Beetle Family of Lightweight and Secure Authenticated Encryption Ciphers

Avik Chakraborti[1], Nilanjan Datta[2], Mridul Nandi[3] and Kan Yasuda[1]

[1] NTT Secure Platform Laboratories, Japan, {lastname.firstname}@lab.ntt.co.jp
[2] Indian Institute of Technology, Kharagpur, nilanjan_isi_jrf@yahoo.com
[3] Indian Statistical Institute, Kolkata, mridul.nandi@gmail.com

Abstract

This paper presents a lightweight, sponge-based authenticated encryption (AE) family called Beetle. When instantiated with the PHOTON permutation from CRYPTO 2011, Beetle achieves the smallest footprint - consuming only a few more than 600 LUTs on FPGA while maintaining 64-bit security. This figure is significantly smaller than all known lightweight AE candidates which consume more than 1,000 LUTs, including the latest COFB-AES from CHES 2017. In order to realize such small hardware implementation, we equip Beetle with an “extremely tight” bound of security. The trick is to use combined feedback to create a difference between the cipher text block and the rate part of the next feedback (in traditional sponge these two values are the same). Then we are able to show that Beetle is provably secure up to $\min\{c - \log r, b/2, r\}$ bits, where b is the permutation size and r and c are parameters called rate and capacity, respectively. The tight security bound allows us to select the smallest security parameters, which in turn result in the smallest footprint.

Keywords: Beetle, sponge, PHOTON, authenticated encryption, lightweight, permutation.

1 Introduction

Due to the recent rise in communication networks operated on small devices, authenticated encryption (AE) is expected to play a key role in securing these networks, providing both confidentiality and authenticity via symmetric-key cryptographic primitives. In light of CAESAR competition [3] for AE and NIST’s lightweight cryptography project [27], people recognize the apparent lack of AE standards suitable for the whole spectrum of lightweight applications. As a result, several lightweight AE proposals have emerged. These include: ASCON [19], CLOC/SILC [22, 23], Gibbon/Hanuman [6], JAMBU [32] and Ketje [15]from the CAESAR competition, as well as the recently developed COFB [17, 18].

1.1 Block-Cipher-Based vs. Sponge-Based Constructions

We can classify the above AE proposals into two groups based on their designs. The first group: CLOC/SILC, JAMBU and COFB, follow a rather classical style of iterating a block cipher. The second group: ASCON, Gibbon/Hanuman and Ketje, are based on the sponge construction introduced by Bertoni et al. in 2007 [9].

The sponge construction, now standardized as the SHA-3 hash function, consists of a sequential application of a permutation f on a state of b bits. This state is partitioned into an r-bit rate (or outer part) and a c-bit capacity (or inner part), where $b = r + c$. In the absorption phase, message blocks of size r bits are absorbed by the outer part and the state is transformed using f, while in the squeezing phase, digests are extracted from the outer part r bits at a time.

One of the advantages of the sponge-based design is that they can build lightweight hash functions. Indeed, since the introduction of the design, a number of lightweight hash algorithms have been proposed, including SPONGENT [16], QUARK [7] and PHOTON [20]. Mainly due to the smaller size of the total state values, the footprint of these algorithms is generally smaller than classical Merkle-Damgård hash functions, which iterate a compression function (rather than a permutation), and these compression functions essentially employ the design of block ciphers.

Alongside being used as a "simple" hash function (such as SHA-3 standard Keccak [29, 12]), the keyed variants of sponge mode have become very popular modes of operation for a permutation to build a wide spectrum of symmetric-key primitives like message authentication codes [13], pseudorandom functions, Extendable-Output Functions ("XOFs") [29] and authenticated encryption (AE) modes [10, 11]. The keyed Sponge principle also got adopted in Spritz, a new RC4-like stream cipher [31], and in 10 out of 57 submissions to the currently running CAESAR [3] competition on authenticated encryption.

Of these AE proposals, unexpectedly, one of the smallest is the COFB, which is block-cipher-based. This does not seem to be consistent with what we have learned from the designs of hash functions; we should be able to build more lightweight schemes with the sponge construction than with block ciphers. Where is this gap coming from? This work answers this question by demonstrating that actually, also for AE, one can build smaller schemes with the sponge construction.

1.2 Existing Security Bounds of Sponge-based AE

Encryption via the Sponge is typically done via the Duplex construction [11], a stateful construction consisting of an initialization interface and a duplexing interface. The initialization interface can be called to initialize an all-zero state; the duplexing interface absorbs a message of size $< r$ bits and squeezes $\leq r$ bits of the outer part. The security of the Duplex traces back to the indifferentiability of the classical Sponge, yielding a $O(2^{c/2})$ security bound. Bertoni et al. [11] showed that the Duplex, in turn, allows for authenticated encryption in the form of SpongeWrap. This mode is, de facto, the basis of the majority of Sponge-based submissions to the CAESAR competition. Jovanovic et al. [24] claimed that Sponge-based constructions for authenticated encryption can achieve the significantly higher bound of $\mathsf{min}\{2^{b/2}, 2^c, 2^k\}$ asymptotically, with $b > c$ the permutation size.

1.2.1 Limitations of Jovanovic et al's Result

In the above mentioned result, for the integrity security the authors have assumed that the number of forgery blocks is limited. To be more specific, total number of forgery attempted blocks σ_v is restricted to satisfy the following:

$$q_p + \sigma_e + \sigma_v \ \leq \ 2^c/\sigma_v,$$

where σ_e is the total number of encryption query blocks and q_p is the number of permutation queries. The above equation clearly suggest that the number of decryption blocks should be at most $2^{c/2}$.

But, in the real life applications, it is more likely that the adversary would make a large number of decryption queries to mount the integrity (or forging) attack, and hence the overall bound should be given in terms decryption queries along with the total number of encryption and permutation queries. Considering the decryption queries, their result achieves $\mathsf{min}\{2^{c/2}, 2^k\}$ integrity security.

1.3 Our Contributions

In this paper, we present an efficient sponge based authenticated encryption mode called Beetle that provides $min\{c - \log r, b/2, r\}$ bits of security without any restrictions on decryption queries.

1. **Combining a Feedback Function with Sponge.** In traditional sponge based modes, the plain text is XORed with the rate part of the permutation output to get the cipher text, and the same value is used as the rate part of the next feedback (i.e. input to the subsequent permutation). However, in Beetle construction, a Combined Feedback is used to create a difference between the cipher text block and the rate part of the next feedback (in traditional sponge these two values are the same). With this simple tweak, we have shown that the mode achieves improved security without any additional storage.

2. PHOTON **Instantiation and Hardware Implementation.** We present our hardware implementation results for our recommended instantiations, mainly targeting two environments: one for the lightweight applications, other to achieve high security. Our implementation results depicts that (i) the lightweight version has excellent performance and it achieves the smallest footprint among all known lightweight candidates, (ii) the highly-secure version provides good security and most lightweight among other existing constructions achieving about 121 bits of security.

1.4 Design Rationale behind Our Construction

1.4.1 Sponge-Based

Design of sponge based AE schemes have drawn lots of attention in the recent years. They adopt the design rational of sponge functions more precisely duplex sponge functions. SpongeWrap [11] is a primitive of this type. Several constructions following this type exist in the literature. In this work, we take the approach of designing a sponge based AE scheme that achieves the above security level with a minimized hardware area. We observed that, if we adopt a simple combined feedback (described in [18]) during data absorption and release phase, we can achieve the desired security bound without any additional overhead as compared to the traditional duplex sponge mode. Moreover, this result is significant as we can minimize the hardware area by adopting a low state permutation but with a standard security bound of 64-bit (this is a standard bound in lightweight crypto). However, if we adopt a permutation with a larger state size, still we can get a better hardware area than the existing schemes (with comparable security bound) as the design has a negligible overhead from the traditional sponge design.

1.4.2 Combined Feedback

Chakraborti et. al. in [18] used combined feedback to design a lightweight block cipher based AE scheme COFB. However, this design needs to maintain an additional secret state for masking. We have observed that, we can remove this extra state by adopting a sponge mode where the full state is not exposed through the ciphertext and the message block (only the rate part is exposed). Moreover, this combined feedback helps us to avoid any other additional operations but only to follow the traditional sponge mode.

1.4.3 Choosing PHOTON

Finally, we show that if we use a very lightweight permutation like PHOTON [20, 21] then to the best of our knowledge, we can achieve an AE scheme with the lowest hardware footprint. The benchmark in Sect 5 proves our statement. The benchmark also shows that even if we adopt a larger state permutation, still we have better hardware area with a high security bound than the existing schemes.

1.5 Beetle in the Light of NIST Lightweight Cryptography Project

Here we provide a brief discussuion on the significance of our design Beetle in the light of NIST Lightweight Cryptography Project. In addition to the standard functionalities, NIST has set the following minimum requirements from AE submissions, when the key size is restricted to 128 bits:

- Time Complexity: any cryptanalytic attack should need at least $T = 2^{112}$ computations (includes the total time required to process the offline evaluations of the underlying permutation) in a single key setting.
- Data Complexity: the total number of message bytes (among all messages and associated data) processed through the underlying permutation under a single key should be at least $D = 2^{50} - 1$.

Now, the best known bound for the original sponge mode SpongeAE is $O(\frac{D^2+D.T}{2^c})$, while Beetle provides a security bound of $O(\frac{D^2+DT}{2^b} + \frac{r.(D+T)}{2^c})$. This clearly depicts that if SpongeAE mode is instantiated with a 256-bit permutation, then the capacity must be at least $112 + 48 = 160$ bits. However, only 120-bit capacity is sufficient for Beetle. This essentially ensures lesser number of permutation invocations in case of Beetle, which makes it more energy efficient. In fact, in case of short messages of length 16 bytes, Beetle leads to 33.3% savings in the energy consumption, which is quite significant for the lightweight applications.

2 Preliminaries

In this section we build up all the notations and recall basic security definitions for authenticated encryption. We also recall some important basic results on the security of authenticated encryption.

2.1 Notation

Fix three positive integer b, r and c to represent state size, rate and capacity of a sponge construction. By definition, $b = r + c$. We denote a block by an element of $\{0,1\}^r$ (i.e, a block is an r-bit string). For any $X \in \{0,1\}^*$, where $\{0,1\}^*$ is the set of all finite bit strings (including λ, the empty string), we denote the number of bits of X by $|X|$. Note that $|\lambda| = 0$. For two bit strings X and Y, $X\|Y$ denotes the concatenation of X and Y. A bit string X is called a *complete* (or *incomplete*) block if $|X| = r$ (or $|X| < r$ respectively). We write the set of all complete (or incomplete) blocks as $\mathcal{B}$ (or $\mathcal{B}^<$ respectively). Let $\mathcal{B}^{\le} = \mathcal{B}^< \cup \mathcal{B}$ denote the set of all blocks. For $B \in \mathcal{B}^{\le}$, we define $\overline{B}$ using 10^* padding with B, to make it complete. Given $Z \in \{0,1\}^*$, we define the parsing of Z into r-bit blocks as $(Z[1], Z[2], \ldots, Z[z]) \xleftarrow{r} Z$, where $z = \lceil |Z|/r \rceil$, $|Z[i]| = r$ for all $i < z$ and $1 \le |Z[z]| \le r$ such that $Z = (Z[1] \,\|\, Z[2] \,\|\, \cdots \,\|\, Z[z])$. If $Z = \lambda$, we let $z = 1$ and $Z[1] = \lambda$. We write $||Z|| = z$ (number of blocks present in Z). We similarly write $(Z[1], Z[2], \ldots, Z[z]) \xleftarrow{r} Z$ to denote the parsing of the bit string Z into r bit strings $Z[1], Z[2], \ldots, Z[z-1]$ and $1 \le |Z[z]| \le r$. Given any sequence $Z = (Z[1], \ldots, Z[s])$ and $1 \le a \le b \le s$, we represent the sub sequence $(Z[a], \ldots, Z[b])$ by $Z[a..b]$. Similarly, for integers $a \le b$, we write $[a..b]$ for the set $\{a, a+1, \ldots, b\}$. We use the notation $\mathsf{trunc}(Z, r)$ to denote the most significant r bits of the binary string Z. Let $\gamma = (\gamma[1], \ldots, \gamma[s])$ be a tuple of equal-length strings. We define $\mathsf{mcoll}(\gamma) = m$ if there exist distinct $i_1, \ldots, i_m \in [1..s]$ such that $\gamma[i_1] = \cdots = \gamma[i_m]$

and m is the maximum of such integer. We call $\{i_1, \ldots, i_m\}$ to be an m-multi-collision set for γ.

2.2 Security Model

In this section we provide the security definitions for authenticated encryption in ideal permutation model.

An authenticated encryption (AE) is an integrated scheme that provides both privacy of a plaintext $M \in \{0,1\}^*$ and authenticity of M as well as associate data $A \in \{0,1\}^*$. Taking a nonce N (which is a value never repeats at encryption) together with associated date A and plaintext M, the encryption function of AE, $\mathcal{E}_K$, produces a tagged-ciphertext (C, T) where $|C| = |M|$ and $|T| = t$. Typically, t is a fixed length. The corresponding decryption function, $\mathcal{D}_K$, takes (N, A, C, T) and returns a decrypted plaintext M when the verification on (N, A, C, T) is successful, otherwise returns the atomic error symbol denoted by $\perp$.

Unified Security Notion for AE in Random Permutation Model. Let f be the underlying idealized permutation of an AE scheme $\mathcal{E}$. We define the advantage of an adversary $\mathcal{A}$ in breaking the AE-security of $\mathcal{E}$ under *ideal permutation* model as follows:

$$\mathbf{Adv}^{\mathrm{AE}}_{\mathcal{E}}(\mathcal{A}) := |\Pr[\mathcal{A}^{f^{\pm}, \mathcal{E}_K, \mathcal{D}_k} = 1] - \Pr[\mathcal{A}^{f^{\pm}, \$, \perp} = 1]|,$$

where the probabilities are taken over the random choices of f, \$, K, and the random choices of $\mathcal{A}$, if any. The fact that the adversary has access to both the forward and inverse permutations in f is denoted by $f^{\pm}$. We assume that adversary $\mathcal{A}$ is nonce-respecting, which means that it never makes two queries to $\mathcal{E}_K$ or \$ with the same nonce. By $\mathbf{Adv}^{\mathrm{AE}}_{\mathsf{SP}}(q_e, q_p, q_v, \sigma_e, \sigma_v)$ we denote the maximum advantage taken over all adversaries that makes at most q_e encryption queries with a total length of at most σ_e, at most q_p queries to $f^{\pm}$ and at most q_v decryption queries with a total length of at most σ_v.

2.3 Coefficients-H Technique

In this section we give a quick high-level outline of coefficients H technique due to Patarin [30]. We will use this technique (without giving a proof) to prove our main theorem. Consider two oracles $\mathcal{O}_0$ (the ideal oracle) and $\mathcal{O}_1$ (the real oracle). Let $\mathcal{T}$ denote the set of all possible *transcripts* an adversary can obtain. For any view $\tau \in \mathcal{T}$, we will denote the probability to realize the view as $\mathsf{ip}_{\mathsf{real}}(\tau)$ (or $\mathsf{ip}_{\mathsf{ideal}}(\tau)$) when it is interacting with the real (or ideal respectively) oracle. We call these *interpolation probabilities.* w.o.l.g., we assume that adversary is deterministic. Hence, the interpolation probabilities are the properties of the oracles only. As we deal with stateless oracles, these probabilities are independent of the order of query responses in the view.

Theorem 1. *Suppose for a set $\mathcal{T}_{good} \subseteq \mathcal{T}$ of views (called the* **good views***) the following hold:*

1. *For any adversary $\mathcal{A}$ playing against $\mathcal{O}_0$ (the ideal), the probability of getting a view in $\mathcal{T}_{good}$ is at least $1 - \epsilon_1$. We may denote the set $\mathcal{T} \setminus \mathcal{T}_{good}$ by $\mathcal{T}_{bad}$. Hence, the the probability of getting a view in $\mathcal{T}_{bad}$ is at most ϵ_{bad}.*
2. *For any view $\tau \in \mathcal{T}_{good}$, we have*

$$\mathsf{ip}_{\mathsf{real}}(\tau) \geq (1 - \epsilon_{ratio}) \cdot \mathsf{ip}_{\mathsf{ideal}}(\tau)$$

For an oracle $\mathcal{O}_1$ satisfying (1) and (2) above, for any adversary $\mathcal{A}$, we have $\mathbf{Adv}^{\mathrm{prf}}_{\mathcal{O}_1}(\mathcal{A}) \leq \epsilon_{bad} + \epsilon_{ratio}$.

3 Specification of Beetle

In this section, we provide a formal specification of Beetle family, a variant of traditional duplex sponge. Similar to the original construction, we use a b-bit permutation f with rate r and capacity $c = b - r$. First we specify all the basic building blocks and parameters used in our construction, and then provide the formal algorithm along with a pictorial description.

3.1 External Parameters and Recommended Parameter Sets

The rate r and capacity c is taken as the external parameter. Recall that, these two explicitly determine the state size i.e. $b = r + c$. Depending on the application, we suggest the following recommended parameter sets:

Beetle[Light+] (for lightweight applications): This instantiation mainly focuses on lightweight applications. Here, we recommend to use the permutation P_{144} with the parameter set $(r = 64, c = 80)$.

Beetle[Secure+] (for highly secure applications): This is a highly secure instantiation of Beetle. Here, we recommend to use the permutation P_{256} (described in [20, 21]) with the parameter set $(r = 128, c = 128)$.

3.2 Mathematical Components

3.2.1 Feedback Function ρ

Let $W \in \{0,1\}^r$ and $(W[1], W[2]) \overset{r/2}{\leftarrow} W$, where $W[i] \in \{0,1\}^{r/2}$. We define $\mathsf{shuffle} : \mathcal{B} \to \mathcal{B}$ as $\mathsf{shuffle}(W) = (W[2], W[2] \oplus W[1])$. For $I_1 \in \mathcal{B}$ and $I_2 \in \mathcal{B}$, we define the feedback function $\rho : \mathcal{B} \times \mathcal{B} \to \mathcal{B} \times \mathcal{B}$ as follows:

$$\rho(I_1, I_2) = (O_1, O_2),\ \text{where } O_1 := \rho_1(I_1, I_2) = \mathsf{shuffle}(I_1) \oplus I_2,\ O_2 := \rho_2(I_1, I_2) = I_1 \oplus I_2.$$

The corresponding inverse feedback function $\rho' : \mathcal{B} \times \mathcal{B} \to \mathcal{B} \times \mathcal{B}$ is defined as

$$\rho'(I_1, O_2) = (O_1, I_2),\ \text{where } O_1 := \rho'_1(I_1, O_2) = \mathsf{shuffle}(I_1) \oplus I_1 \oplus O_2,\ I_2 := \rho_2(I_1, O_2) = I_1 \oplus O_2.$$

Note that, here we need the following requirements on the function shuffle:

- The mapping $I \to \mathsf{shuffle}(I)$ is bijective.
- The mapping $I \to \mathsf{shuffle}(I) \oplus I$ is also bijective.

It is easy to see that our choice of shuffle satisfies both the requirements.

3.2.2 PHOTON Permutation

PHOTON is a permutation on b bits where b can be written as $b = d^2.s$. The internal state of PHOTON is represented as a $(d \times d)$ matrix whose each cell element is of s bits long. The permutation is composed of 12 rounds, each round containing four layers:

- AddConstants (AC). This function applies round-dependent constants to each cell of the first column.
- SBox (SBox). This function applies the s-bit Sbox to every cell of the internal state. For $s = 4$, Present SBox and $s = 8$, AES SBox are used.
- ShiftRows (SR): This function simply rotates each cell located at row i by i positions to the left.

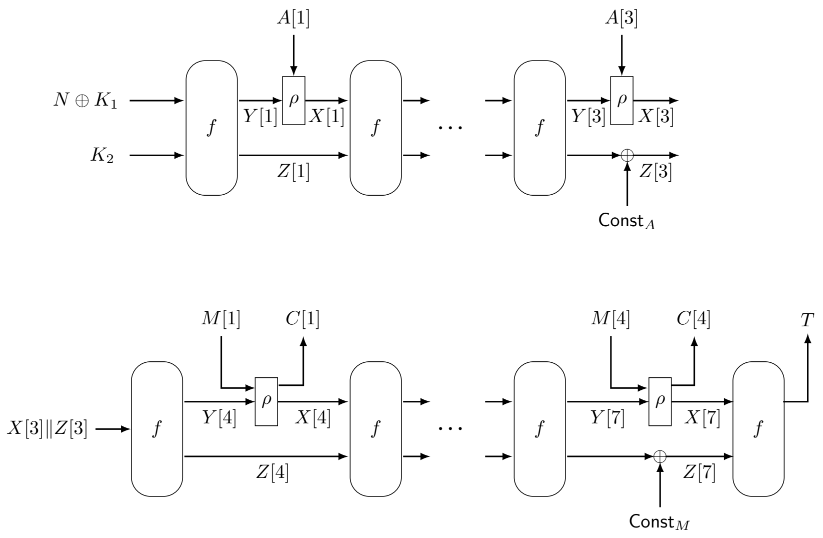

Figure 1: Beetle Construction with $a = 3$ associated data blocks and $m = 4$ message blocks. The value Const_A and Const_M is used to denote whether last associated data block and message block is complete or not.

- MixColumns (MC): This function updates linearly all columns independently using a light non MDS matrix M serially (d times) such that M^d is an MDS matrix. This provides maximal diffusion and also makes the circuit efficient for hardware implementation.

Complete details of PHOTON can be found in [21, 20]. In this work, we use two versions of PHOTON permutations on 144 bits and 256 bits which are denoted as P_{144} and P_{256} respectively. The parameters for these two versions are ($s = 4$, $d = 6$) and ($s = 4$, $d = 8$) respectively.

3.3 Formal Specification of Beetle

Formal specification of Beetle is presented in Fig. 2. The algorithm takes an r-bit nonce N and a master key b-bit K. The master key is $K = K_1\|K_2$, where $|K_1| = r$ and $|K_2| = c$. N and K are first loaded into the initial state as $(N \oplus K_1)\|K_2$. The state is then processed using f and next the encryption algorithm takes non-empty A and non-empty M, and outputs C and T such that $|C| = |M|$ and $|T| = r$. The decryption algorithm takes (N, A, C, T) with $|A|, |C| \neq 0$ and outputs M or $\perp$.

The hash module is a variant of traditional sponge mode with an API suitable for using it in an authenticated encryption mode. It takes IV to initialize the state and data D. The other input parameters such as ℓ is used to initialize the counter for the internal state variables, const is used for domain separation for the associated data process and the message process, release is an indicator to output ciphertext/ message blocks along with the state value from hash and enc is used to denote whether we are invoking hash inside the encryption or decryption module. For associated data processing, we set release $= \mathbf{0}$ to denote that it is not releasing any output block, it only updates the state.

3.4 Security of Beetle

Here we state the main security theorem for Beetle:

```
Module hash($IV, D, \ell$, const, release, enc)
  1. $(D[1], \ldots, D[d]) \xleftarrow{r} D$
  2. $X[\ell] \| Z[\ell] \leftarrow IV$
  3. for $i = 1$ to $d$
  4.     $Y[\ell+i] \| Z[\ell+i+1] \leftarrow f(X[\ell+i-1] \| Z[\ell+i-1])$
  5.     If enc $= 1$ Then:
             $(X[\ell+i], O[i]) \leftarrow \rho(Y, D[i])$
  6.     Else:
             $(X[\ell+i], O[i]) \leftarrow \rho'(Y, D[i])$
  7. $Z[\ell+d] \leftarrow Z[\ell+d] \oplus \mathsf{const}$
  8. If release $= 0$ Then:
         Return $X[\ell+d] \| Z[\ell+d]$
  9. Else:
         Return $((O[1], \ldots, O[d]), X[\ell+d] \| Z[\ell+d])$

Module Proc-A($IV, A, \ell$, const)
  1. $X \| Z \leftarrow \mathsf{hash}(IV, A, \ell, \mathsf{const}, \mathbf{0}, \mathbf{1})$
  2. Return $X \| Z$

Module Proc-M($IV, M, \ell$, const)
  1. $(O, X \| Z) \leftarrow \mathsf{hash}(IV, M, \ell, \mathsf{const}, \mathbf{1}, \mathbf{1})$
  2. Return $(O, X \| Z)$

Module Proc-C($IV, C, \ell$, const)
  1. $(O, X \| Z) \leftarrow \mathsf{hash}(IV, C, \ell, \mathsf{const}, \mathbf{1}, \mathbf{0})$
  2. Return $(O, X \| Z)$
```

```
Algorithm Beetle-$\mathcal{E}_K(N, A, M)$
  1. $(A[1], \ldots, A[a]) \xleftarrow{r} A$
  2. $(M[1], \ldots, M[m]) \xleftarrow{r} M$
  3. $\mathsf{const}_A = \begin{cases} 1 & \text{if } |A[a]| < r \\ 2 & \text{otherwise.} \end{cases}$
  4. $\mathsf{const}_M = \begin{cases} 3 & \text{if } |M[m]| < r \\ 4 & \text{otherwise.} \end{cases}$
  5. $X[a] \| Z[a] \leftarrow \mathsf{Proc\text{-}A}(N \oplus K_1 \| K_2, \overline{A}, 0, \mathsf{Const}_A)$
  6. $(C, X[a+m] \| Z[a+m]) \leftarrow \mathsf{Proc\text{-}M}(X[a] \| Z[a], \overline{M}, a, \mathsf{Const}_M)$
  7. $T \leftarrow \mathsf{trunc}(f(X[a+m] \| Z[a+m]), r)$
  8. Return $C[1..|M|], T$

Algorithm Beetle-$\mathcal{D}_K(N, A, C, T)$
  1. $(A[1], \ldots, A[a]) \xleftarrow{r} A$
  2. $(C[1], \ldots, C[c]) \xleftarrow{r} C$
  3. $\mathsf{const}_A = \begin{cases} 1 & \text{if } |A[a]| < r \\ 2 & \text{otherwise.} \end{cases}$
  4. $\mathsf{const}_M = \begin{cases} 3 & \text{if } |M[m]| < r \\ 4 & \text{otherwise.} \end{cases}$
  5. $X[a] \| Z[a] \leftarrow \mathsf{Proc\text{-}A}(N \oplus K_1 \| K_2, \overline{A}, 0, \mathsf{Const}_A)$
  6. $(M, X[a+c] \| Z[a+c]) \leftarrow \mathsf{Proc\text{-}C}(X[a] \| Z[a], \overline{C}, a, \mathsf{Const}_M)$
  7. $T' \leftarrow \mathsf{trunc}(f(X[a+c] \| Z[a+c]), r)$
  8. If $T' = T$ Then:
         Return $M[1..|C|]$
  9. Else:
         Return $\perp$
```

Figure 2: The encryption and decryption algorithms of Beetle. Here $\forall i, |X[i]| = |Y[i]| = |O[i]| = r$ and $|Z[i]| = c$.

$$\mathbf{Adv}^{\mathrm{AE}}_{\mathsf{Beetle}}(q_e, q_p, q_v, \sigma_e, \sigma_v, t) \leq \frac{2(\sigma_e+\sigma_v)(\sigma_e+q_p)}{2^b} + \left(\frac{q_p}{2^{r-1}}\right)^r + \left(\frac{q_p^2}{2^{r+c-1}}\right)^r + \frac{r\sigma_v}{2^c} + \frac{q_v}{2^r}.$$

Based on the above theorem, Beetle[Light+] achieves 64-bit AE security and Beetle[Secure+] achieves 121-bit AE security. The proof of the above theorem is detailed in Sect. 4.

3.5 Features

The sponge based mode described above aims to achieve high security bound. This in turn makes the mode lightweight by minimizing the state size. We follow the approach of boosting the security by using a combined feedback technique over the traditional duplex sponge. The AE security level increases from $c/2$ to $c - \log r$. This in turns helps us to construct a scheme with the same security level but with a reduced state size. For example, Beetle[Secure+] achieves almost 128-bit security (121-bit security) with only a 256-bit state and $c = 128$. Beetle is a lightweight design (with small overheads from the tradition duplex sponge) that boosts the security without any restriction. Here we present a comparative study on the state size and security trade-off in Table 1.

Beetle also enjoys flexibility. It is easy to fit any permutation into this structure. This depicts that, when used with lighter permutations, it consumes lower hardware footprints. We can also play with r and c to make a proper trade off between the data absorption and the security level.

Table 1: Comparative Study on the State size and Security Trade-off: for all the constructions, we have assumed that $n/2$ bits message is processed per primitive call. Here SpongeAE refers to the AE mode using traditional duplex sponge mode.

Design	State size	Security
Beetle	$\mathbf{n}$	$\mathbf{n/2}-\log\mathbf{n/4}$
COFB	$(1.5n+k)/2$	$n/4 - \log n/2$
SpongeAE	n	$n/4$

We would like to point out that, we have concentrated only on the round-based implementation with n-bit data path and that essentially ensures that the state size to be n. However, we agree that a real serialized implementation could make it area efficient, but state size would grow up close to $1.5n$ ($b + r$ in general). In addition, as far we know, in principle we need n bits for state and $n/2$ bits for message buffer. However if we process message in one clock cycle we don't need to store the message buffer to an internal state. Note that this argument holds for duplex sponge, not imposed by the addition of feedback.

4 Formal Security Proof

In this section, we present the security analysis of Beetle, mainly we prove Theorem 2:

$$\mathbf{Adv}^{\mathrm{AE}}_{\mathsf{Beetle}}(q_e, q_p, q_v, \sigma_e, \sigma_v, t) \leq \frac{2(\sigma_e+\sigma_v)(\sigma_e+q_p)}{2^b} + \left(\frac{q_p}{2^{r-1}}\right)^r + \left(\frac{q_p^2}{2^{b-1}}\right)^r + \frac{r(q_p+\sigma_v)}{2^c} + \frac{q_v}{2^r}.$$

4.1 Notations and Set-up

Fix a deterministic non-repeating query making distinguisher adv that interacts with either (1) the real oracle (Beetle^f, f) or (2) the ideal oracle ($\$, f$) making at most

1. q_e encryption queries $(N_i, A_i, M_i)_{i=1..q_e}$ with an aggregate of total σ_e many blocks,

2. q_f offline or direct forward queries $(Q_i^+)_{i=1..q_f}$ to f,
3. q_b direct backward queries $(Q_i^-)_{i=1..q_b}$ to f^{-1} and
4. attempts to forge with q_v many queries $(N_i^*, A_i^*, C_i^*, T_i^*)_{i=1..q_v}$ having a total of σ_v many blocks.

We assume $q_p = q_f + q_b$ to be the total no of offline or direct queries. Also assume that, $\forall i$, M_i and A_i have m_i and a_i blocks respectively and C_i^* and A_i^* have c_i^* and a_i^* blocks respectively. We use the notation X^*, Z^*, Y^* and Z'^* to denote the intermediate variables corresponding to the forging queries.

4.2 Overview of the Attack Transcript

We begin with a description of the ideal oracle which consists of two phases. In the on line phase, for any encryption queries $(N_i, A_i = (A_i[1], \ldots, A_i[a_i]), M_i = (M_i[1], \ldots, M_i[m_i]))$, the oracle samples the cipher text blocks $C_i = (C_i[1], \ldots, C_i[m_i]) \leftarrow_\$ \{0,1\}^{r.m_i}$ and the tag $T_i \in \{0,1\}^r$ uniformly at random and returns it to $\mathcal{A}$. For any direct forward query Q_i^+ to f, the oracle returns $(L_i^+ \| R_i^+) := f(Q_i^+)$ uniformly at random. Here L_i^+ and R_i^+ denote the rate (most significant r bits) and capacity (least significant c bits) of the value $f(Q_i^+)$. Similarly for any direct backward query Q_i^- to f, the oracle returns $L_i^- \| R_i^- := f^{-1}(Q_i^-)$ uniformly at random. For any forging query $(N_i^*, A_i^*, C_i^*, T_i^*)$, the oracle returns $\perp$.

Offline Chain. We call that there exists a chain of sequence $(L_{i_1}, L_{i_2}, \ldots, L_{i_{\mu+1}})$, denoted by $\mathsf{Chain}(L_{i_1}, L_{i_2}, \ldots, L_{i_{\mu+1}})$ if there exists $R_{i_1}, \ldots, R_{i_{\mu+1}}$ such that the following chain is obtained via offline queries:

$$\begin{aligned}
f(L_{i_1} \| R_{i_1}) &= L'_{i_2} \| R_{i_2}, \\
f(\rho'_1(L'_{i_2}, L_{i_2}) \| R_{i_2}) &= L'_{i_3} \| R_{i_3}, \\
&\vdots \\
f(\rho'_1(L'_{i_{\mu-1}}, L_{i_{\mu-1}}) \| R_{i_{\mu-1}}) &= L'_{i_\mu} \| R_{i_\mu}, \\
f(\rho'_1(L'_{i_\mu}, L_{i_\mu}) \| (R_{i_\mu} \oplus \mathsf{const})) &= L_{i_{\mu+1}} \| R_{i_{\mu+1}},
\end{aligned}$$

We use the notation $\mathsf{mChain}(L_{i_1}, L_{i_2}, \ldots, L_{i_{\mu+1}})$ to denote number of $(R_{i_1}, R_{i_2}, \ldots, R_{i_{\mu+1}})$ for which the event $\mathsf{Chain}(L_{i_1}, L_{i_2}, \ldots, L_{i_{\mu+1}})$ occurs. We use the notation $\mathsf{Init}(L_{i_1}, L_{i_2}, \ldots, L_{i_{\mu+1}})$ to denote the set of R_{i_1} values for which $\mathsf{Chain}(L_{i_1}, L_{i_2}, \ldots, L_{i_{\mu+1}})$ occurs.

Next, in the offline phase (i.e. after $\mathcal{A}$ makes all the queries responses), it sets all the $X_i[a_i + k], Y_i[a_i + k]$ values: $Y_i[a_i + k] := M_i[k] \oplus C_i[k]$, $X_i[a_i + k] := \mathsf{shuffle}(M_i[k] \oplus C_i[k]) \oplus M_i[k]$, for all $k = 1, \ldots, m_i$. Then it samples $X_i[k]$, $Y_i[k]$, for all $k = 1, \ldots, a_i$ and the internal chaining values i.e. $Z_i[k]$, for all $k = 1, \ldots, a_i + m_i$ uniformly at random from $\{0,1\}^c$. For all $k \leq a_i^*$, the values $Y_i^*[k], Z_i^*[k]$ are set to $Y_j[k], Z_j[k]$ respectively, if $\exists j, N_i^* = N_j, A_i^*[1..k] = A_j[1..k]$. Similarly, for all $a_i^* + 1 \leq k \leq a_i^* + m_i^*$, the values $Y_i^*[k], Z_i^*[k]$ are set to $Y_j[k], Z_j[k]$ respectively, if $\exists j, N_i^* = N_j, A^* = A_j, C_i^*[1..k] = C_j[1..k]$. The value $X_i^*[k]$ is defined as $Y_i^*[k] \oplus A^*[k]$, for $k \leq a_i^*$. The value $X_i^*[a_i^* + k]$ is defined as $Y_i^*[a^* + k] \oplus M^*[k]$, for $k \leq m_i^*$. All the remaining Y^* values are sampled uniformly at random and corresponding X^* variables are set accordingly.

If $\mathcal{A}$ interacts with real oracle, we always have (i) $f(N_i \| K) = X_i[0] \| Z_i[0]$ and (ii) $f(X_i[j] \| Z_i[j-1]) = Y_i[j] \| Z_i[j]$, $i = 1..q_e, j = 1,.., a_i + m_i - 1$ and $f(X_i[a_i + m_i] \| Z_i[a_i + m_i]) = Y_i[a_i + m_i] \| Z_i[a_i + m_i] \oplus \mathsf{Const}_M$, where $X[i]$ and $Y[i]$s are computed via ρ.

Overall, the transcript of the adversary $\tau := (\tau_e, \tau_p, \tau_v)$ be the list of queries and responses of $\mathcal{A}$ that constitutes the query response transcript of $\mathcal{A}$, where

- $\tau_e = (N_i, A_i, M_i, C_i, Z_i, T_i)_{i=1..q_e}$,
- $\tau_p = (Q_i^+, f(Q_i^+))_{i=1..q_f} \cup (f^{-1}(Q_i^-), Q_i^-)_{i=1..q_b}$ and
- $\tau_v = (N_i^*, A_i^*, C_i^*, T_i^*, \perp), i = 1..q_v$.

We denote ip_{ideal} and ip_{real} the probability distribution of transcript τ induced by the ideal world and real world respectively.

4.3 Identifying and Bounding Bad Events

Now, we fix $\lambda > 0$, and define a set of events (initial bad events) for which the adversary aborts.

- B1: initColl. This event denotes that the initial state collides with a direct forward query or the response of a direct backward query, i.e.

$$\exists i, j \ (N_i \oplus K_1 \parallel K_2) \in \{Q_j^+, L_k^- \| R_k^-\}.$$

- B2: $\mathsf{mColl}(X) > \lambda$. This event signifies that λ multi-collision occurs in the rate part of the encryption queries. Formally, $\exists w \geq \lambda, i_1, i_2 \ldots, i_w$ such that

$$X_{i_1}[j_1] = X_{i_2}[j_2] = \cdots = X_{i_w}[j_w].$$

- B3: inpColl. This event denotes a state collision in the input of a permutation. This can happen when: (i) two input states to the permutation collides during the online (encryption) queries or (ii) an input state in the online (encryption) query collides with a direct forward query or the response of a direct backward query. Technically speaking, this event occurs when either

$$\exists i, j, i', j' \ni (X_i[j] \| Z_i[j]) = (X_{i'}[j'] \| Z_{i'}[j']) \text{ or } \exists i, j, k \ (X_i[j] \| Z_i[j]) \in \{Q_k^+, L_k^- \| R_k^-\}.$$

- B4: outColl. This event denotes a state collision in the output of a permutation. This can happen when: (i) two output states to the permutation collides during the online (encryption) queries or (ii) an output state in the online (encryption) query collides with a direct backward query i.e. either

$$\exists i, j, i', j' \ni (Y_i[j] \| Z_i[j+1]) = (X_{i'}[j'] \| Z_{i'}[j'+1]) \text{ or}$$
$$\exists i, j, k \ (Y_i[j] \| Z_i[j+1]) \in \{Q_k^-, L_k^+ \| R_k^+\}.$$

- B5: $\mathsf{mColl}(L^+) > \lambda$. This event signifies that λ multi-colllision occurs in the rate part of the direct permutation queries. Technically seen, this event can be written as: $\exists w \geq \lambda, i_1, i_2 \ldots, i_w$ such that

$$L_{i_1}^+ = L_{i_2}^+ = \cdots = L_{i_w}^+.$$

- B6: $\mathsf{mColl}(L^-) > \lambda$. This event signifies that λ multi-colllision occurs in the rate part of the direct inverse permutation queries. Technically seen, this event can be written as: $\exists w \geq \lambda, i_1, i_2 \ldots, i_w$ such that

$$L_{i_1}^- = L_{i_2}^- = \cdots = L_{i_w}^-.$$

- B7: $\mathsf{mMitM}(L^+, R^+, L^-, R^-) > \lambda$. This event signifies that λ Meet-in-the-Middle type collision occurs via the direct permutation and inverse permutation queries. More formally, the event is expressed as: $\exists w > \lambda, i_1, i_2 \ldots, i_w$ such that

$$\begin{aligned} \rho_1'(L_{i_1}^+, L_{j_1}^-) = \rho_1'(L_{i_2}^+, L_{j_2}^-) = \quad &\cdots \quad = \rho_1'(L_{i_\lambda}^+, L_{j_\lambda}^-), \\ R_{i_1}^+ = R_{j_1}^-, R_{i_2}^+ = R_{j_2}^-, \quad &\cdots \quad , R_{i_\lambda}^+ = R_{j_\lambda}^-. \end{aligned}$$

- B8: Forge. This event signifies that for some forging query, none of the states are fresh and the final tag also matches. Technically, $\exists\ i$ such that
  - $\forall j \leq (a_i^* + m_i^*),\ X_i^*[j] \| Z_i^*[j]$ is not fresh, and
  - $f(X_i^*[a_i^* + m_i^*] \| Z_i^*[a_i^* + m_i^*]) = T_i^* \| \star$.

The following lemma bounds the probability of bad transcripts in ideal oracle:

Lemma 1. *Let ϵ_{bad} denotes the probability of the event ($B1 \vee B2 \cdots \vee B8$). Then,*

$$\epsilon_{bad} \leq \frac{(\sigma_e q_p + \sigma_e^2 + \sigma_e \sigma_v + q_p \sigma_v)}{2^b} + \frac{\lambda q_p}{2^c} + \frac{q_p^\lambda}{2^{r(\lambda-1)}} + \frac{q_p^{2\lambda}}{2^{r(\lambda-1)+c\lambda}} + \frac{\lambda \sigma_v}{2^c}.$$

Proof. Here we provide the upper bounds for the bad events (in ideal oracle) one by one, as follows:

Bounding Pr[B1]. As the key K_1 and K_2 is chosen uniformly at random, for any fix i and j, we have

$$\Pr[(N_i \oplus K_1) \| K_2 \in \{Q_j^+, L_j^- \| R_j^-\}] = 2^{1-b}.$$

Now, varying over all choices of i and j, we have

$$\Pr[\mathsf{B1}] \leq \frac{q_e . q_p}{2^{b-1}}.$$

Bounding Pr[B2]. Fix $i_1, \ldots i_\lambda$. As, all the X values are (i) either sampled uniformly at random, or (ii) defined from C values which are sampled uniformly at random. Hence,

$$\Pr[X_{i_1}[j_1] = X_{i_2}[j_2] = \cdots = X_{i_\lambda}[j_\lambda]] \leq 2^{-r(\lambda-1)}.$$

Now, varying over all possible choices of $i_1, \ldots, i_\lambda$,

$$\Pr[\mathsf{B2}] \leq \frac{\binom{\sigma_e}{\lambda}}{2^{r(\lambda-1)}}.$$

Bounding Pr[B3]. This probability stands for the (i) input state collisions between two online queries or (ii) between one online query and one offline query. As all the $C_i[j]$ (and hence $X_i[j]$), $Z_i[j]$ values are sampled uniformly at random, for any fixed i, j, i', j', we have $\Pr[(X_i[j] \| Z_i[j]) = (X_{i'}[j'] \| Z_{i'}[j'])] = 2^{-b}$. On the other hand, for any fixed i, j, k, $\Pr[X_i[j] \| Z_i[j] \in \{Q_k^+, L_k^- \| R_k^-\}] = 2^{1-b}$ (if the offline query is made earlier) and $\Pr[X_i[j] \| Z_i[j] \in Q_k^+] = 2^{-c}$ (as adversary can choose the offline query keeping the rate part as $X_i[j]$). For the last case, given $\overline{B2}$, the number of choices for (i, j) can be at most λ. Now, combining all the cases and varying over all possible choices, we obtain

$$\Pr[\mathsf{B3}|\mathsf{B2}] \leq \frac{\binom{\sigma_e}{2}}{2^{b-1}} + \frac{\sigma_e . q_p}{2^{b-1}} + \frac{\lambda . q_p}{2^c}.$$

Bounding Pr[B4]. This probability stands for the output state collisions between two online queries or between one online query and one offline query. Again, with a similar logic as used in the previous case, one can show

$$\Pr[\mathsf{B4}|\mathsf{B2}] \leq \frac{\binom{\sigma_e}{2}}{2^{b-1}} + \frac{\sigma_e . q_p}{2^{b-1}} + \frac{\lambda . q_p}{2^c}.$$

Bounding Pr[B5]. Fix $i_1, \ldots i_\lambda$. As, all the L^+ values are sampled uniformly at random,

$$\Pr[L_{i_1}^+ = L_{i_2}^+ = \cdots = L_{i_\lambda}^+] \leq 2^{-r(\lambda-1)}.$$

Now, varying over all possible choices of $i_1, \dots, i_\lambda$,

$$\Pr[\mathsf{B5}] \le \frac{\binom{q_p}{\lambda}}{2^{r(\lambda-1)}}.$$

Bounding $\Pr[\mathsf{B6}]$. Fix $i_1, \dots i_\lambda$. As, all the L^- values are sampled uniformly at random,

$$\Pr[L^-_{i_1} = L^-_{i_2} = \cdots = L^-_{i_\lambda}] \le 2^{-r(\lambda-1)}.$$

Now, varying over all possible choices of $i_1, \dots, i_\lambda$,

$$\Pr[\mathsf{B6}] \le \frac{\binom{q_p}{\lambda}}{2^{r(\lambda-1)}}.$$

Bounding $\Pr[\mathsf{B7}]$. Fix $i_1, \dots i_\lambda$. As, all the L^+, L^- values are sampled uniformly at random,

$$\Pr[\rho'_1(L^+_{i_1}, L^-_{j_1}) = \cdots = \rho'_1(L^+_{i_\lambda}, L^-_{j_\lambda});\ R^+_{i_1} = R^-_{j_1}, \cdots, R^+_{i_\lambda} = R^-_{j_\lambda}] \le \frac{1}{2^{r(\lambda-1)}.2^{c\lambda}}.$$

Now, varying over all possible choices of $i_1, j_1, \dots, i_\lambda, j_\lambda$,

$$\Pr[\mathsf{B7}] \le \frac{\binom{q_p}{2\lambda}}{2^{r(\lambda-1)}.2^{c\lambda}}.$$

Bounding $\Pr[\mathsf{B8}]$. Recall that the event $\mathsf{B7}$ occurs if there exists a forging query $(N^*_i, A^*_i, C^*_i, T^*_i)$ such that $\forall j \le (a^*_i + m^*_i + 1)$, $X^*_i[j] \| Z^*_i[j]$ is not fresh, and $f(X^*_i[a^*_i + m^*_i] \| Z^*_i[a^*_i + m^*_i]) = T^*_i \| \star$. Now we consider the following cases:

CASE A. $\forall j,\ N^*_i \ne N_j$. The probability that the initial state $(N \oplus K_1 \| K_2)$ matches with any encryption or direct query state, can be bounded by probability $\frac{\sigma_e + q_p}{2^b}$.

CASE B. $N^*_i = N_j, A^*_i \ne A_j$. Let p be the common prefix block index of A^*_i and A_j i.e. $A^*_i[1..p] = A_j[1..p]$, $A^*_i[p+1] \ne A_j[p+1]$. The probability that the state $X^*[a^*_i + p + 1] \| Z^*[p+1]$ matches with any encryption or direct query state, can be bounded by probability $\frac{\sigma_e + q_p}{2^b}$.

CASE C. $(N^*_i, A^*_i) = (N_j, A_j)$. Let p be the common prefix of the corresponding ciphertexts i.e $C^*_i[1..p] = C_j[1..p]$, $C^*_i[p+1] \ne C_j[p+1]$. It is easy to see that $X^*_i[a^*_i + p + 1] = \rho'_1(C_j[p+1] \oplus M_j[p+1], C^*_i[p+1])$. Now, for $\mathsf{B8}$ to hold, we need

(i) $\mathsf{mChain}(X^*_i[a^*_i + p + 1], C^*_i[p+2], \dots, C^*_i[m^*_i], T^*) \ge 1$ and $Z^*_i[a^*_i + p + 1] \in \mathsf{Init}[X^*_i[p + 1], C^*_i[p+2], \dots, C^*_i[m^*_i], T^*]$, or

(ii) $\exists\ j, k;\ X^*_i[a^*_i + p + 1] = X_j[k]$ and $Z^*_i[a^*_i + p + 1] = Z_j[k]$

Now we make the following claim:

Claim. *mChain*$(X^*_i[a^*_i + p + 1], C^*_i[p+2], \dots, C^*_i[m^*_i], T^*) \le (m^*_i + 1)\lambda$*, given that* *B5, B6 and B7 have not occured.* [1]

Proof. Suppose the claim is not true. Now by simple application of Pigeon-hole principle, the above offine chain of length at most m^*_i necessarily gives rise to one of three events:

- There exist at least λ many offine chain with f queries only: this essentially forces a λ-multicollision in L^+ values (and the value is T^*), which is nothing but the event $\mathsf{B5}$.

[1]Note that, in the original sponge construction $\mathsf{SpongeAE}$, the above claim doesn't hold as $(m^*_i + 1)\lambda$ many collisions in an offine chain doesn't necessarily converges to λ-multicollisions there.

- There exist at least λ many offine chain with f^{-1} queries only: this forces a λ-multicollision in L^- values (and the value is $X_i^*[a_i^*+p+1]$), which is nothing but the event B6.
- There exists at least λ many offine chain with f and f^{-1} that meets in the i^{th} state: this forces a λ-multicollision of type MitM at the i^{th} state, which is is nothing but event B7.

Given the above claim holds, the probability that $Z_i^*[a_i^*+p+1] \in \mathsf{Init}(X_i^*[a_i^*+p+1], C_i^*[p+2], \ldots, C_i^*[m_i^*], T^*)$ is bounded by $\frac{(m_i^*+1)\lambda}{2^c}$. This is due to the uniform random sampling of the Z values. On the other hand, given that B2 hasn't occur, the probability that $Z_i^*[a_i^*+p+1] = Z_j[k]$ can be bounded by $\frac{\lambda}{2^c}$. Hence,

$$\begin{aligned}
\Pr[\mathsf{B8}|\overline{B2} \wedge \overline{B5} \wedge \overline{B6} \wedge \overline{B7}] &\leq \sum_{i=1}^{q_v} \Big(\frac{\sigma_e + q_p}{2^b} + \frac{(m_i^*+2)\lambda}{2^c} \Big) \\
&\leq \frac{(\sigma_e + q_p).q_v}{2^b} + \frac{\sigma_v \lambda}{2^c}.
\end{aligned}$$

The lemma follows as we sum all the probabilities. □

4.4 Analysis of Good Transcripts

Having defined and bounded the probability of realizing bad transcript in ideal world, it only remains to lower bound the ratio of real and ideal interpolation probability for a good transcript. Fix a good attainable transcript $\tau := (\tau_e, \tau_p, \bot_{\mathsf{all}})$ where $\bot_{\mathsf{all}}$ is used to represent that all the verification queries output is $\bot$. It is easy to see that,

$$\begin{aligned}
ip_{\mathsf{ideal}} = \mathsf{Pr}_{\mathsf{ideal}}[\tau] &= \mathsf{Pr}_{\mathsf{ideal}}[\tau_e, \tau_p, \bot_{\mathsf{all}}] \\
&= \mathsf{Pr}_{\mathsf{ideal}}[\tau_e].\mathsf{Pr}_{\mathsf{ideal}}[\tau_p].\mathsf{Pr}_{\mathsf{ideal}}[\bot_{\mathsf{all}}] \\
&= \frac{1}{(2^b)^{\sigma_e+q_e+q_p}}.
\end{aligned}$$

Now, we consider the probability of the transcript in real oracle. By definition,

$$\begin{aligned}
ip_{\mathsf{real}} = \mathsf{Pr}_{\mathsf{real}}[\tau] &= \mathsf{Pr}_{\mathsf{real}}[\tau_e, \tau_p, \bot_{\mathsf{all}}] \\
&\geq \mathsf{Pr}_{\mathsf{real}}[\tau_e, \tau_p] - \sum_i \mathsf{Pr}_{\mathsf{real}}[\tau_e, \tau_p, \top_i] \\
&= \mathsf{Pr}_{\mathsf{real}}[\tau_e, \tau_p] - \mathsf{Pr}_{\mathsf{real}}[\tau_e, \tau_p] \sum_i \mathsf{Pr}[\top_i | \tau_e, \tau_p] \\
&= \mathsf{Pr}_{\mathsf{real}}[\tau_e, \tau_p](1 - \epsilon_{\mathsf{ratio}}) \\
&= \frac{1}{\mathbf{P}(2^b, (\sigma_e + q_e + q_p))}.(1 - \epsilon_{\mathsf{ratio}}) \\
&\geq ip_{\mathsf{ideal}}.(1 - \epsilon_{\mathsf{ratio}}),
\end{aligned}$$

where $\epsilon_{\mathsf{ratio}} = \sum_i \epsilon_i$, with $\epsilon_i = \mathsf{Pr}[\top_i | \tau_e, \tau_p]$.

Calculation of $\epsilon_{\mathsf{ratio}}$. Now we calculate $\epsilon_{\mathsf{ratio}}$ by bounding ϵ_i. Since τ is a good transcript, and hence the event B_8 doesn't hold, either (i) $\exists j^* : (X_i^*[j^*] \| Z_i^*[j^*])$ is fresh or (ii) the event $\top_i$ holds with probability 0. For case (ii), we trivially have $\epsilon_i = 0$. So, we consider case (i). Now we have the following observations:

- If $(X_i^*[j^*] \| Z_i^*[j^*])$ is fresh, then permutation f and the ρ function ensures that $(X_i^*[j^*+1] \| Z_i^*[j^*+1])$ would be non-fresh with probability at most $\frac{(\sigma_e+\sigma_p)}{2^b}$. This is due to the fact that usage of the ρ function restricts the adversary to have any control over $X_i^*[j^*+1]$ if the value $Y_i^*[j^*]$ is random. This is in contrast to the original sponge construction SpongeAE, where the adversary can always control the $X_i^*[j^*+1]$ using $C_i^*[j^*+1]$ irrespective of the value of $Y_i^*[j^*]$.

- Extending this argument, we claim that the sequence of states any one of the following states $(X_i^*[j_i^*] \| Z_i^*[j_i^*]), (X_i^*[j_i^*+1] \| Z_i^*[j_i^*+1]), \ldots, (X_i^*[m_i^*] \| Z_i^*[m_i^*])$ will be non-fresh with probability at most $\frac{m_i^*(\sigma_e+\sigma_p)}{2^b}$.
- It is easy to see that, if $(X_i^*[m_i^*] \| Z_i^*[m_i^*])$ is fresh, then
$$\Pr[f(X_i^*[m_i^*] \| Z_i^*[m_i^*]) = T_i \| \star] = \frac{1}{2^r}.$$

Putting everything together,
$$\epsilon_i \leq \frac{1}{2^r} + \frac{m_i^*(\sigma_e+q_p)}{2^b}.$$

Hence, we bound $\epsilon_{\mathsf{ratio}}$ as:
$$\epsilon_{\mathsf{ratio}} = \sum_{i=1}^{q_v} \epsilon_i \;\leq\; \sum_{i=1}^{q_v} \frac{1}{2^r} + \frac{m_i^*(\sigma_e+q_p)}{2^b} \;\leq\; \frac{q_v}{2^r} + \frac{\sigma_v.(\sigma_e+q_p)}{2^b}.$$

4.5 Combining Everything Together and Use Patarin's H-Coeffcient

Applying Theorem 1 (Patarin's H-Coeffcient technique) with $\epsilon_{\mathsf{bad}} = \frac{(\sigma_e q_p + \sigma_e^2 + \sigma_e\sigma_v + q_p\sigma_v)}{2^b} + \frac{rq_p}{2^c} + (\frac{q_p}{2^{r-1}})^r + (\frac{q_p^2}{2^{(r-1)+c}})^r + \frac{r\sigma_v}{2^c}$ (putting $\lambda = r$) and $\epsilon_{\mathsf{ratio}} = \frac{q_v}{2^r} + \frac{\sigma_v.(\sigma_e+q_p)}{2^b}$, the result of Theorem 2 follows. $\square$

5 Hardware Implementation of Beetle

5.1 Overview

Beetle primarily aims to be implemented in small hardware devices. In several applications, the hardware resource for implementation plays an important role and the implementation memory dominates in the total hardware resource and parallelization of the internal modules for scalability are not needed. For this purpose, a small state size and completely serial implementation is more desirable.

From implementation perspective, Beetle has a simple structure since it consists of a permutation and a few basic operations (such as bitwise XORs with simple combined feedback computations). Beetle uses a small sized state (only the permutation state) and the implementation area of Beetle is largely dominated by the underlying permutation.

In this paper, we only describe hardware architecture of Beetle[Light+]. Hardware architecture for Beetle[Secure+] is similar (only differences are the bit lengths of the variables) and can be easily followed. We provide our hardware implementation results on both Virtex 6 and Virtex 7 under Xilinx 13.4. We first provide a brief analysis on the clock cycles required to process the input bytes. This is a conventional way to estimate the speed. Beetle[Light+] gets an a-block associated data and an m-block message and needs $13(a+m)+12$ cycles (13 for each of the data blocks and 12 for the first permutation call for initialization). Note that, when the data (associated data or message) is empty, the algorithm processes one 0^r block and it takes $25 = 13+12$ clock cycles. Table 2 shows the number of average cycles per input message bytes, which we call cycles per byte (cpb), assuming the associated data has the same length as message (i.e, $a = m$) and rate is 64 bits or 8 bytes. That is, the *cpb* is $(13 \cdot 2m + 12)/8m = 3.25 + 1.5/m$. Clearly, when m is large then *cpb* converges to 3.25.

Table 2: Clock cycles per message byte for Beetle[Light+] with $r = 64$.

	Message length (Bytes)										
	8	16	24	32	64	128	256	512	1024	2048	16384
cpb	3.4375	3.3437	3.3125	3.2969	3.2734	3.2617	3.2559	3.2529	3.2514	3.2507	$\approx$ 3.2500

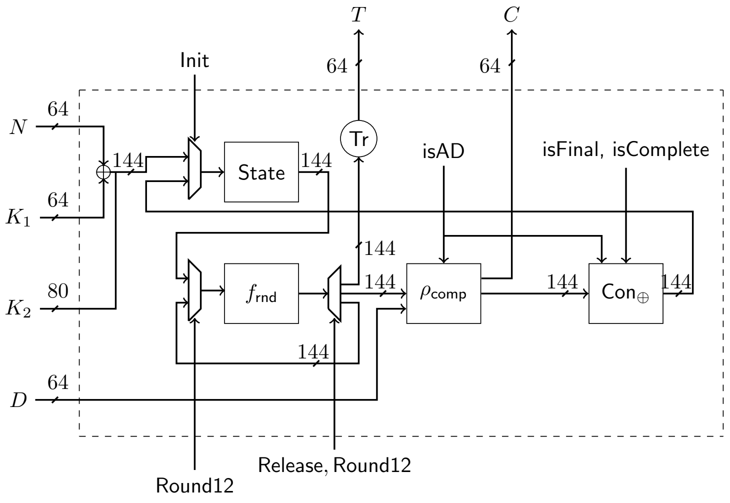

Figure 3: Hardware circuit diagram for Beetle[Light+]

5.2 Hardware Architecture

We implement both instantiations and report the results. They are base implementations without any pipe lining. We mainly focus on the encryption only as the combined encryption-decryption circuit has very small overhead in hardware area. This is mainly due to the fact that the decryption algorithm does not need inverse permutation routine.

We use the same hardware for both the associated data and message processing phase as they have similar computations. Only a single bit switch for the two types of input data is required to distinguish. The main modules of the architecture are described below. We also describe the finite state machine (FSM) that controls the circuit flow by setting up and updating internal signals and sending them to the internal modules. The FSM has a simple flow structure. The overall hardware architecture for the lighter version is described in Fig. 3. The description of the architecture is based on Beetle[Light+] only.

1. **State Register.** The architecture for Beetle[Light+] contains only one state register State. This register is used to store intermediate variables after each iteration. We use an 144-bit state register for the permutation. We don't need any register for the key, as it is used only to initialize the state. The hardware circuit in Fig. 3 shows that State is updated after each round of f executed by f_{rnd} module.

2. **Module f_{rnd}.** f_{rnd} computes one round of f. It takes an 144-bit input from the state register, computes one round of f and then either updates the state or send the output to the ρ computation module ρ_{comp} or releases the output as the tag. The entire operation is serial and we need 12 clock cycles with 12 f_{rnd} executions to execute f.

3. **Module ρ_{comp}.** ρ_{comp} module computes ρ on a 64-bit data block and a 144-bit

intermediate state (output from the f computation). The output is a 144-bit feedback value (passed to $\mathsf{Con}_\oplus$ module to perform a constant addition in the capacity part).

4. **Module** $\mathsf{Con}_\oplus$. This module executes a constant addition (addition of the constant **1/2/3/4**) based on the signals it receives. The signals are described below.

5. **Data Signals**. The hardware circuit uses several internal data signals controlled by the finite state machine (FSM). The circuit uses the following signals.

   - Start: This signal indicates the start of the computations of the circuit. It actually resets all the variables and sets the control to the reset state of the circuit.
   - Init: This signal indicates that the initialization is done and the circuit can now process data blocks.
   - isAD: This signal indicates whether the current block is an associated data block or a message block. It actually controls the feedback computation module as the associated data processing phase does not release any ciphertext block and only computes the next feedback, where as the message processing phase computes the next feedback as well as releases the ciphertext block. This signal is also used with isFinal signal to decide which constant addition.
   - isFinal: This signal indicates whether the current data block is the final block or not. If it is the final block then the constant addition module becomes active. It is used to control the $\mathsf{Con}_\oplus$ module. It uses an internal demultiplexer that decides whether to pass the data to the state register or add a constant to it. This decision is taken using the isFinal signal.
   - isComplete: This signal together with isAD decide which constant to add as they differ by the completeness of the last block and type of the data block.
   - Round12: This signal indicates whether f_{rnd} is executing its last round. If yes then ρ computation module will start executing, otherwise the control returns to the f_{rnd} module.
   - Release: This signal indicates the release of the tag. After the tag release, the circuit ends its functionalities.

   These signals are actually generated and controlled by a controller module, which can be viewed as a finite state machine (FSM). For the sake of simplicity, we intentionally omit the description of FSM in Fig. 3. We describe the FSM separately in Fig. 4. The description of the FSM is given below.

6. **FSM**. This module controls the whole circuit. It generates and sends signals to different modules and divides the functionalities of the circuit into several states. This is depicted in Fig. 4. The individual states are described below. Note that, the signal isComplete is implicitly taken care by the controller and for the sake of simplicity, it does not appear in Fig. 4.

   - Reset: Starts the functionality of the circuit. It denotes everything is reset and the circuit will begin to work now.
   - Load: Initialize the permutation state where the secret key and the nonce are loaded into the permutation state. The control next enters into invocation of f. This is denoted by $\mathsf{Init} = 1$.
   - Invoke_f: Invocation of f. This state is further described by four states .
     - Reset_f: Resets the permutation state before the execution.

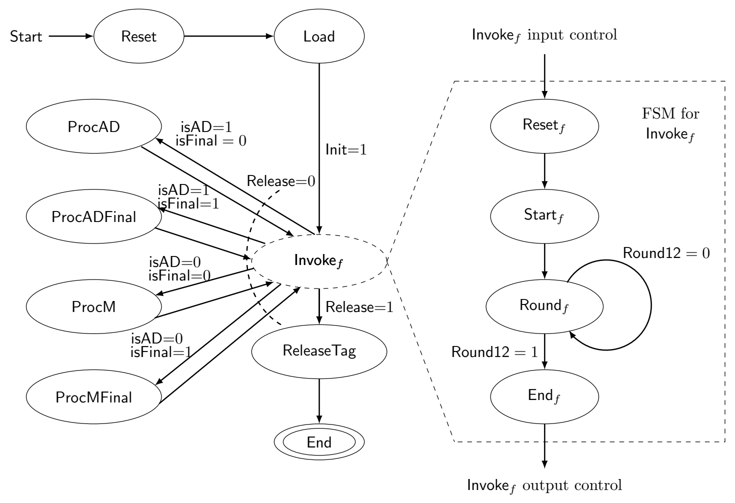

Figure 4: Finite State Machine

- Start$_f$: Start of the permutation execution. It actually loads the permutation state with the current internal value.
- Round$_f$: f round computations. The control remains in the same state until the control reaches the last round. When Round12 becomes 1, the control goes to the end of the permutation invocation.
- End$_f$: Indicates the end of the permutation execution and the control comes out of the permutation module.

If Release $= 1$, control goes to the ReleaseTag state, else to the following four stages.

- ProcAD: The circuit is now processing an associated data block but not the final block. This information is passed to the circuit by isAD $= 1$ and isFinal $= 0$.
- ProcADFinal: The circuit is now processing the final associated data block. This information is passed to the circuit by isAD $= 1$ and isFinal $= 1$.
- ProcM: The circuit is now processing a message block but not the final block. This information is passed to the circuit by isAD $= 0$ and isFinal $= 0$.
- ProcMFinal: The circuit is now processing the final message block. This information is passed to the circuit by isAD $= 0$ and isFinal $= 1$.
- ReleaseTag: The circuit releases the tag and the control enters into the End state to signify the end of the computations.
- End: This state signifies the end of the computations by the circuit.

5.3 Basic Implementation

We describe a basic flow of our implementation of Beetle[Light+], which generally follows the pseudo code of Fig. 2. During the initialization, **State** register is loaded with $(N \oplus K_1)\|K_2$. Next, the initialization is started by processing the associated data blocks. After each of the f invocations the output is processed with data blocks to compute the next feedback. During the final associated data block processing, one constant is added to the capacity part for domain separation. The message is processed in the same way as the associated data, except the circuit releases one ciphertext block at each clock cycle and also different constant is added when the final message block is processed. Finally, after the message is processed the tag is released by truncating the output of the final f execution.

Remark 1. (Combined Encryption and Decryption) As mentioned earlier, we have focus on the encryption-only circuit. However, due to the similarity between the encryption and the decryption modes, the combined hardware for encryption and decryption can be built with a small increase in the area, with the same throughput. This can be done by adding a control flow to a binary signal for mode selection.

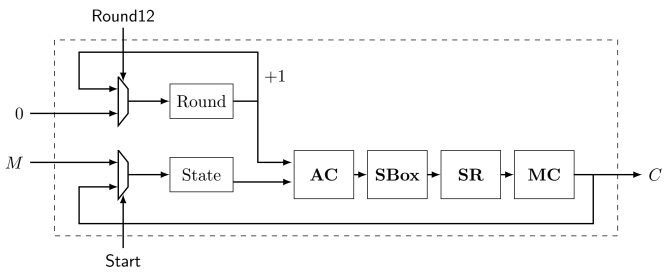

Figure 5: Hardware circuit diagram for Round Based Implementation of f

5.4 Hardware Implementation of PHOTON

In this section, we briefly describe our own round based hardware implementation of the P_{144} permutation [20, 21] (our choice of f). The architecture described in Fig. 5 follows a simple base implementation. The FSM for the architecture has been depicted in Fig. 4. We intentionally remove the control unit from Fig. 5 for the sake of simplicity. The f module receives 144-bit (18 bytes) plaintext data as input and processes it in 12 cycles (one clock cycle for each of the 12 rounds). Hence, *cpb* for f is $18/12 = 1.5$. The circuit maintains a 144-bit state register which is first loaded with the input and then gets updated after each round. It also maintains a 4-bit register *Round* which stores the current round number. It is initialized with 0 and is incremented by one after each round. After the permutation executes all the rounds, the register is again reset to 0. As mentioned already, the architecture computes one permutation round in one clock cycle and the round function consists of 4 sequential submodules AC, SBox, SR and MC. AC is the first module of the that adds a round dependent constant to the first column of the intermediate state. It takes two inputs, the intermediate state and the current round number. Next, the SBox module applies a non linear substitution to each of the nibbles (4-bit) of the state. The SR module shifts each of the rows in the state and finally the MC module multiplies the state with a serial matrix 6 times successively (multiplying once is denoted by MCS), such that altogether it becomes multiplication by an MDS matrix. We implement this version on Virtex 6 (target device xc6vlx760) and Virtex 7 (target device xc7vx415t). We use the RTL approach and a basic iterative type architecture. We would like to emphasize that our implementation is round based and it uses 144-bit data path.

We have also implemented P_{256} [20, 21] on the same platforms and and using the same approach. The AC, $SBox$ SR and MC operations are similar except they use larger state (8×8 matrix with 4-bit cells). The permutation consumes 256-bit (32 bytes) input and processes it in 12 cycles (12 rounds in 12 clock cycles). Hence, *cpb* is $32/12 = 2.67$. In Sect. 5.5 below we report the overall hardware implementation results for Beetle[Light+] and Beetle[Secure+].

Remark 2. We would like to point out that a possible way of optimization is to have serial implementation with smaller data paths (e.g. 4 or 24 bits) as used in [4, 5]. It seems that such implementation could make the construction even more area efficient, though the state size would increase by 64 and 128 bits for Beetle[Light+] and Beetle[Secure+] respectively. This is due to the fact that we need to store the message buffer to an internal state. This has nothing to do with the introduced feedback, and holds in general for any duplex sponge mode.

5.5 Implementation Results

The hardware architecture of Beetle[Light+] is programmed in VHDL language and is implemented on the same Virtex 6 (target device xc6vlx760) and Virtex 7 (target device xc7vx415t) under Xilinx 13.4. We use the same RTL approach and a basic iterative type architecture. The areas are listed in terms of the number of Slice Registers, Slice LUTs and Occupied Slices. The number of slice registers, LUTs and slices are 185, 616 and 252 respectively on Virtex 6 and the same on Virtex 7 are 185, 608 and 312 respectively. The frequencies reported are 381.592 MHZ on Virtex 6 and 425.595 MHZ on Virtex 7 where 13 (12 for f and one for ρ computation) cycles are required to process one 64 bit data block. Thus, the throughputs for long messages on Virtex 6 and 7 are 1.879 and 2.211 Gbps respectively. However we also calculate the area efficiency metric. The detailed hardware results are presented in Table 3 below.

Table 3: Beetle[Light+] Implemented FPGA Results

Platform	**# Slice Registers**	**# LUTs**	**# Slices**	**Frequency (MHZ)**	**Gbps**	**Mbps/ LUT**	**Mbps/ Slice**
Vertex 6	185	616	252	381.592	1.879	3.050	7.369
Virtex 7	185	608	312	425.595	2.095	3.445	6.715

We also report the hardware implementation results for Beetle[Secure+] on the same platform using the same approach. The detailed results are described in Table 4 below.

Table 4: Beetle[Secure+] Implemented FPGA Results

Platform	**# Slice Registers**	**# LUTs**	**# Slices**	**Frequency (MHZ)**	**Gbps**	**Mbps/ LUT**	**Mbps/ Slice**
Vertex 6	281	998	434	256.000	2.520	2.525	5.806
Vertex 7	305	1101	512	303.965	2.993	2.718	5.846

5.6 Component Wise Hardware Area Calculation for lightweight Beetle[Light+]

The architecture of lightweight Beetle[Light+] consists of several modules. In this section, we provide a brief description of the distribution of hardware area among the modules. The main modules in this circuit are f, ρ_{comp} and the control unit. The underlying register *State* is used by f. The hardware area utilizations (in Virtex 6) by different modules are presented in Fig. 6. It has been observed that, the majority of the hardware footprint is used by the underlying f permutation and rest of the circuit uses a few lightweight operations. The orange part denoted by "Others" consists of some additional operations like key and nonce load, state updates, constant additions etc.

5.7 Benchmarking Beetle[Light+]

We benchmark hardware implementation results of Beetle[Light+] using a few of the Athena listed implementations along with the implementation results in [2, 1] on both Virtex 6

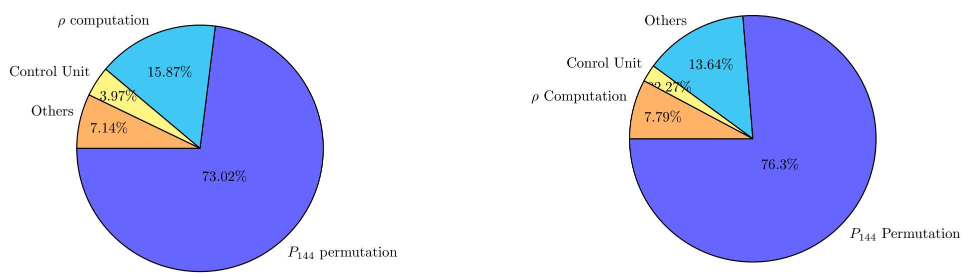

Figure 6: Distribution of Components by #Slices (left) and #LUTs (right)

and 7. We would like to mention that this is a rough benchmark. Our implementation ignores the overhead associated with the CAESAR API (an updated version of GMU hardware API) and also this is an encryption only circuit while most of the others support both encryption-decryption. We know that the GMU hardware API used in the SHA-3 competition hardware benchmarking, can cause 25% overhead in terms of the area compared to other interfaces they have provided [25, 26]. Nevertheless, our current implementation results for Beetle[Light+] depict that it consumes lower hardware footprints and provides highly competitive results than other constructions even if we add the overhead for supporting GMU API and decryption circuit. We have chosen the candidates for the benchmark by the following criteria

- CAESAR or non CAESAR lightweight Blockcipher based AE schemes and
- Sponge based AE schemes with smaller permutation size.

We observe that Beetle[Light+] occupies the lowest hardware area among the listed implementations. it occupies much lower hardware footprint than the closest competitors: COFB-AES, JAMBU-SIMON96 and Ketje-JR etc. and also achieves a better throughput/area metric. These two benchmarks depicts that Beetle[Light+] is one of the best candidates for lightweight applications. The benchmark is detailed in Table 5 and 6. The hardware implementation results of COFB-AES have been taken from [18, 17] and that of the other schemes have been taken from the ATHENA database [2, 1].

Table 5: Benchmarking Beetle[Light+] on Virtex 6. Sponge(b, r) denotes the scheme follows sponge mode with rate r and uses a b-bit permutation. BC(n) denotes the scheme follows a blockcipher based mode with an n-bit blockcipher.

Scheme	**Underlying Primitive**	**Security (in Bits)**	**# LUTs**	**# Slices**	**Gbps**	**Mbps/ LUT**	**Mbps/ Slice**
Beetle[Light+]	**Sponge(144, 64)**	64	**616**	**252**	**1.879**	**3.050**	**7.369**
Ketje-JR [15]	Sponge(200, 16)	96	1236	412	2.832	2.292	6.875
ASCON-128 [19]	Sponge(320, 64)	128	1274	451	3.118	2.447	6.914
JAMBU-SIMON96 [32]	BC(64)	48	1035	386	0.931	0.899	2.411
CLOC-TWINE80 [22]	BC(80)	32	1689	532	0.343	0.203	0.645
SILC-LED80 [23]	BC(80)	32	1684	579	0.245	0.145	0.422
SILC-PRESENT80 [23]	BC(80)	32	1514	548	0.407	0.269	0.743
COFB-AES [17, 18]	BC(128)	58	1075	442	2.850	2.240	6.450

Table 6: Benchmarking Beetle[Light+] on Virtex 7.

Scheme	# LUTs	# Slices	Gbps	Mbps/ LUT	Mbps/ Slice
Beetle[Light+]	**608**	**312**	**2.095**	**3.445**	**6.715**
Ketje-JR	1567	518	4.080	2.604	7.876
ASCON-128	1557	432	4.059	2.607	9.396
JAMBU-SIMON96	1376	387	0.938	0.682	2.423
CLOC-TWINE80	1552	439	0.432	0.279	0.985
SILC-LED80	1682	524	0.267	0.159	0.510
SILC-PRESENT80	1514	484	0.479	0.316	0.990
COFB-AES	1456	55	2.820	2.220	5.080

5.8 Benchmarking Beetle[Secure+]

We also benchmark hardware implementation results of Beetle[Secure+] using a few of the Athena listed implementations along with the implementation results in [2, 1] on both Virtex 6 and 7. We have chosen the candidates for the benchmark by the following criteria

- Popular Sponge based CAESAR candidates with higher security level (available in [2, 1]).

We observe that Beetle[Secure+] occupies the lowest hardware area among the listed implementations. In fact, it occupies lower hardware footprint than the closest competitors: ASCON-128, Hanuman and Ketje-JR. This benchmark is detailed in Table 7 and 8.

Table 7: Benchmarking Beetle[Secure+] on Virtex 6.

Scheme	Underlying Primitive	Security (in Bits)	# LUTs	# Slices	Gbps	Mbps/ LUT	Mbps/ Slice
Beetle[Secure+]	**Sponge(256, 128)**	**121**	**998**	**434**	**2.520**	**2.525**	**5.806**
ASCON-128 [19]	Sponge(320, 64)	128	1274	451	3.118	2.447	6.914
NORX [8]	Sponge(1024, 768)	128	5495	1724	24.524	4.463	9.139
Ketje-SR [15]	Sponge(400, 32)	128	1903	613	5.772	3.033	9.416
Riverkeyak [14]	Sponge(800, 544)	128	6234	1751	7.417	1.190	4.236
Lakekeyak [14]	Sponge(1600, 1344)	128	19860	7130	12.603	0.635	1.768
Gibbon [6]	Sponge(280, 40)	120	1807	653	1.280	0.708	1.960
Hanuman [6]	Sponge(280, 40)	120	1769	626	0.693	0.392	1.107
ICEPOLE128a [28]	Sponge(1280, 1024)	128	5734	1995	44.464	7.754	22.288

Table 8: Benchmarking Beetle[Secure+] on Virtex 7. We use U to denote that the corresponding results are unavailable.

Scheme	# LUTs	# Slices	Gbps	Mbps/ LUT	Mbps/ Slice
Beetle[Secure+]	**1101**	**512**	**2.993**	**2.718**	**5.846**
ASCON-128	1557	432	4.080	2.604	7.876
NORX	7877	2088	19.712	2.502	9.441
Ketje-SR	2592	724	6.752	2.605	9.326
Riverkeyak	8169	U	8.704	1.065	U
Lakekeyak	18581	4877	16.672	0.897	3.418
Gibbon	1894	600	1.169	0.617	1.948
Hanuman	1829	595	0.654	0.358	1.099
ICEPOLE128a	5733	1742	37.461	6.534	21.505

6 Conclusion

This paper presents Beetle, a sponge mode for AE focusing on the state size as well as optimizing the security. It is instantiated with two versions, where the first version

Beetle[Light+] aims to be lightweight and the second version Beetle[Secure+] aims to be highly secure, yet lightweight (though much heavier than Beetle[Light+]). When instantiated with a b-bit blockcipher, Beetle operates at a rate r ($b = 144$, $r = 64$ for Beetle[Light+] and $b = 256$, $r = 128$ for Beetle[Secure+]) and requires state size of b bits, and is provable secure up to $min\{c - \log r, b/2, r\}$ queries under the ideal permutation model. To be precise, Beetle[Light+] has 64-bit security and Beetle[Secure+] has 121-bit security. The key idea of Beetle is a feedback function combining both plaintext and ciphertext blocks. We have also presented the hardware implementation results, which demonstrate the effectiveness of our approach.

References

[1] ATHENa: Automated Tool for Hardware Evaluation. https://cryptography.gmu.edu/athena/.

[2] Authenticated Encryption FPGA Ranking. https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/rankings_view.

[3] CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. http://competitions.cr.yp.to/caesar.html/.

[4] N. Nalla Anandakumar, Thomas Peyrin, and Axel Poschmann. A very compact FPGA implementation of LED and PHOTON. *IACR Cryptology ePrint Archive*, 2014:738, 2014.

[5] N. Nalla Anandakumar, Thomas Peyrin, and Axel Poschmann. A very compact FPGA implementation of LED and PHOTON. In Willi Meier and Debdeep Mukhopadhyay, editors, *Progress in Cryptology - INDOCRYPT 2014 - 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings*, volume 8885 of *Lecture Notes in Computer Science*, pages 304–321. Springer, 2014.

[6] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Florian Mendel, Bart Mennink, Nicky Mouha, Qingju Wang, and Kan Yasuda. PRIMATEs v1.02. Submission to CAESAR. 2016. https://competitions.cr.yp.to/round2/primatesv102.pdf.

[7] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010.

[8] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. NORX v3.0. Submission to CAESAR. 2016. https://competitions.cr.yp.to/round3/norxv30.pdf.

[9] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. *IACR Cryptology ePrint Archive*, 2011:499, 2011.

[11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer, 2011.

[12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013.

[13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the security of the keyed sponge construction. In *Symmetric Key Encryption Workshop*, 2011.

[14] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. CAESAR submission: Keyak v2. Submission to CAESAR. 2016. https://competitions.cr.yp.to/round3/keyakv22.pdf.

[15] Guido Bertoni, Michaël Peeters Joan Daemen, Gilles Van Assche, and Ronny Van Keer. Ketje v2. Submission to CAESAR. 2016. https://competitions.cr.yp.to/round3/ketjev2.pdf.

[16] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. spongent: A lightweight hash function. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *LNCS*, pages 312–325. Springer, 2011.

[17] Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, and Mridul Nandi. Blockcipher-based authenticated encryption: How small can we go? *IACR Cryptology ePrint Archive*, 2017:649, 2017.

[18] Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, and Mridul Nandi. Blockcipher-based authenticated encryption: How small can we go? In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 277–298. Springer, 2017.

[19] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to CAESAR. 2016. https://competitions.cr.yp.to/round3/asconv12.pdf.

[20] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011.

[21] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. *IACR Cryptology ePrint Archive*, 2011:609, 2011.

[22] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, Sumio Morioka, and Eita Kobayashi. CAESAR Candidate CLOC. DIAC 2014.

[23] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, Sumio Morioka, and Eita Kobayashi. CAESAR Candidate SILC. DIAC 2014.

[24] Philipp Jovanovic, Atul Luykx, and Bart Mennink. Beyond 2 c/2 security in sponge-based authenticated encryption modes. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 85–104. Springer, 2014.

[25] B. Jungk and M. Stttinger. Hobbit: Smaller but faster than a dwarf: Revisiting lightweight SHA-3 FPGA implementations. pages 1–7.

[26] Sachin Kumar, Jawad Haj-Yihia, Mustafa Khairallah, and Anupam Chattopadhyay. A comprehensive performance analysis of hardware implementations of CAESAR candidates. *IACR Cryptology ePrint Archive*, 2017:1261, 2017.

[27] Kerry A. McKay, Larry Bassham, Meltem Sönmez Turan, and Nicky Mouha. Report on Lightweight Cryptography. 2017. http://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf.

[28] PawełMorawiecki, Kris Gaj, Ekawat Homsirikamol, Krystian Matusiewicz, Josef Pieprzyk, Marcin Rogawski, Marian Srebrny, and Marcin Wójcik. ICEPOLE v2. Submission to CAESAR. 2016. https://competitions.cr.yp.to/round2/icepolev2.pdf.

[29] NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. FIPS PUB 202, 2015.

[30] J. Patarin. Etude des Générateurs de Permutations Basés sur le Schéma du D.E.S. Phd Thèsis de Doctorat de l'Université de Paris 6, 1991.

[31] Ronald L. Rivest and Jacob C. N. Schuldt. Spritz - a spongy rc4-like stream cipher and hash function. *IACR Cryptology ePrint Archive*, 2016:856, 2016.

[32] Hongjun Wu and Tao Huang. The JAMBU Lightweight Authentication Encryption Mode (v2.1). Submission to CAESAR. 2016. https://competitions.cr.yp.to/round3/jambuv21.pdf.