What the Fork: Implementation Aspects of a Forkcipher

Antoon Purnal[1], Elena Andreeva[2], Arnab Roy[3], and Damian Vizár[4]

[1] imec-COSIC, KU Leuven, Belgium
antoon.purnal@esat.kuleuven.be
[2] Technical University of Denmark, Denmark
elean@dtu.dk
[3] University of Bristol, United Kingdom
arnab.roy@bristol.ac.uk
[4] CSEM, Switzerland
damian.vizar@csem.ch

Abstract. Lightweight cryptography refers to cryptographic designs that are heavily optimized to minimize resources, such as computational complexity, latency, energy/power consumption, hardware area, code size, and RAM, or to be very efficient in a particular application scenario, where the "conventional" cryptography would not suffice. Prompted by the growing demand for such designs, NIST launched the Lightweight Cryptography project which is supposed to identify and possibly standardize suitable lightweight authenticated encryption (AE) and hashing algorithms in a well established open competition framework. One of these submissions is ForkAE. ForkAE proposes a new primitive ForkSkinny and AE modes optimized for applications where very short messages dominate the communication. In this paper, we investigate multiple implementation/trade-off strategies for ForkAE, benchmark the synthesized hardware and compare it with several other lightweight AE primitives, and give performance and area estimates for the implementation of the ForkAE modes, as well as some selected competitors.

1 Introduction

Lightweight cryptography (LWC) is the general term used for cryptography tailored for resource-constrained devices and applications [23], where the computational complexity, latency, energy and/or power consumption, hardware area, code size, or RAM usage of "conventional" cryptography simply does not fit in the budget. For example, a lightweight blockcipher may be designed to have a tiny hardware implementation but be slower than average [16], or very fast but to require larger hardware area and power [15], or be especially suitable for hardware [14]. Viewed from another perspective, lightweight cryptography can be understood as a set of designs that occupy the more extreme axes in the design/trade-off space. Unlike the "conventional" cryptography, which usually aims to cover a wide variety of platforms and applications, LWC is targeting narrower classes of applications with (very) particular constraints [13]. The need for specialized and highly optimized cryptography is evidenced both by the massive growth of the application markets (such as the "Internet of Things") and by the recent NIST Lightweight Cryptography project, which attracted 56 candidate designs [24] in round one.

Short messages. An important class of LWC applications which is of interest for the design of authenticated encryption with associated data (AEAD or AE) are applications where *the majority of messages is of short length* (e.g., 8 bytes). This class covers a wide range of practical scenarios.

The Secure Onboard Communication in the automotive industry [4] are expected to handle short messages with stringent latency requirements. Critical communication and massive IoT domains of 5G will have to process frequent bursts of very short messages [1]. Narrowband IoT allows a minimum payload size of 16 bits [3, 2], which will dominate the communication in applications such as smart parking lots that need to transmit information encoded on a few bits (e.g., "free" or "occupied" status). Most medical implant devices, such as pacemakers, transmit messages of length at most 16 bytes to and from the device programmer. Advanced robotic prosthetics wirelessly transmit bursts of short messages with stringent latency requirements, as well as 1-byte temporal synchronization messages [5]. Wireless aircraft tyre pressure monitoring systems usually transmit payloads of ≤ 10 bytes [25].

AE for short messages. Most of the modern AE schemes (e.g., CCM [28], GCM [22], OCB [21] and of the CAESAR candidates [11]) are constructed as modes of operation for a low-level cryptographic primitive, such as a (tweakable) blockcipher or a cryptographic permutation. When processing a nonce-associated data (AD)-plaintext tuple, virtually every such AE scheme makes a few calls to the primitive that are in addition to and independent of $(a + m)$, the lengths of the associated data (AD) a and the message m. These additional calls serve different purposes, typically they perform a nonce-based setup, or a computation of a key-dependent ciphertext redundancy. Such fixed-cost computation is well amortized on long inputs. However, for short inputs where the message processing may entail as little as a single primitive call, the extra calls result in a significant overhead.

Recently, Andreeva et al. [6] proposed the new symmetric primitive *forkcipher*. When forkcipher is coupled together with the appropriate AEAD mode of operation it achieves optimal $(a+m)$ primitive calls for the shortest messages [6]. This is achieved at the cost of constructing an expanding forkcipher primitive and utilizing its inverse in decryption (else additional primitive calls are always incurred). More precisely, a forkcipher is a tweakable expanding primitive; it produces *two* redundant output blocks. This allows for building modes that have a zero fixed cost and are able to completely process the shortest messages with a single primitive call while still being able to process longer inputs, albeit somewhat less efficiently. The proposed forkcipher instance ForkSkinny is an iterated design that follows the TWEAKEY framework [20]. Roughly speaking, ForkSkinny is like the tweakable blockcipher Skinny [10] except that halfway through the encryption, its state is duplicated (or else *forked*), and each fork is further encrypted with independent round keys with a total computational complexity of ≈ 1.6 of Skinny. Intuitively, ForkSkinny modes should outperform any modes of Skinny for the shortest queries.

Implementing ForkSkinny. The authors of the ForkAE submission [7] give results for a preliminary hardware (HW) implementation. In this paper, we investigate further the HW implementation aspects of ForkSkinny and its AE modes. More specifically, we (1) explore the HW implementation strategies and trade-offs that are available for an iterated forkcipher, (2) benchmark the obtained implementations and compare them with other lightweight (tweakable) blockciphers and permutations submitted to the NIST LW competition, and (3) estimate the resource costs of ForkSkinny modes and compare them to those of other similar lightweight AE schemes.

Contributions. In this work we describe **several implementation strategies for the newly proposed ForkSkinny**. Those can be additionally generalized to any it-

erated forkcipher. Our optimizations are targeting: (1) Post-fork parallelism; we show how to exploit the almost-independent processing of the two ForkSkinny branches after the forking point. (2) Recomputation; for small area constraints, we describe an efficient rewind/restart mechanism for serialization of the forkcipher branching. (3) Unrolling: for low latency, we describe different unrolled forkcipher implementations.
We further **compare the performance and area of the synthesized implementation of ForkSkinny in its modes with a suitable and manageable subset of NIST LWC candidates:** Skinny-AEAD [9], Romulus [19] and Ascon [18]. We employ a methodology that allows us to swiftly and reproducibly compile a meaningful comparison. In that effort: (1) We use or provide a full implementation for each involved lower level primitive, and synthesize with a freely available technology library. (2) We give a fair comparison by estimating the overall implementation cost and performance of each mode under the same assumptions. Moreover, we provide **configurable ForkSkinny implementations in the public domain**[§]. Finally, we **identify promising future research directions and applications for the forkcipher** primitive.

2 ForkSkinny Specification

A forkcipher is a function $\mathsf{F} : \{0,1\}^{\kappa} \times \{0,1\}^{t} \times \{0,1\}^{n} \times \{0,1,\mathsf{b}\} \rightarrow \{0,1\}^{n} \cup \{0,1\}^{2n}$ which takes a tweakey $\in \{0,1\}^{t+\kappa}$, a message $\in \{0,1\}^{n}$ as and an output-switch as input and produces the "left", the "right" or "both" n-bit output blocks according to the output-switch. κ and t denotes the length (in bits) of the secret key and tweak respectively.

Andreeva et al. have recently proposed an instantiation of forkcipher called ForkSkinny. ForkSkinny is constructed following the iterate-fork-iterate (IFI) paradigm using the tweakable block cipher Skinny[10]. The outline of the ForkSkinny construction is depicted in Figure 1.

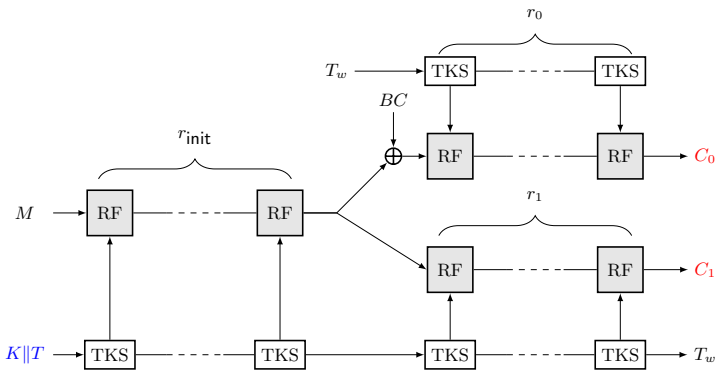

Fig. 1: The structure of ForkSkinny. TKS denote the round tweakey schedule function and RF denotes the round function (output-switch omitted)

The round function of ForkSkinny is almost identical to the round function of Skinny. Each round can be described as

$$\mathcal{R}_i = \texttt{Mixcolumn} \circ \texttt{Addconstants} \circ \texttt{Addroundtweakey} \circ \texttt{Shiftrow} \circ \texttt{Subcell}$$

[§]All current and future ForkAE implementations are available at https://github.com/byt3bit/forkae

where the `Mixcolumn`, `Shiftrow`, `Subcell` and `Addroundtweakey` functions are same as in SKINNY. Note that the `Addroundtweakey` function is used in FORKSKINNY to generate round tweakeys for $r_{\mathsf{init}} + r_0 + r_1$ rounds, where r_1 and r_0 denote the number of rounds in the left and right branch of FORKSKINNY and r_{init} denotes the number of rounds before forking. The `Addconstants` function in FORKSKINNY differs from SKINNY. Unlike SKINNY (which has 6 bit round constants), the `Addconstants` in FORKSKINNY generates 7 bit round constants using an LFSR. FORKSKINNY always have a key size $\kappa = 128$ (bits) and for each instance $r_0 = r_1$. We will denote an instance of FORKSKINNY as FORKSKINNY-n-$t + \kappa$-(r_{init}, r_0). Following this notation there instances of FORKSKINNY are: FORKSKINNY64-192-$(17, 23)$, FORKSKINNY-128-192-$(21, 27)$, FORKSKINNY-128-256-$(21, 27)$ and FORKSKINNY-128-288-$(25, 31)$. For a more detailed description of the FORKSKINNY algorithm we refer the readers to the article [6].

3 Forkcipher Modes

Andreeva et al. proposed two modes for ForkAE, the parallelizable mode PAEF and the sequential mode SAEF [6]. Both are provably secure nonce-based AE schemes [26] (we skip the syntax of AE schemes for brevity). The former achieves optimal quantitative security (thus allowing for secure instances with a small block size) while the latter is secure up to the birthday bound but requires a smaller internal state. Both these modes are designed to be most efficient for the shortest queries (with 1 or 2 blocks input data), and their performance deteriorates for longer inputs.

PAEF. PAEF processes blocks of AD and message with single call to F each, using tweaks composed of the nonce (with length $0 < \nu \leq t - 4$), a domain separation constant and a counter. See Figure 2.

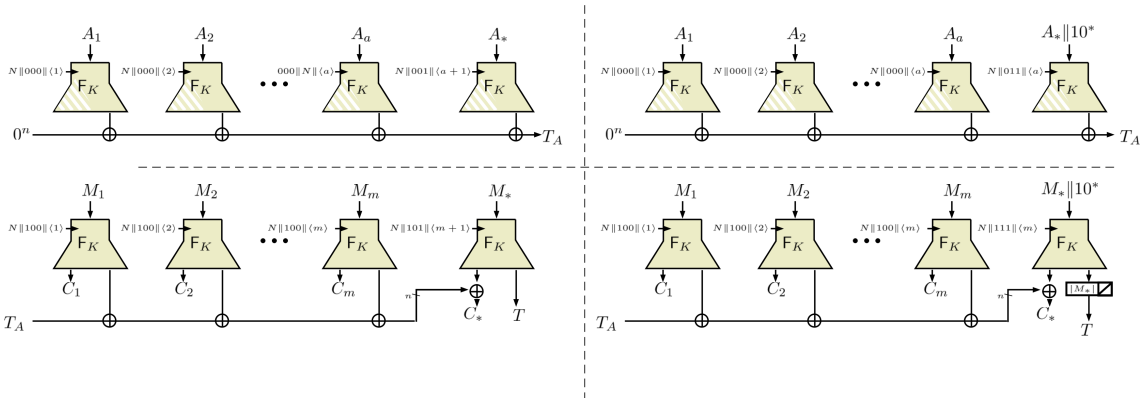

Fig. 2: The encryption algorithm of PAEF[F] mode. The picture illustrates the processing of AD when length of AD is a multiple of n (**top left**) and when the length of AD is not a multiple of n (**top right**), and the processing of the message when length of the message is a multiple of n (**bottom left**) and when the length of message is not a multiple of n (**bottom right**). The white hatching denotes that an output block is not computed. If $|M| = 0$, the tag is equal to T_A; else if $|A| = 0$, the AD processing is skipped.

SAEF. SAEF processes blocks of AD and message with single call to F each, using tweaks composed of the either a padded nonce (of length $t - 4$) or a string of $n - 3$ zeros, and a domain separation constant. See Figure 3.

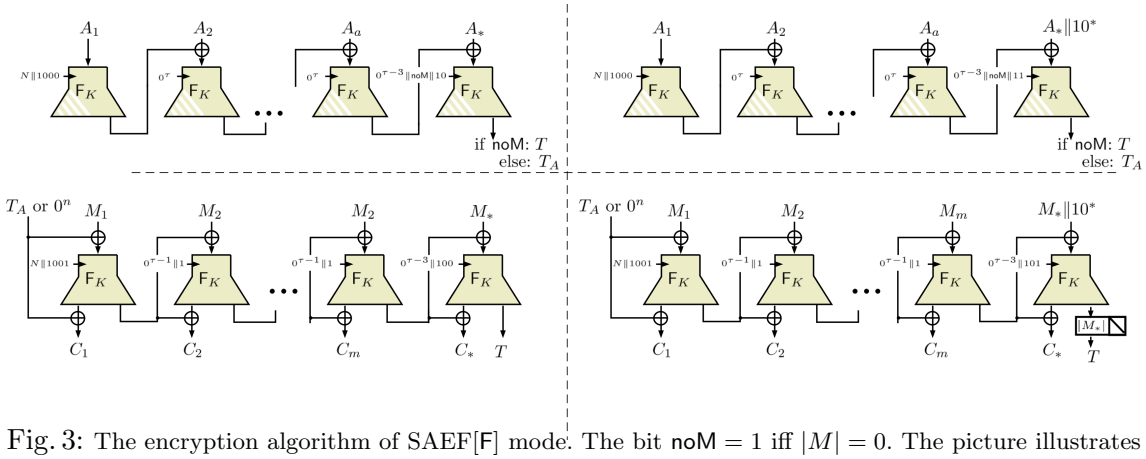

Fig. 3: The encryption algorithm of SAEF[F] mode. The bit $\mathsf{noM} = 1$ iff $|M| = 0$. The picture illustrates the processing of AD when length of AD is a multiple of n (**top left**) and when the length of AD is not a multiple of n (**top right**), and the processing of the message when length of the message is a multiple of n (**bottom left**) and when the length of message is not a multiple of n (**bottom right**). The white hatching denotes that an output block is not computed. If $|M| = 0$, the tag is equal to T_A; else if $|A| = 0$, the AD processing is skipped.

4 Trade-offs and implementation strategies

This section describes suitable hardware implementation strategies for the forkcipher primitive both in a more general sense (based on the IFI approach) and concretely for the ForkSkinny instantiation of IFI. Our work largely focuses on round-based implementations, as they are often the most suitable choice in practice. Furthermore, we establish strategies for serialized, unrolled and pipelined implementations, as well as illustrate both extremes of the speed-area trade-off space.

4.1 Round-based implementations

Hardware architecture. Figure 4a presents a round-based ForkSkinny hardware architecture, where combined encryption-decryption functionality can be enabled. The IS and TK registers respectively store internal cipher state and tweakey, and computations occur directly in these registers. The L register stores the state at the forking point, RC denotes the round constant, and BC is a combinational implementation of the branch constant. Decryption requires computing the decryption tweakey with $TKS^{r_{init}+r_1}(\cdot)$.

Internal parallelism. The (almost) independent nature of the two branches after the forking step in ForkSkinny gives rise to *internal* primitive parallelism, allowing to decrease the computational time beyond lowering the number of rounds. Inherent to an iterate-fork-iterate forkcipher, and regardless of the particular instantiation, the parallelism is *always* available, both in hardware and software. Note however, that for a key schedule where there is a function f that describes the tweakey of the C_0 branch as a function of the tweakey of the C_1 branch, the parallelism can be exploited at a very little cost in hardware. In the current ForkSkinny key schedule, such a function is: $f = TKS^{r_0}(\cdot)$. Figure 4b depicts a simplified version of the resulting architecture, revealing that the only overhead is a second instance of the round function and the function f itself (which is cheap, owing to Skinny's lightweight key schedule). As the natural way to implement a

forkcipher in hardware, it allows to compute a forkcipher call at the latency of a conventional tweakable block cipher [†]. In what follows, we refer to this strategy as *fast-forwarding* and we let `(//)` denote an implementation that makes use of the parallelism.

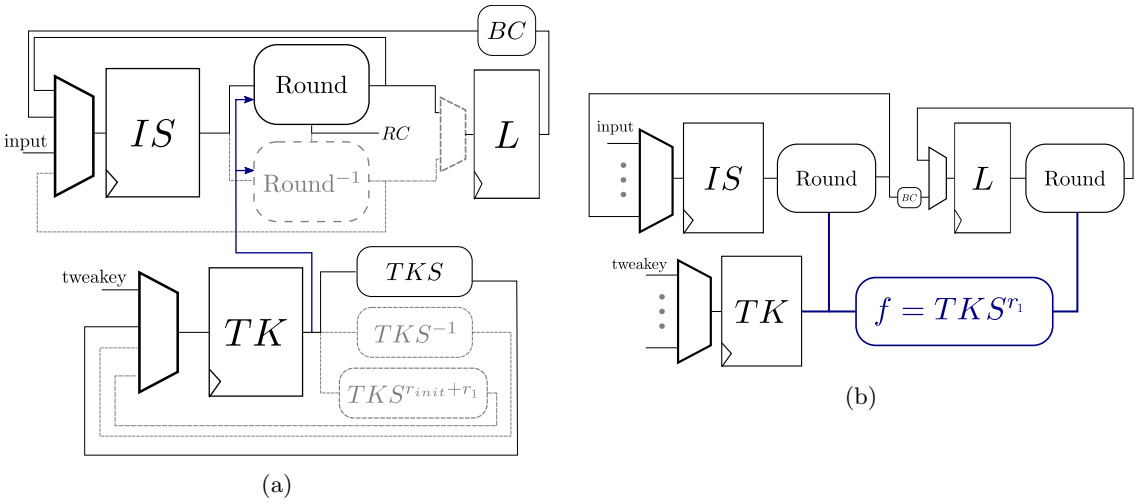

Fig. 4: Hardware architecture diagrams of **(a)** regular encryption/decryption architecture, (the optional decryption functionality is shaded)
(b) *fast-forwarding* the tweakey for efficient parallelism after forking

Decryption can also exploit this kind of parallelism, albeit at the cost of duplicating the key storage. Indeed, there currently exists no such f that captures the relation between the tweakey in the decryption and reconstruction branch (it changes every round). The attractiveness of the forkcipher can hence be further enhanced by designing a fork-friendly key schedule for efficient parallel decryption.

Instance-specific optimizations and features. For the sake of clarity, the architectures in Figure 4 abstract away the low-level, instance-dependent optimizations. For instance, setting the decryption key for ForkSkinny-128-192 and ForkSkinny-128-256 is much cheaper, because $TKS^{48}(\cdot) = TKS^{0}(\cdot)$ for a large part of the tweakey state. Moreover, since ForkSkinny-128-192 and ForkSkinny-128-288 do not utilize all of the available tweakey, storage of the unused cells can be replaced with conditional AND gates (zeroizing these cells every other round). Finally, all ForkSkinny instances can optionally *rewind* key and nonce from the computation registers, removing the need for additional storage.

Hardware implementations. Following the submission of this document, we will place the VHDL hardware description of the round-based architectures in the public domain. These implementations are highly configurable in terms of family members, encryption-only and encryption-decryption instances, and the exploitation of post-fork parallelism. In doing so, we hope to reduce the friction of including ForkAE in third-party benchmarking. Indeed, implementing a novel primitive requires considerably more exploration than a block cipher or stream cipher, for which the implementation trade-offs are already firmly established.

[†]Assuming, of course, that $r_{init} + r_0 = r_{init} + r_1$ is equal to the original number of TBC rounds.

4.2 Unrolled and high-throughput implementations

Unrolling strategies. Unrolling the rounds of a cryptographic primitive is a useful technique to reduce the latency of the primitive by amortizing the set-up and hold-times of sequential logic, or to maintain a high speed even when the design is to be clocked at a much lower frequency. *Fully unrolled* implementations are the most extreme and yield the output in a single cycle by instantiating all $r_{init}+r_0+r_1$ in hardware. Another, seemingly natural, strategy for an IFI forkcipher is *three-fold unrolling*, in which one implements an instance of $max(r_{init}, r_0, r_1)$ unrolled rounds with output taps at r_{init}, r_0, r_1 to compute resp. the rounds before forking, the C_0 branch and the C_1 branch. Figure 5 presents the synthesis results for the fully unrolled and three-fold unrolled strategies. In case encryption-decryption functionality is required, the combinational logic should approximately be doubled, while the sequential logic can be shared by introducing multiplexers. Like other cryptographic primitives, forkciphers can also be unrolled less aggressively (e.g. two or three rounds).

Fully unrolled **(1 cycle)**	Area [GE]	Critical path [ns]
ForkSkinny-64-192	34167	26
ForkSkinny-128-192	62387	37

Three-fold unrolled **(3 cycles)**	Area [GE]	Critical path [ns]
ForkSkinny-64-192	16221	14
ForkSkinny-128-192	29666	20

Fig. 5: Unrolled implementations (encryption) in Nangate 45nm

High-throughput implementations. As demonstrated by the designers, the low circuit depth of the Skinny round function makes it extremely well-suited for high-throughput, pipelined implementations [10]. While serial AE modes like Saef, Romulus or Ascon can only benefit from pipelining when considering a relatively high-end device (e.g., a server) that can interleave the messages of many communicating nodes, parallel modes like Paef can exploit a large pipeline depth to the fullest. Using the *fast-forwarding* approach from Section 4.1, it is relatively straightforward to construct pipelined ForkSkinny from the publicly available Skinny implementations [27], with a similar critical path. As another motivating example, high-throughput implementations are a common strategy for FPGA platforms [10], because the necessary pipelining registers come "for free" in an FPGA slice.

4.3 Reducing the area requirements

For highly serialized implementations, storage elements and multiplexers constitute the dominant resource utilization. Forkciphers are flexible in the sense that they allow to retrieve the forking state rather than to store it, either by *rewinding* from C_1 or by *restarting* from M. This flexibility allows to compute the $a+m$ forkcipher calls as $a+2m$ BC calls, bearing similarity with two-pass schemes (e.g. Sundae [8] with $a+2m+2$ BC calls).

5 Synthesis results and comparison

Synthesis flow. For reproducing the results in this article, we fully describe the synthesis parameters. We allow the use of Scan Flip-Flops. Synthesis occurs with exactly the same

parameters for all designs: `Synopsys Design Compiler 2017.N3` using `compile`, using the NANGATE 45NM open cell technology library in typical operating conditions.

ForkSkinny results. Figure 6 presents the synthesis results for SKINNY and FORKSKINNY (already with `write_enable` for the tweakey state). We can observe that the area requirements of a forkcipher are not that much larger than for a block cipher, and that the critical path only slightly increases. Moreover, as conjectured in Section 4.1, the internal forkcipher parallelism comes at a relatively low cost.

Area [GE]

Primitive	ENC-ONLY		ENC+DEC	
	Regular	Parallel	Regular	Parallel
SKINNY-64-192	3003	/	4522	/
SKINNY-128-256	4992	/	6355	/
SKINNY-128-384	5914	/	8311	/
FORKSKINNY-64-192	3692	4307	5362	6229
FORKSKINNY-128-192	5299	6113	7305	8608
FORKSKINNY-128-256	5842	6688	8101	9450
FORKSKINNY-128-288	6751	7917	9182	10876

Maximal frequency [MHz]

Primitive	ENC-ONLY		ENC+DEC	
	Regular	Parallel	Regular	Parallel
SKINNY-64-192	1351	/	1087	/
SKINNY-128-256	1087	/	1020	/
SKINNY-128-384	1020	/	962	/
FORKSKINNY-64-192	1282	1253	980	952
FORKSKINNY-128-192	1064	1020	877	877
FORKSKINNY-128-256	1064	1020	917	884
FORKSKINNY-128-288	990	962	862	820

Fig. 6: Synthesis results for SKINNY and FORKSKINNY primitives (in NANGATE 45NM)

Comparison targets. In the remainder of this section, we compare round-based implementations of the FORKSKINNY modes with a subset of NIST LWC candidates in similar categories: SKINNY-AEAD [9] (SKINNY-based, parallel, full security), ROMULUS [19] (SKINNY-based, serial, short message performance) and ASCON [18] (short message performance).

Mode estimation methodology. We acknowledge the engineering effort and added value of fairly benchmarking hardware implementations on different platforms. Given the current timeline, we provide a hybrid estimate of the area of the compared designs, in our effort to provide timely feedback before the announcement of the second round candidates. As a first-order estimate, we *synthesize* the underlying primitives (see Figure 6) under identical conditions and *estimate* the area of a straightforward implementation of the modes (using the NANGATE 45NM numbers: `7.67GE` for Scan flip-flops (SFF), `2.33GE` for multiplexers (MUX), `2GE` for XOR/XNOR and `1GE` for NAND). In our estimates, we consider a bus interface of $n/4$ bits. While we do not count the area of interfaces (e.g. FIFOs at input and output), a reasonably-sized bus implies that it will not be possible to write all inputs in a single go, requiring `write_enable` for all registers that store inputs. Importantly for AEAD schemes, we count either the storage for key, nonce (if applicable) and counter (if applicable), or the logic required to recompute them (the latter approach is best for SKINNY-based designs). Although some multiplexers are possibly added by implementing a mode on top of a primitive, we assume for this very coarse estimation that the critical path of the primitive is unchanged.

Exemplifying this estimate framework, for *encryption-only implementations*, the PAEF mode (with l-bit counter) requires n SFF, $n + t + l$ MUX ($t + l$ MUX for the parallel 128-bit versions as recomputing $TK1$ happens automatically), $5n/4$ XOR/XNOR and n NAND, on top of the primitive as synthesized in Figure 6. Similarly, the SAEF mode requires n SFF, $2n$ MUX (n MUX for the parallel 128-bit versions) and $5n/4$ XOR/XNOR

for encryption. The numbers for *encryption-decryption* ForkSkinny are obtained in a similar fashion.

We estimate Skinny-AEAD in the same way, also using the primitives of Figure 6. For Romulus, we consider the architecture suggested by the designers [19], adding 128 MUX for `write_enable` of the internal state. For Ascon, we resynthesize the publicly available Asconv1.1 implementations [17] (no performance changes w.r.t Asconv1.2), yielding `8125GE` with a critical path of 1.71 ns for Ascon128 and `8338 GE` with a critical path of 2.06 ns for Ascon128a. To match the assumptions of the other targets, we add key storage (`982GE`) but subtract `100GE` for control, which is not included for the other schemes.

Comparison with Skinny-based AE schemes. For the Skinny-based designs, Figure 7 compares round-based implementations that encrypt a blocks of associated data and m blocks of message. The modes are partitioned first on tag sizes, then on the underlying Skinny[‡]and on properties of the mode; Saef and Romulus are serial modes with birthday-bounded security, whereas Skinny-AEAD and Paef are parallel modes with full n-bit security. When the input consists of a single message block, the ForkSkinny modes are up to twice as fast as the competition. From the visual presentation in Figure 8, one can identify from which input sizes ForkSkinny performance reduces.

Implementation (round-based)	Area [GE] E-only	Area [GE] EncDec	Number of cycles for encrypting $(a+m)$ **64-bit** blocks						
			$a=0$			$a=1$			General
			$m=1$	$m=2$	$m=3$	$m=0$	$m=1$	$m=2$	
Sk-AEAD M6	8095	9458	96	96	144	48	96	**96**	$48(\lceil \frac{a}{2} \rceil + \lceil \frac{m}{2} \rceil + 1)$
PAEF-64-192	5034	6704	**63**	126	189	**40**	103	166	$40(a + 1.575m)$
PAEF-64-192 (//)	5500	7422	**40**	**80**	**120**	**40**	**80**	120	$40(a+m)$

Implementation (round-based)	Area [GE] E-only	Area [GE] EncDec	Number of cycles for encrypting $(a+m)$ **128-bit** blocks						
			$a=0$			$a=1$			General
			$m=1$	$m=2$	$m=3$	$m=0$	$m=1$	$m=2$	
Romulus-N3	6288	6406	96	144	192	**48**	**96**	**144**	$48(\lceil \frac{a-1}{1.75} \rceil + m + 1)$
SAEF-128-192	7197	9203	**75**	150	225	**48**	123	198	$48(a + 1.562m)$
SAEF-128-256	7740	9999	**75**	150	225	**48**	123	198	$48(a + 1.562m)$
SAEF-128-192 (//)	7713	10804	**48**	**96**	**144**	**48**	**96**	**144**	$48(a+m)$
SAEF-128-256 (//)	8288	11646	**48**	**96**	**144**	**48**	**96**	**144**	$48(a+m)$
Sk-AEAD M5	8746	10109	96	144	192	96	144	192	$48(a+m+1)$
PAEF-128-192 (//)	8020	11112	**48**	**96**	**144**	**48**	**96**	**144**	$\boldsymbol{48(a+m)}$
PAEF-128-256 (//)	8745	12103	**48**	**96**	**144**	**48**	**96**	**144**	$\boldsymbol{48(a+m)}$
Romulus-N1	7018	7136	112	168	224	**56**	**112**	**168**	$56(\lceil \frac{a-1}{2} \rceil + m + 1)$
Sk-AEAD M1-2	9966	12363	112	168	224	112	168	224	$56(a+m+1)$
PAEF-128-288	9274	11705	**87**	174	261	**56**	**143**	230	$56(a + 1.553m)$
PAEF-128-288 (//)	10141	13697	**56**	**112**	**168**	**56**	**112**	**168**	$56(a+m)$

Fig. 7: Nangate 45nm area and cycle counts for round-based implementations of Skinny-based authenticated encryption, considering a blocks of associated data and m blocks message. As established earlier, the area is partly synthesized and partly estimated.

General speed-area investigation of NIST candidates. Comparing with NIST candidates that are based on *other* (T)BC or even permutations (like Ascon) is more com-

[‡]Although having equal tag sizes, Skinny-AEAD M6 and PAEF-64-192 are not a perfect match. Of all Skinny-based designs, they are the closest competitor when tiny messages ($\leq$8 bytes) are predominant.

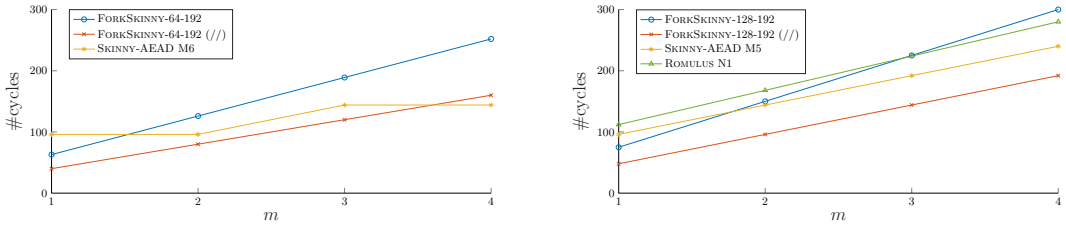

Fig. 8: Number of cycles for round-based implementations as a function of input size ($a = 0$, m variable) for 64-bit blocks **(left)** and 128-bit blocks **(right)**

plicated. Admitting that we cannot cover every implementation strategy for every scheme with limited resources, we attempt to approximate the speed-area positions of these candidates in the trade-off space using the aforementioned conservative estimation methodology. The speed metric (execution time per message) incorporates both the number of cycles (cf. Figure 7) and the critical path of the design (cf. Figure 6) to account for the difference in circuit depth between the primitives. Figure 9 plots some interesting comparison targets in a speed-area graph in four configurations, two of which with 64-bit input blocks and two with 128-bit input blocks. We can observe that for very short messages (≤ 8 bytes), the PAEF-64-192 instances have excellent properties, either outperforming Ascon or providing a similar latency with smaller area. Considering messages of 128 bits or longer, while the encryption-decryption ForkSkinny architectures are no longer the absolute best in class, they are capable of occupying several good positions in the speed-area plane, while the encryption-only architectures still have excellent performance.

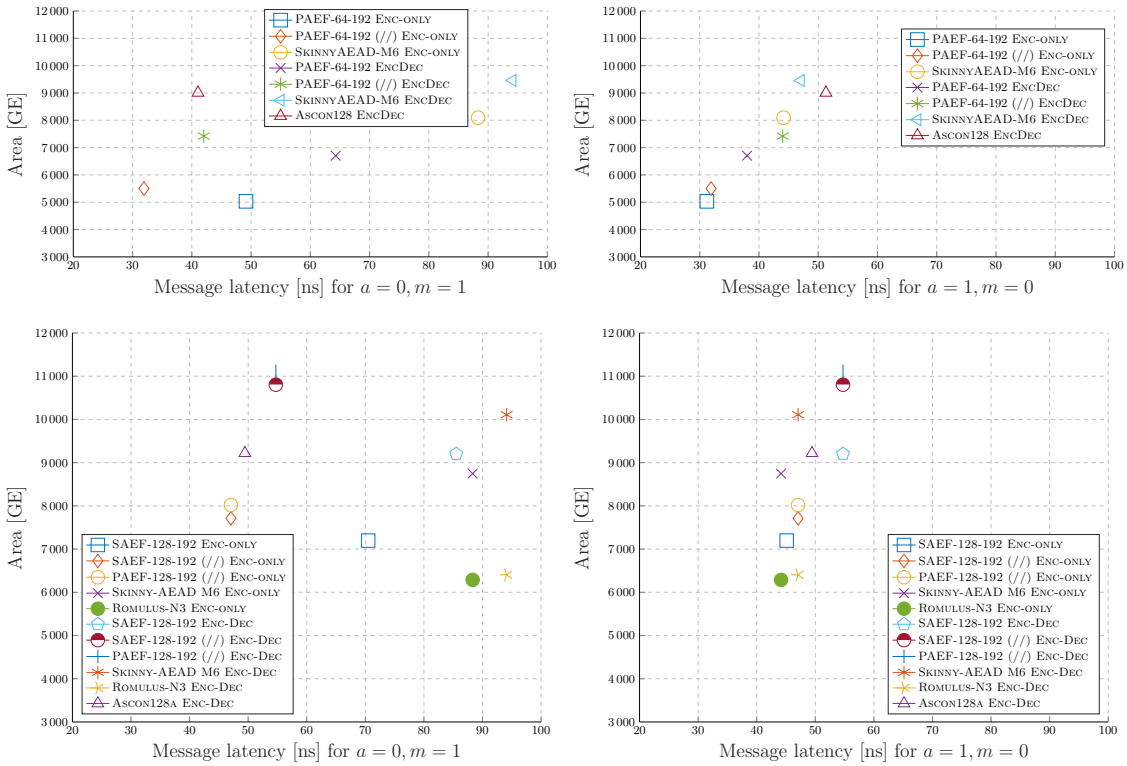

Fig. 9: Speed vs area for several NIST candidates;
64-bit input blocks **(top)** 128-bit input blocks **(bottom)**

6 Conclusion and future work

Forkcipher and its instantiation ForkSkinny are novel constructions which come with new implementation challenges and possibilities. In this work we explore those. We describe multiple implementation strategies for ForkSkinny that allow us to fine-tune the desired performance-area trade-off. Some of these strategies are standard (round-based, serial, unrolled) but others are using the intrinsic properties of ForkSkinny (internal parallelism through fast forwarding, and restarting/rewinding). The latter strategies can be directly applied to any iterate-fork-iterate forckcipher. We note that these strategies cannot be exploited by the existing fixed-input length primitives (although similar strategies are pertinent for recent variable-input length primitives such as Farfalle [12]).
Furthermore, we design a set of highly configurable implementations of ForkSkinny that allow to mix-and-match the desired instance together with a set of implementation strategies. We also release these implementations in the public domain.
Finally, we provide a comparison of the performance and area of a subset of the NIST LWC candidates targeting the short message scenario. The data for the comparison is obtained by a hybrid, reproducible method that combines an actual implementation of the primitive, and a conservative estimation of the mode.

Future work. Our results highlight several future research avenues. The ForkSkinny decryption cannot benefit from the same optimizations as the encryption because of the post-fork key schedules going in opposite direction. One idea in that direction is to construct a key schedule for efficient parallel decryption, which would allow also for an efficient switching between the subkeys of the two forks both in encryption *and* decryption.

Acknowledgements. This work was supported in part by the Research Council KU Leuven C1 on Security and Privacy for Cyber-Physical Systems and the Internet of Things with contract number C16/15/058. In addition, this work is supported by the Horizon 2020 research and innovation programme under Cathedral ERC Advanced Grant 695305. Arnab Roy is supported by the EPSRC grant No. EPSRC EP/N011635/1.

References

[1] 3GPP TS 22.261: Service requirements for next generation new services and markets. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3107
[2] 3GPP TS 36.213: Evolved Universal Terrestrial Radio Access (E-UTRA); Physical layer procedures. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2427
[3] NB-IoT: Enabling New Business Opportunities. http://www.huawei.com/minisite/iot/img/nb_iot_whitepaper_en.pdf
[4] Specification of Secure Onboard Communication. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf
[5] DeTOP Dexterous Transradial Osseointegrated Prosthesis with neural control and sensory feedback. http://www.detop-project.eu/ (2016)
[6] Andreeva, E., Lallemand, V., Purnal, A., Reyhanitabar, R., Roy, A., Vizár, D.: Forkcipher: a New Primitive for Authenticated Encryption of Very Short Messages. In: Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security
[7] Andreeva, E., Lallemand, V., Purnal, A., Reyhanitabar, R., Roy, A., Vizár, D.: ForkAE v1. Submission to NIST Lightweight Cryptography Project (2019)
[8] Banik, S., Bogdanov, A., Peyrin, T., Sasaki, Y., Sim, S.M., Tischhauser, E., Todo, Y.: SUNDAE-GIFT. NIST Lightweight Cryptography: Submission to Round **1**

[9] Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: SKINNY-AEAD and SKINNY-Hash

[10] Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In: Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II. pp. 123–153 (2016). https://doi.org/10.1007/978-3-662-53008-5_5

[11] Bernstein, D.J.: Cryptographic competitions: CAESAR. http://competitions.cr.yp.to

[12] Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Assche, G.V., Keer, R.V.: Farfalle: parallel permutation-based cryptography. IACR Trans. Symmetric Cryptol. **2017**(4), 1–38 (2017), https://tosc.iacr.org/index.php/ToSC/article/view/801

[13] Biryukov, A., Perrin, L.: State of the Art in Lightweight Symmetric Cryptography. IACR Cryptology ePrint Archive **2017**, 511 (2017), http://eprint.iacr.org/2017/511

[14] Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4727, pp. 450–466. Springer (2007). https://doi.org/10.1007/978-3-540-74735-2_31, https://doi.org/10.1007/978-3-540-74735-2_31

[15] Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçin, T.: PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In: Wang, X., Sako, K. (eds.) Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7658, pp. 208–225. Springer (2012). https://doi.org/10.1007/978-3-642-34961-4_14, https://doi.org/10.1007/978-3-642-34961-4_14

[16] Cannière, C.D., Dunkelman, O., Knezevic, M.: KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings. Lecture Notes in Computer Science, vol. 5747, pp. 272–288. Springer (2009). https://doi.org/10.1007/978-3-642-04138-9_20, https://doi.org/10.1007/978-3-642-04138-9_20

[17] IAIK, T.G.: Hardware implementations of the authenticated encryption design ASCON. https://github.com/IAIK/ascon_hardware/tree/master/generic_implementation

[18] Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: Ascon v1.2. Submission to NIST Lightweight Cryptography Project (2019)

[19] Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: Romulus v1. Submission to NIST Lightweight Cryptography Project (2019)

[20] Jean, J., Nikolić, I., Peyrin, T.: Tweaks and Keys for Block Ciphers: The TWEAKEY Framework. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology – ASIACRYPT 2014. pp. 274–288. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

[21] Krovetz, T., Rogaway, P.: The Software Performance of Authenticated-Encryption Modes. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 306–327. Springer (2011)

[22] McGrew, D.A., Viega, J.: The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In: Canteaut, A., Viswanathan, K. (eds.) INDOCRYPT 2004. LNCS, vol. 3348, pp. 343–355. Springer (2004)

[23] Mouha, N.: The Design Space of Lightweight Cryptography. IACR Cryptology ePrint Archive **2015**, 303 (2015), http://eprint.iacr.org/2015/303

[24] NIST: Ligthweight Cryptography. https://csrc.nist.gov/projects/lightweight-cryptography

[25] PLC, M.: Wireless Tyre Pressure Monitoring (wTPMS). https://www.meggitt.com/products-services/tyre-pressure-monitoring/

[26] Rogaway, P.: Authenticated-encryption with associated-data. In: Atluri, V. (ed.) Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002. pp. 98–107. ACM (2002). https://doi.org/10.1145/586110.586125, https://doi.org/10.1145/586110.586125

[27] Skinny: Skinny Hardware Implementations - Application-Specific Integrated Circuits (ASIC). https://sites.google.com/site/skinnycipher/implementation

[28] Whiting, D., Housley, R., Ferguson, N.: Counter with CBC-MAC (CCM). IETF RFC 3610 (Informational) (Sep 2003), http://www.ietf.org/rfc/rfc3610.txt