = 1,5$ $/2,6/3,7$	
26	24/30	24/24/24	24/24	29/25	27/33	≥29/29/35	20
σ_{21} ={7,9,0,12,15,8,2,13,6,1,14,11,5,3,10,4}				σ_{22} ={9,7,14,8,11,13,4,0,3,12,6,1,10,5,2,15}			
σ_{23} ={11,4,14,9,2,13,10,7,1,15,6,0,3,5,12,8}				σ_{24} ={14,9,6,3,13,11,2,12,15,1,8,4,7,0,10,5}			
26	26/32	24/26/26	26/≥24	32/25	30/≥37	30/35/≥35	18
σ_{25} ={4,2,10,11,0,13,14,6,12,7,9,5,8,3,15,1}				σ_{26} ={4,15,1,13,0,11,7,9,12,10,2,3,8,5,6,14}			
σ_{27} ={8,3,6,5,12,7,10,9,0,11,14,13,4,15,2,1}				σ_{28} ={8,15,14,1,12,3,2,5,0,7,6,9,4,11,10,13}			
σ_{29} ={12,7,3,5,8,6,14,15,4,1,2,10,0,11,13,9}				σ_{30} ={12,9,10,2,8,3,5,1,4,15,11,13,0,14,6,7}			
26	35/30	24/≥25/24	≥24/24	30/25	28/35	31/34/≥35	17
σ_{31} ={4,9,11,6,8,13,15,10,12,1,3,14,0,5,7,2}				σ_{32} ={4,13,14,7,12,11,15,10,0,6,1,5,8,9,2,3}			
σ_{33} ={8,7,11,6,12,2,13,1,4,5,14,15,0,9,10,3}				σ_{34} ={8,14,9,13,0,1,10,11,12,5,6,15,4,3,7,2}			
σ_{35} ={12,13,6,7,8,1,2,11,0,15,3,14,4,10,5,9}				σ_{36} ={12,14,9,11,0,2,13,15,4,6,1,3,8,10,5,7}			
26	26/28	24/25/25	27/24	31/24	27/37	30/35/≥34	17
σ_{37} ={4,7,12,15,8,13,10,9,0,1,3,14,2,6,5,11}				σ_{38} ={4,8,6,10,12,11,14,15,1,5,13,7,0,9,2,3}			
σ_{39} ={4,9,6,5,12,13,15,10,14,2,1,7,0,3,8,11}				σ_{40} ={4,13,6,7,8,12,10,14,0,15,2,3,5,9,1,11}			
σ_{41} ={6,10,9,15,8,11,0,3,12,1,14,13,4,5,7,2}				σ_{42} ={7,14,15,10,12,13,3,0,4,6,1,5,8,11,2,9}			
σ_{43} ={8,7,10,11,13,1,9,3,12,5,14,15,0,4,2,6}				σ_{44} ={8,9,10,13,0,14,2,12,4,5,6,15,7,1,11,3}			
σ_{45} ={8,9,11,6,10,14,13,3,12,15,4,7,0,5,2,1}				σ_{46} ={8,9,15,12,0,2,13,1,4,7,14,5,3,10,11,6}			
σ_{47} ={8,14,9,11,0,3,2,15,4,13,12,5,10,1,7,6}				σ_{48} ={9,13,5,15,8,1,10,11,12,0,14,2,4,3,6,7}			
σ_{49} ={11,5,15,7,12,13,14,1,4,2,6,0,8,9,10,3}				σ_{50} ={12,5,4,13,2,9,15,14,0,6,1,3,8,11,10,7}			
σ_{51} ={12,10,14,8,0,1,2,11,3,13,7,15,4,5,6,9}				σ_{52} ={12,13,14,7,15,9,3,11,0,1,2,5,8,6,10,4}			
σ_{53} ={12,14,9,13,0,3,10,1,15,6,7,2,4,5,11,8}				σ_{54} ={12,15,6,13,11,2,3,14,0,1,7,4,8,10,5,9}			
σ_{55} ={12,15,14,11,0,9,8,1,6,13,3,2,4,10,5,7}				σ_{56} ={14,5,11,10,12,2,13,15,4,7,6,3,8,1,0,9}			
26	26/26	24/24/25	24/24	30/26	29/32	31/31/35	16
σ_{57} ={4,15,5,11,14,1,8,2,12,7,13,6,9,3,0,10}				σ_{58} ={4,15,5,14,1,11,8,2,12,7,13,3,6,9,0,10}			
σ_{59} ={10,13,4,14,8,3,9,2,5,15,12,6,0,11,1,7}				σ_{60} ={13,7,4,14,8,3,9,15,2,5,12,6,0,11,1,10}			

Table 9. Strong permutation candidates found for SMAC-1 and SMAC-3/4.

Appendix B: Cluster characteristics of selected permutations for SMAC-3/4

Case		A few selected permutations from Table 9 for SMAC-3/4								
		σ_{21}	σ_{25}	σ_{27}	σ_{31}	σ_{32}	σ_{36}	σ_{37}	σ_{42}	σ_{57}
$\Delta t = 3$	b	39	44	44	8	8	8	12	12	38
	c	24	28	28	8	8	8	12	12	32
	p	$2^{-159.67}$	2^{-162}	$2^{-162.41}$	2^{-163}	2^{-163}	2^{-163}	2^{-162}	2^{-163}	$2^{-160.67}$
Sboxes	#	26	26	26	26	26	26	26	26	26
$\Delta t = 4$	b	1/2	4/4	4/4	236/4	236/4	236/4	4/4	4/4	4/5
	c	1/1	4/4	4/4	212/4	212/4	212/4	4/1	4/1	4/2
$k = 1$	p	2^{-151}	2^{-163}	2^{-164}	$2^{-218.41}$	$2^{-216.41}$	$2^{-218.41}$	2^{-162}	2^{-163}	2^{-161}
$k = 2$	p	$2^{-195.41}$	2^{-208}	2^{-208}	2^{-194}	2^{-192}	2^{-193}	$2^{-178.19}$	$2^{-179.67}$	$2^{-167.83}$
Sboxes	#	24/30	26/32	26/32	35/30	35/30	35/30	26/28	26/28	26/26
$\Delta t = 5$	b	1158 /3/7	2583 /52/342	2616 /98/304	2888 /102/12	3172 /8/17	3272 /102/8	171 /5/4	155 /5/4	796 /14/203
	c	40/2/4	90/19/4	100/32/4	48/36/4	52/1/8	52/36/4	20/5/1	20/5/1	27/4/25
$k = 1$	p	$2^{-146.23}$	$2^{-145.57}$	$2^{-146.08}$	$2^{-139.75}$	$2^{-139.27}$	$2^{-137.83}$	$2^{-151.37}$	$2^{-152.29}$	$2^{-146.02}$
$k = 2$	p	$2^{-149.67}$	$2^{-164.91}$	$2^{-165.09}$	$2^{-165.67}$	$2^{-160.41}$	$2^{-164.67}$	2^{-164}	2^{-165}	$2^{-152.54}$
$k = 3$	p	$2^{-157.41}$	$2^{-159.06}$	$2^{-152.39}$	$2^{-156.54}$	2^{-156}	$2^{-156.83}$	$2^{-153.83}$	$2^{-154.83}$	$2^{-149.21}$
Sboxes	#	24/24/24	24/26/26	24/26/26	24/26/24	24/25/24	24/26/24	24/25/25	24/25/25	24/24/25
$\Delta t = 6$	b	4/191						–/316	–/264	1333/112
	c	1/12	—	—	—	—	—	–/45	–/35	14/6
$k = 2$	p	$2^{-159.83}$						–	–	$2^{-147.86}$
$k = 3$	p	$2^{-145.79}$						$2^{-151.21}$	$2^{-152.86}$	$2^{-149.04}$
Sboxes	#	24/24	26/25	26/24	24/24	24/24	25/24	27/24	27/24	24/24
$\Delta t = 7$	b							–/5*	–/6*	
	c	—	—	—	—	—	—	–/2	–/1	—
$k = 1,5$	p							–	–	
$k = 3$	p							$2^{-156.60}$	$2^{-156.83}$	
Sboxes	#	29/25	32/25	32/25	30/25	30/25	30/25	31/24	31/24	30/26
Max.	p	$2^{-145.79}$	$2^{-145.57}$	$2^{-146.08}$	$2^{-139.75}$	$2^{-139.27}$	$2^{-137.83}$	$\mathbf{2^{-151.21}}$	$\mathbf{2^{-152.29}}$	$2^{-146.02}$

* *Only significant trails that have at most 57 active Sboxes were enumerated.*

In this table, b is the number of byte-trails, c is the number of clusters, p is the probability of the cluster, and value(s) in k mean that $\Delta M^{t+k} = 0$ are dummy blocks.

Table 10. Cluster characteristics of differential trails of SMAC-3/4.

Appendix C: A fast heuristic algorithm to find a small guess base in guess-and-determine attack scenarios

The main idea on describing relations between variables comes from [HE22], but since Autoguess works very slow already for $d \geq 3$, we decided to develop a simplified yet powerful enough tool to solve GnD systems where all variables have the same weight (weight 1 to all byte variables in our case) and only the basic type of relation supported.

A relation on n variables $[x_0, x_1, \ldots, x_{n-1}]$ is added to the system when the knowledge of any $n-1$ variables results in the knowledge of the remaining unknown in the list. To note, a new relation and new variables are added to the system only at the points of branching. In our case, we have two such points.

Relations for `XOR`: Let us have a branching point such as $c = a \oplus b$, then the relation here is simply $[a, b, c]$, meaning that the knowledge of any 2 values (bytes) would result in the knowledge of the third value. Note that here a new variable c is introduced into the system.

Relations for `MixColumn`: Consider a 4-by-4 `MixColumn` operation from AES. The input is four existing variables (x_0, x_1, x_2, x_3), and the output are new variables (y_0, y_1, y_2, y_3) to be added to the system. The `MixColumn` linear transformation is such that knowing any 4 values from 8 input and output variables would result in the knowledge of all other 4 values. This can be described with 56 5-tuple relations (8 choose 5) such as $[x_0, x_1, x_2, x_3, y_0], \ldots, [y_0, y_1, y_2, y_3, x_3]$.

Application of Sboxes and permutations of the array of variables do not create any new relation nor introduce any new variable. This way, the complete system comprises a set of v variables (some of which can be set as known) and a set of r relations between these variables, and that system can be described by a *binary* matrix R of size $r \times v$, e.g.:

$$R_{r,v} = \begin{array}{cccc} x_0 & x_1 & \ldots & x_{v-1} \\ \hline 1 & 0 & \ldots & 1 \\ 1 & 1 & \ldots & 0 \\ \vdots & & & \\ 0 & 1 & \ldots & 1 \\ \hline \end{array}$$

where v columns represent variables and r rows are relations. Introduce the following v-bit vectors:

- K_v – the vector of variables that are known from the start, such as observed output bytes or those bytes where we insert IV, which are known, etc.
- G_v – the vector of the guess base, initialised as $G_v = \mathbf{0}$.

The target of the solver is to find G with the minimum Hamming weight such that given only variables from the set $(K \vee G)$ one can derive all other variables in time $O(1)$ by using the relation matrix R.

Let us now introduce a **knowledge propagation function:** $F_{KP}(R_{r\times v}, K_v) \rightarrow K'_v$, which is, given the relation matrix R and a vector of all known variables at the moment K, derives a new vector of known variables K' after applying the matrix with relations R. This function works as follows:

Algorithm 6 Knowledge propagation function.

```
1: function $F_{KP} : (R, K) \rightarrow K'$
2:     Set $K' = K$
3:     for all $i = 0, 1, \dots, r-1$ do
4:         if Hamming weight of $(\texttt{NOT}(K') \wedge R.\texttt{row}(i))$ is 1 then
5:             $K' = K' \vee R.\texttt{row}(i)$
6:         end if
7:     end for
8:     If at least one bit was added to $K'$ during the above for-loop, repeat that loop
   again until no more new bits can be added to $K'$.
9: end function
```

The algorithm of finding the guess base consists of two phases – the *Approximation* and *Reduction* phases, as briefly described below. The algorithm is a variation of a greedy approach, but comparing to Autogess it works extremely fast and still gives quite good results. Since it is still a heuristic algorithm, one should expect that the resulting guess base may not be optimal, but hopefully close to the minimum.

Approximation phase. We start by computing $Y = F_{KP}(R, K \vee G)$, and then also remove rows $R.\texttt{row}(i)$ from R where Hamming weight of $Y \wedge R.\texttt{row}(i)$ is zero – i.e., these relations become not helpful in the GnD flow.

Then, in each step of this phase we try all unknown variables one by one (those where Y is '0'), and collect **metrics** for each of these unknowns – we will talk about various metrics further. The unknown variable with the best metric is added to the guess base G, and Y is updated as $Y = F_{KP}(Y \vee x)$, while also removing rows from R that in this step became covered by Y.

The phase ends when the Hamming weight of Y becomes equal to v, i.e., all variables became known.

In an improved variant each step we test all possible pairs of unknown variables and the one with best metrics is added to the guess base G, and Y is updated with two points added. Testing a triple-point is more costly but still feasible time. However, we used the 3-points method only on few analysis cases.

Metrics. Let us pick one unknown variable x that is not in Y. We have identified two main metrics:

(a) the Hamming weight of $F_{KP}(R, Y \vee x)$ – the larger Hamming weight the more variables become known if that particular x is added to the guess base.
(b) the Hamming weight of the column of (the truncated) R corresponding to x – the more 1s are removed from R, the more new variables may be derived through the knowledge propagation.

There can be any order of (a) and (b) metrics for the decision which candidate for the base guess is better to adopt, and we have tried both orders in our simulations and finally took the shorted guess base from both methods. In case of a tie-break decision, we apply additional metrics:

(c) choose the best candidate between two equal options at random
(d) prioritise the candidate involving the unknown variable closer to other known variables – i.e., in case of SMAC analysis we prefer to avoid guessing variables somewhere in the middle of d rounds of `InitFinal`.

Reduction phase. After an approximate guess base is received, we then start the last phase of reduction of the base. We simply try to remove two guessed variables from the guess base, and see whether adding one other unknown would still give a valid guess base. In an improved variant, one may also remove 3 variables, try to add 1 or 2 other unknowns and check if the guess base is still valid. But this appeared to bee too timely and thus we did not use 3-points reduction.

Appendix D: Example trail

Clocks	Variables	Intermediate differences
0	$\Delta A1$	`00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00`
	$\Delta A2$	`00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00`
	$\Delta A3$	`00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00`
1	$\Delta S(A1)$	`00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00`
	$\Delta S(A2)$	`00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00`
	ΔM^t	`80,00,00,00,00,00,59,00,00,0e,00,00,00,84,00,00`
	$\Delta A1$	`80,00,00,00,00,84,00,00,00,00,59,00,00,00,00,0e`
	$\Delta A2$	`80,00,00,00,00,00,59,00,00,0e,00,00,00,84,00,00`
	$\Delta A3$	`80,00,00,00,00,00,59,00,00,0e,00,00,00,84,00,00`
2	$\Delta S(A1)$	`39,00,00,00,00,a8,00,00,00,00,01,00,00,00,00,71`
	$\Delta S(A2)$	`11,00,00,00,00,00,64,00,00,40,00,00,00,01,00,00`
	ΔM^{t+1}	`ed,00,00,00,00,44,40,00,00,32,24,00,00,ac,00,10`
	$\Delta A1$	`ed,00,00,00,00,ac,24,00,00,10,40,00,00,44,00,32`
	$\Delta A2$	`0c,00,00,00,00,44,40,00,00,32,24,00,00,ac,00,10`
	$\Delta A3$	`cf,11,11,33,c0,c4,00,40,03,30,25,01,64,00,c8,74`
3	$\Delta S(A1)$	`28,00,00,00,00,a8,64,00,00,40,01,00,00,01,00,71`
	$\Delta S(A2)$	`28,00,00,00,00,a8,64,00,00,40,01,00,00,01,00,71`
	ΔM^{t+2}	`c3,11,11,33,c0,80,40,40,03,02,01,01,64,ac,c8,64`
	$\Delta A1$	`00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00`
	$\Delta A2$	`00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00`
	$\Delta A3$	`00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00`

Table 11. An example trail for SMAC-1 in $\Delta t = 3$.

Appendix E: Reference implementation (C/C++, SIMD)

```
#define IS_SMAC1 1  /* Select the instance: SMAC-1 or SMAC-3/4 */
#define SIGMA (IS_SMAC1\
        ? _mm_setr_epi8(0,7,14,11,4,13,10,1,8,15,6,3,12,5,2,9)\
        : _mm_setr_epi8(7,14,15,10,12,13,3,0,4,6,1,5,8,11,2,9))

#define load(ptr)      _mm_loadu_si128((__m128i*)(ptr))
#define store(ptr, x)  _mm_storeu_si128((__m128i*)(ptr), x)
#define aes(a, k)      _mm_aesenc_si128(a, k)
#define sigma(x)       _mm_shuffle_epi8(x, SIGMA)
#define xor2(x, y)     _mm_xor_si128(x, y)
#define xor3(x, y, z)   xor2(xor2(x,y),z)

void SMAC_Compress(__m128i& A1, __m128i& A2, __m128i& A3, uint8_t* msg)
{       __m128i M = msg ? load(msg) : _mm_cvtsi32_si128(1);
        __m128i T = sigma(xor3(A2, A3, M));
        A3 = aes(A2, M);
        A2 = aes(A1, M);
        A1 = T;
}

void SMAC_InitFinal(__m128i& A1, __m128i& A2, __m128i& A3)
{       __m128i T1 = A1, T2 = A2, T3 = A3;
        for (int i = 0; i < 9; i++)
                SMAC_Compress(A1, A2, A3, NULL);
        A1 = xor2(A1, T1);
        A2 = xor2(A2, T2);
        A3 = xor2(A3, T3);
}

// (!) In this implementation, aad/ct must reserve 16/32 extra bytes, resp.
void SMAC(uint8_t key[32], uint8_t iv[16], uint8_t* aad, int aad_sz, uint8_t
     * ct, int ct_sz, uint8_t * tag, int tag_sz)
{       // initialise with the key and iv
        __m128i A1 = load(key + 16), A2 = load(key), A3 = load(iv);
        SMAC_InitFinal(A1, A2, A3);

        // zeroise ending unaligned bytes, and add LEN-block to ct
        memset(aad + aad_sz, 0, 16);
        memset(ct + ct_sz, 0, 16);
        int aad_blocks = (aad_sz + 15) >> 4;
        int ct_blocks = (ct_sz + 15) >> 4;
        *(uint64_t*)(ct + (ct_blocks * 16) + 0) = aad_sz * 8;
        *(uint64_t*)(ct + (ct_blocks * 16) + 8) = ct_sz * 8;

        // compress full blocks, including the ending LEN-block
        for (int i = 0; i <= (aad_blocks + ct_blocks); i++)
        {       uint8_t* msg = i < aad_blocks ? (aad + i * 16)
                        : (ct + (i - aad_blocks) * 16);
                SMAC_Compress(A1, A2, A3, msg);
                if (!IS_SMAC1 && (i % 3) == 2)
                        SMAC_Compress(A1, A2, A3, NULL);
        }

        // finalise and derive the MAC value
        SMAC_InitFinal(A1, A2, A3);
        memcpy(tag, (uint8_t*)&A2, (tag_sz <= 16 ? tag_sz : 16));
        if (tag_sz > 16)
                memcpy(tag + 16, (uint8_t*)&A3, tag_sz - 16);
}
```

Listing 2. Reference implementation of SMAC-1 and -3/4 in C/C++ (Little endian).

Appendix F: Test vectors

The MAC tag is taken as $(A2||A3)_\tau$ after the finalisation phase.

```
=== TEST 1 ===
       KEY = { 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
               00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 }
        IV = { 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 }
       AAD = { }
    CIPHER = { }
For SMAC-1:
    After initialisation:
        A1 = { fa 4e 8b ba 5b a3 79 be 90 a7 ee d8 00 12 03 5b }
        A2 = { 8c 99 e7 01 95 ba 79 b6 e1 3f 0f 56 6a d4 5c 60 }
        A3 = { 59 ec 45 58 1d a5 08 9e e4 ad 8e 4d e2 da b1 08 }
    After compression (1 clock):
        A1 = { d5 28 ed 1b 88 0e 81 75 05 68 71 59 88 1f a2 92 }
        A2 = { 55 78 3c 27 19 f8 94 9f 13 00 3a 13 60 9d 98 fe }
        A3 = { 69 dd 17 95 fd 62 4f b9 e9 81 51 53 2a b5 53 27 }
    After finalisation:
        A1 = { aa 8c 58 31 e0 ce 87 91 08 b7 c2 63 1e 2e 9b f9 }
        A2 = { d8 2c 49 ea 46 81 ca 1f ba 97 93 49 5f 9a 60 85 }
        A3 = { 39 ce be 86 12 c8 0f 70 60 cf 18 41 2e 98 92 ee }
For SMAC-3/4:
    After initialisation:
        A1 = { 10 34 48 ab 43 0d ac c5 e1 b8 38 03 ed 27 fe 80 }
        A2 = { 7c 90 c3 d8 c9 55 eb 3a 83 98 a1 5f 92 30 fb 56 }
        A3 = { ae 25 80 ee 48 1a bd 5e b7 73 2f 62 a9 a6 ed 37 }
    After compression (1 clock):
        A1 = { 64 16 61 8e 3b 96 36 d2 81 56 b5 4f 34 3d 43 eb }
        A2 = { 27 bb 5f 14 59 76 bd 3d 50 2b 61 da 68 b6 f9 80 }
        A3 = { bc 14 40 87 05 21 26 f7 61 16 2f 1e 18 60 ac dd }
    After finalisation:
        A1 = { df a6 f5 9c 06 06 36 cf b5 85 9d 4c a5 ca bc f7 }
        A2 = { 66 49 62 35 b1 7d 4c 42 2c ce 5f 42 9d 45 6c 91 }
        A3 = { 3f 41 13 bc 6d 27 65 ac bb 5e 83 72 ca 99 41 f1 }

=== TEST 2 ===
       KEY = { 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
               00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 }
        IV = { 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 }
       AAD = { 03 }
    CIPHER = { }
For SMAC-1:
    After initialisation:
        A1 = { 42 e3 c1 df bd 96 9f a1 04 02 9a 30 c4 93 fa 26 }
        A2 = { 3a fc c4 b7 10 45 50 5e b1 d1 09 49 f9 d6 de 13 }
        A3 = { 25 c5 ff c2 ab 2b 5b 15 62 43 fc f0 e5 6a be 33 }
    After compression (2 clocks):
        A1 = { 61 dc 6e d4 a7 60 66 11 04 c3 c0 a7 5e 45 be a8 }
        A2 = { e0 64 7e 6f b9 f4 78 dc b4 3a 74 c1 96 4d 44 cb }
        A3 = { 48 34 ed 24 58 af a3 e2 9d 2e 4c ac 5b 10 07 52 }
    After finalisation:
        A1 = { 13 0e 94 2c 5b 1f 89 23 5e c6 9a c0 77 f6 9c 91 }
        A2 = { a1 35 23 df 28 37 ed d8 0f 6b 56 aa 61 17 80 b3 }
        A3 = { 8a 7b 4b e4 8f 4b 4b de b7 d5 af 8c 82 6d 81 6d }
For SMAC-3/4:
    After initialisation:
        A1 = { a3 1a 8c d8 b9 c6 d7 24 d4 9b 5b 75 ff 67 41 64 }
        A2 = { 5f db ff 2f c9 aa f4 3e 32 ef f5 a9 ff 07 42 33 }
        A3 = { 1e eb df 0b eb e4 70 6b b8 3f f6 da cf 73 cf 24 }
    After compression (2 clocks):
        A1 = { 0a b0 36 58 d2 b0 88 ee 90 99 0f 98 e0 9c e3 f9 }
        A2 = { 32 bf f6 69 25 03 a3 12 6b b1 93 89 02 b1 3e b7 }
        A3 = { 39 57 c5 65 51 50 6a a6 c6 b8 8c 8f ea 46 eb 84 }
    After finalisation:
        A1 = { f2 33 7e b0 54 87 37 5b 6e f6 f3 64 67 07 93 80 }
        A2 = { 39 bf fe 0e 2c 33 11 f7 51 69 8e 64 d0 4e 52 70 }
        A3 = { c0 99 5e 83 54 a5 a8 22 57 94 06 c0 49 f2 0a 6f }

=== TEST 3 ===
       KEY = { b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
               c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf }
        IV = { d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df }
       AAD = { e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec dd de ef }
    CIPHER = { f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff }
For SMAC-1:
    After initialisation:
        A1 = { db 1d 65 de 28 12 23 17 15 8d ab 00 04 5f 22 5c }
        A2 = { e1 11 bc 2b c5 47 d0 19 15 83 6f 95 1f 47 5f 84 }
        A3 = { 00 91 7c c9 66 f0 f3 84 07 b2 6d 48 cf 7c 06 f8 }
    After compression (3 clocks):
        A1 = { 0f 0f 18 4b 4a 3a 25 0a ef b3 82 01 ce 2c 59 9c }
        A2 = { 0f 1a 78 a3 a9 a9 00 63 e8 21 f0 ed 82 52 80 12 }
        A3 = { 56 b8 b1 7f 40 bf a9 16 e9 5a 19 9f dd b9 98 60 }
    After finalisation:
        A1 = { a9 d2 6c f8 c3 75 b6 6f b5 28 d3 e2 80 75 b8 cc }
        A2 = { 61 3f ad 89 9e 94 51 48 1a eb d1 7a 5c 64 dd 18 }
        A3 = { 9a c4 ac 2e 18 74 a4 e1 cf 9b 42 92 15 38 a9 a1 }
For SMAC-3/4:
    After initialisation:
        A1 = { 30 b2 8e a9 d7 6b 44 d1 74 21 21 c5 68 43 45 62 }
        A2 = { 84 79 59 30 73 11 5b a2 bd 12 a3 85 66 66 43 20 }
        A3 = { c6 56 7a de ff 9f 3e fa a0 fb a4 6f f2 73 b8 d3 }
    After compression (4 clocks):
        A1 = { 13 5b 81 4d 81 50 f1 cf 5a cf 7b cf e5 1e b0 7c }
        A2 = { 72 13 2e cf 8b 8a f1 54 0c f2 8b 27 c4 66 b8 0d }
        A3 = { 75 3b de a8 94 36 d3 da 52 49 e6 17 8c 92 78 7c }
    After finalisation:
        A1 = { 98 d5 fe f2 0c e2 c7 4d 74 2a ed b1 25 81 3e da }
        A2 = { db 13 1c b3 ff bc a2 ed ae a4 78 93 58 18 67 5a }
        A3 = { 6b b8 f5 a9 83 7b c5 9f 4d 45 fd a7 60 31 cf 53 }

=== TEST 4 ===
       KEY = { 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
               10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f }
        IV = { ff fe fd fc fb fa f9 f8 f7 f6 f5 f4 f3 f2 f1 f0 }
       AAD = { 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10
               11 12 13 }
    CIPHER = { 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 }
For SMAC-1:
    After initialisation:
        A1 = { 5c f1 48 92 aa 70 1c 6c 0f 7b 8d 57 96 0c 39 4b }
        A2 = { ee 31 08 25 85 30 fc 59 8e a6 c3 ec 57 2f dc 59 }
        A3 = { 45 cc 6d 77 d2 56 28 a9 be 38 5a 78 4b a1 ba 14 }
    After compression (4 clocks):
        A1 = { 50 62 2a 64 fc 70 1b 1d 8e 6d 9f 12 dc f5 b7 7c }
        A2 = { d8 9d 7b 14 68 94 59 74 51 74 60 c7 56 d2 16 3f }
        A3 = { 45 73 4e 18 8e d3 4d ae 78 31 d3 59 6b fa 47 c7 }
    After finalisation:
        A1 = { 82 98 b1 ab 90 54 76 e4 24 76 b3 78 d6 14 e8 08 }
        A2 = { c3 44 52 16 99 48 2d 93 28 3c 03 ec 7c 3d b8 b5 }
        A3 = { c7 77 64 62 16 89 98 ee 28 03 06 f9 25 33 09 7c }
For SMAC-3/4:
    After initialisation:
        A1 = { b7 1a 78 eb a6 e1 a2 02 6f 0b 87 2d f3 82 29 93 }
        A2 = { 06 46 fe a4 94 d8 20 18 e3 3d 52 b3 bd b7 19 5e }
        A3 = { 34 55 a2 94 e7 11 e2 10 cc b8 89 fb c9 98 29 6d }
    After compression (5 clocks):
        A1 = { fa 9e 30 f3 39 72 e3 0b c3 57 f3 49 1f 76 cc c3 }
        A2 = { cb db bb df 38 4f 34 f1 ef 48 fd 7f d3 1f 7d a7 }
        A3 = { e9 cd ed 82 6b eb 7e e2 20 db 2f df 34 bf 8e 55 }
    After finalisation:
        A1 = { 84 9c ca a1 1b 55 64 ba 15 72 b2 b9 0d 73 ba d3 }
        A2 = { 69 6e d0 a9 9e 04 84 3a 59 6d a5 b6 25 7d db de }
        A3 = { 65 6d 19 04 1d bb 04 58 35 c3 42 3b c4 92 61 4f }
```

Listing 3. Test vectors.

References

3GP. 3GPP. 3GPP confidentiality and integrity algorithms. `https://www.3gpp.org/specifications-technologies/specifications-by-series/confidentiality-algorithms`.

BCK96. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15, Santa Barbara, CA, USA, August 18–22, 1996. Springer, Heidelberg, Germany.

BDF11. Charles Bouillaguet, Patrick Derbez, and Pierre-Alain Fouque. Automatic search of attacks on round-reduced AES and applications. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 169–187, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany. `https://dx.doi.org/10.1007/978-3-642-22792-9_10`.

BdSF$^+$22. Alex Biryukov, Luan Cardoso dos Santos, Daniel Feher, Vesselin Velichkov, and Giuseppe Vitto. Automated truncation of differential trails and trail clustering in ARX. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021: 28th Annual International Workshop on Selected Areas in Cryptography*, volume 13203 of *Lecture Notes in Computer Science*, pages 286–307, Virtual Event, September 29 – October 1, 2022. Springer, Heidelberg, Germany.

Ber05. Daniel J. Bernstein. The poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49, Paris, France, February 21–23, 2005. Springer, Heidelberg, Germany.

BHK$^+$99. John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and secure message authentication. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 216–233, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany.

Bir07. Alex Biryukov. The design of a stream cipher lex. In *Selected Areas in Cryptography: 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers 13*, pages 67–75. Springer, 2007.

BJKS94. Jürgen Bierbrauer, Thomas Johansson, Gregory Kabatianskii, and Ben Smeets. On families of hash functions via geometric codes and concatenation. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 331–342, Santa Barbara, CA, USA, August 22–26, 1994. Springer, Heidelberg, Germany.

BÖS11. Joppe W. Bos, Onur Özen, and Martijn Stam. Efficient hashing using the AES instruction set. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 507–522, Nara, Japan, September 28 – October 1, 2011. Springer, Heidelberg, Germany.

BRS02. John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

Cam23. Peter Campbell. Glevian and vigornian: Robust beyond-birthday aead modes. Cryptology ePrint Archive, Report 2023/1379, 2023. `https://eprint.iacr.org/2023/1379`.

CG16. Colin Chaigneau and Henri Gilbert. Is AEZ v4.1 sufficiently resilient against key-recovery attacks? *IACR Transactions on Symmetric Cryptology*, 2016(1):114–133, 2016. `https://tosc.iacr.org/index.php/ToSC/article/view/538`.

CMM23. Matthew Campagna, Alexander Maximov, and John PreußMattsson. Galois Counter Mode with Secure Short Tags (GCM-SST). IETF Datatracker, May 2023. `https://www.ietf.org/archive/id/draft-mattsson-cfrg-aes-gcm-sst-00.html`.

DEMS21a. Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon PRF, MAC, and short-input MAC. Cryptology ePrint Archive, Report 2021/1574, 2021. `https://eprint.iacr.org/2021/1574`.

DEMS21b. Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *Journal of Cryptology*, 34(3):33, July 2021.

DR05. Joan Daemen and Vincent Rijmen. A new MAC construction ALRED and a specific instance ALPHA-MAC. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 1–17, Paris, France, February 21–23, 2005. Springer, Heidelberg, Germany. `https://dx.doi.org/10.1007/11502760_1`.

Dwo07. Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, November 2007. `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf`.

EJMY19. Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new SNOW stream cipher called SNOW-V. *IACR Transactions on Symmetric Cryptology*, 2019(3):1–42, 2019.

Eri24. Ericsson Blog. Follow the journey to 6G, 2024. `https://www.ericsson.com/en/6g`.

FLS15. Thomas Fuhr, Gaëtan Leurent, and Valentin Suder. Collision attacks against CAESAR candidates - forgery and key-recovery against AEZ and Marble. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 510–532, Auckland, New Zealand, November 30 – December 3, 2015. Springer, Heidelberg, Germany.

GKM$^+$09. Praveen Gauravaram, Lars R Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S Thomsen. Grøstl-a sha-3 candidate. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2009.

HE22. Hosein Hadipour and Maria Eichlseder. Autoguess: A tool for finding guess-and-determine attacks and key bridges. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22: 20th International Conference on Applied Cryptography and Network Security*, volume 13269 of *Lecture Notes in*

Computer Science, pages 230–250, Rome, Italy, June 20–23, 2022. Springer, Heidelberg, Germany.

Hel80. Martin Hellman. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory*, 26(4):401–406, 1980.

HII$^+$22. Akinori Hosoyamada, Akiko Inoue, Ryoma Ito, Tetsu Iwata, Kazuhiko Minematsu, Ferdinand Sibleyras, and Yosuke Todo. Cryptanalysis of Rocca and Feasibility of Its Security Claim. *IACR Transactions on Symmetric Cryptology*, 2022(3):123–151, September 2022. `https://tosc.iacr.org/index.php/ToSC/article/view/9852`.

HK15. Matthias Hamann and Matthias Krause. Stream cipher operation modes with improved security against generic collision attacks. Cryptology ePrint Archive, Report 2015/757, 2015. `https://eprint.iacr.org/2015/757`.

HK18. Matthias Hamann and Matthias Krause. On stream ciphers with provable beyond-the-birthday-bound security against time-memory-data tradeoff attacks. *Cryptography and Communications*, 10(5):959–1012, 2018.

HKR15. Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. Robust authenticated-encryption AEZ and the problem that it solves. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 15–44, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

HP08. Helena Handschuh and Bart Preneel. Key-recovery attacks on universal hash function based MAC algorithms. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 144–161, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Heidelberg, Germany.

IET24. IETF CFRG Mail Archive. Comments on AES-GCM-SST, 2024. `https://mailarchive.ietf.org/arch/msg/cfrg/51ZYKcZQDKF2RkzRFtMcH4xCy6E/`.

IK03. Tetsu Iwata and Kaoru Kurosawa. OMAC: One-key CBC MAC. In Thomas Johansson, editor, *Fast Software Encryption – FSE 2003*, volume 2887 of *Lecture Notes in Computer Science*, pages 129–153, Lund, Sweden, February 24–26, 2003. Springer, Heidelberg, Germany.

Jou06. Antoine Joux. Authentication failures in nist version of gcm. *NIST Comment*, 3, 2006.

KVW04. Tadayoshi Kohno, John Viega, and Doug Whiting. CWC: A high-performance conventional authenticated encryption mode. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption – FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 408–426, New Delhi, India, February 5–7, 2004. Springer, Heidelberg, Germany.

LPS21. Gaëtan Leurent, Clara Pernot, and André Schrottenloher. Clustering effect in simon and simeck. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part I*, volume 13090 of *Lecture Notes in Computer Science*, pages 272–302, Singapore, December 6–10, 2021. Springer, Heidelberg, Germany.

MW16. John Mattsson and Magnus Westerlund. Authentication Key Recovery on Galois/Counter Mode (GCM). In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 16: 8th International Conference on Cryptology in Africa*, volume 9646 of *Lecture Notes in Computer Science*, pages 127–143, Fes, Morocco, April 13–15, 2016. Springer, Heidelberg, Germany.

NIS07. NIST. SP 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, 2007. `https://csrc.nist.gov/pubs/sp/800/38/d/final`.

NIS24a. NIST. NIST News. NIST to Revise Special Publication 800-38D — Galois/Counter Mode (GCM) and GMAC Block Cipher Modes, 2024. `https://csrc.nist.gov/News/2024/nist-to-revise-sp-80038d-gcm-and-gmac-modes`.

NIS24b. NIST. NIST Workshop on the Requirements for an Accordion Cipher Mode 2024, 2024. `https://csrc.nist.gov/Events/2024/accordion-cipher-mode-workshop-2024`.

oST01. National Institute of Standards and Technology. Advanced encryption standard. *NIST FIPS PUB 197*, 2001.

PC14. Gordon Procter and Carlos Cid. On weak keys and forgery attacks against polynomial-based MAC schemes. In Shiho Moriai, editor, *Fast Software Encryption – FSE 2013*, volume 8424 of *Lecture Notes in Computer Science*, pages 287–304, Singapore, March 11–13, 2014. Springer, Heidelberg, Germany.

PD. Laurent Perron and Frédéric Didier. CP-SAT (v9.9). `https://developers.google.com/optimization/cp/`.

SII23. Kosei Sakamoto, Ryoma Ito, and Takanori Isobe. Parallel SAT framework to find clustering of differential characteristics and its applications. In Claude Carlet, Kalikinkar Mandal, and Vincent Rijmen, editors, *Selected Areas in Cryptography - SAC 2023 - 30th International Conference, Fredericton, Canada, August 14-18, 2023, Revised Selected Papers*, volume 14201 of *Lecture Notes in Computer Science*, pages 409–428. Springer, 2023.

SLI05. JH. Song, J. Lee, and T. Iwata. The AES-CMAC Algorithm. IETF RFC4493, June 2005. `https://www.rfc-editor.org/rfc/rfc4493.html`.

SLN$^+$21. Kosei Sakamoto, Fukang Liu, Yuto Nakano, Shinsaku Kiyomoto, and Takanori Isobe. Rocca: An Efficient AES-based Encryption Scheme for Beyond 5G. *IACR Transactions on Symmetric Cryptology*, 2021(2):1–30, 2021.

SSW23. Yaobin Shen, François-Xavier Standaert, and Lei Wang. Forgery attacks on several beyond-birthday-bound secure MACs. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part III*, volume 14440 of *Lecture Notes in Computer Science*, pages 169–189, Guangzhou, China, December 4–8, 2023. Springer, Heidelberg, Germany.

Sta10. Paul Stankovski. Greedy distinguishers and nonrandomness detectors. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010 - 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15, 2010. Proceedings*, volume 6498 of *Lecture Notes in Computer Science*, pages 210–226. Springer, 2010.

TIHM17. Yosuke Todo, Takanori Isobe, Yonglin Hao, and Willi Meier. Cube attacks on non-blackbox polynomials based on division property. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 250–279, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

Tod15. Yosuke Todo. Structural evaluation by generalized integral property. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer*

Science, pages 287–314, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

VV18. Serge Vaudenay and Damian Vizár. Can caesar beat galois? - Robustness of CAESAR candidates against nonce reusing and high data complexity attacks. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*, volume 10892 of *Lecture Notes in Computer Science*, pages 476–494, Leuven, Belgium, July 2–4, 2018. Springer, Heidelberg, Germany.

WHT$^+$18. Qingju Wang, Yonglin Hao, Yosuke Todo, Chaoyun Li, Takanori Isobe, and Willi Meier. Improved division property based cube attacks exploiting algebraic properties of superpoly. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 275–305, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

WP14. Hongjun Wu and Bart Preneel. AEGIS: A fast authenticated encryption algorithm. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013: 20th Annual International Workshop on Selected Areas in Cryptography*, volume 8282 of *Lecture Notes in Computer Science*, pages 185–201, Burnaby, BC, Canada, August 14–16, 2014. Springer, Heidelberg, Germany.

XZBL16. Zejun Xiang, Wentao Zhang, Zhenzhen Bao, and Dongdai Lin. Applying MILP method to searching integral distinguishers based on division property for 6 lightweight block ciphers. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 648–678, Hanoi, Vietnam, December 4–8, 2016. Springer, Heidelberg, Germany.