The POET Family of On-Line Authenticated Encryption Schemes

Submission to the CAESAR competition

Version 1.02 – March 17, 2014

Farzaneh Abed	Bauhaus-Universität Weimar, farzaneh.abed(at)uni-weimar.de
Scott Fluhrer	Cisco Systems, sfluhrer(at)cisco.com
John Foley	Cisco Systems , foleyj(at)cisco.com
Christian Forler[1]	Bauhaus-Universität Weimar, christian.forler(at)uni-weimar.de
Eik List	Bauhaus-Universität Weimar, eik.list(at)uni-weimar.de
Stefan Lucks[2]	Bauhaus-Universität Weimar, stefan.lucks(at)uni-weimar.de
David McGrew	Cisco Systems, mcgrewd(at)cisco.com
Jakob Wenzel	Bauhaus-Universität Weimar, jakob.wenzel(at)uni-weimar.de

[1]The research leading to these results received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreement no. 307952.

[2]A part of this research was done while Stefan Lucks was visiting the National Institute of Standards and Technologies (NIST), during his sabbatical.

"I would define, in brief, the poetry of words as the rhythmical creation of beauty."

– *Edgar Allen Poe (The* Poet*ic Principle)*

Executive Summary

There is a compelling need for On-Line Authenticated Encryption (OAE) schemes that are fast, secure, flexible, and robust against misuse all at the same time. This work proposes POET (***P**ipelineable **O**n-line **E**ncryption with authentication **T**ag*), a family of OAE schemes which satisfies all the mentioned properties. At its core, POET grounds on the POE family of on-line ciphers (***P**iplineable **O**n-line **E**ncryption*).

POET is fast. Its throughput is comparable to that of reference authenticated ciphers, such as OCB3 or AES-GCM, which lack the robustness provided by POET. When instantiated with the AES and Galois-Field multiplication, POET processes messages at a speed of 3.9 clock cycles per byte per core. Moreover, POET introduces a minimal overhead of only two additional block-cipher calls to generate the authentication tag. For an efficient transmission, POET only transfers the additional tag, avoiding any overhead at the message.

POET is robust. The standard security notions for AE schemes – which POET satisfies up to the birthday bound – assume adversaries to behave "nonce-respectingly", and to ignore decrypted ciphertexts if the authentication fails. Almost all previous AE schemes are insecure whenever these assumptions are violated. This is a highly relevant and greatly underestimated practical issue. POET addresses it by providing security even under *both* "nonce misuse" *and* "decryption misuse".

POET is provably secure. POET bases on well-studied primitives, which simplifies the formal analysis greatly. We provide a security proof, making standard assumptions on the block cipher's security.

POET is flexible. POE and POET are ready-to-use for a variety of applications. We provide a fully generic specification to allow programmers to choose primitives that are tailored to their use case. As a recommendation, we propose the AES as block cipher, and either four-round AES, Galois-Field multiplications, or the full AES for universal hashing. As a desirable side effect of our recommendation, we are convinced that POET can be standardized seamlessly.

POET is efficient on a variety of platforms. POET is well-suited for low-end applications, especially when the AES is used for both encryption and universal hashing, which reduces code size and chip space. Mid-range and high-end devices can run POET efficiently thanks to pipelining. In general, software implementations benefit from the wide availability of AES- and/or Galois-Field native instructions on current platforms. For high-throughput applications, and massively parallelized hardware implementations in particular, we propose a special variant, called POET-m, where m determines the level of parallelism.

Contents

Chapter 1

Introduction

On-Line Authenticated Encryption. (On-Line) Authenticated Encryption (AE) schemes are block cipher modes of operations that protect both privacy and integrity of transmitted messages. Most existing block-cipher based AE schemes, such as EAX [7], GCM [37], or OCB3 [36] are provably secure against *nonce-respecting* adversaries; however, they fail badly when nonces are re-used.

Robustness against Nonce-Misuse. The standard requirement for encryption schemes is to prevent leaking any information about plaintexts except their length. A stateless deterministic encryption scheme enables an adversary to decide if a plaintext has been encrypted multiple times or not. As a countermeasure, a cryptosystem is usually defined as a deterministic algorithm that requires a user-supplied state (i.e., a nonce). Thus, the application programmer is responsible for maintaining the state, which reflects the common practice since the algorithm itself is often implemented by a multi-purpose cryptographic library.

In theory, the concept of nonces is simple. In practice, flawed implementations of nonces are ubiquitous [13, 28, 34, 50]. Apart from implementation failures, there are fundamental reasons why software developers cannot always prevent nonce-reuse. A persistently stored counter, which is increased and written back each time a new nonce is needed, may be reseted by a backup – usually after some previous data loss. Similarly, the internal and persistent state of an application may be duplicated when a virtual machine is cloned.

Most currently used AE schemes neither protect the privacy nor the integrity of messages when nonces repeat [21]. In particular, the security of counter-mode-based schemes falls apart completely.

A modern AE scheme should provide a second line of defense under nonce misuse, i.e., a decent level of security even when nonces repeat.

Robustness against Decryption Misuse. Standard AE security notions assume that an adversary learns nothing about the would-be message whenever a ciphertext fails the authenticity check. This is inherent in the idea of authenticated encryption and part of its strength. Similar to the nonce-misuse setting, this concept is simple in theory, but hard

to ensure in practice.

Beyond limiting the damage in the case of implementation failures, there is also another reason for considering decryption misuse for AE schemes. Sometimes, it is just not possible to store the entire would-be message on a decryption device before its authenticity has been checked. This may be due to plain lack of memory, or due to demanding performance requirements (e.g., high speed, low latency, and long messages). One example for such settings are Optical Transport Networks (OTNs) [29]. In such environment, the links between multiple network channels must be capable of transmitting, multiplexing, and switching between massive data streams in a fast and secure manner. OTNs are characterized by high throughput rates of up to 100 Gbps, low latencies in the order of a few clock cycles, and large message frames of up to 64 kB. At that size, a mode of operation using a 128-bit block cipher would require about 4,096 block cipher invocations to complete a decryption, introducing a latency that exceeds the minimum latency goal of OTNs by far.

In such cases, one can choose between two approaches: One could simply pass the decrypted message before the authenticity was checked – which solves the latency and caching issues. However, under such conditions, most AE schemes can no longer sustain neither the privacy nor the integrity of messages. As an alternative, Fouque et al. [23] proposed to mask the plaintext with an intermediate key before releasing it, and only passing the correct key to the receiver after the message was successfully verified. This practice solves the caching and security issues, but still suffers from high latency.

Thus, there is a practical need for AE schemes that provide a second line of defense under decryption misuse, i.e., a decent level of security, even when decryptions of non-authentic ciphertexts have been compromised.

Intermediate Tags. An AE scheme based on a CCA-secure on-line cipher (OPERM-CCA-secure, hereafter) provides an additional desirable feature: The seamless integration of *intermediate tags* [9]. This can be achieved by adding well-formed redundancy (e.g., fixed constants or non- cryptographic checksums) to the plaintexts. For instance, the headers of IP, TCP, or UDP [42, 43, 41] packets contain a 16-bit checksum each, which is *verified* by the receiver and/or network routers. In OTNs, a single 64-kB message frame usually consists of multiple IP packets. Due to the low-latency constraints, receiving routers are not allowed to buffer incoming messages and must forward the first packets towards their destination. However, they can test the validity of the individual packet's checksum to efficiently detect forgery attempts. OPERM-CCA-security ensures that the first TCP/IP packet with an invalid CRC-16 checksum only passes with a probability of at most 2^{-16}. Even if this packet passes, the next packet would again only pass with the same probability and so on and so forth.

POET in a Nutshell. This work introduces the first non-sequential robust[1] on-line AE scheme, called *Pipelineable On-line Encryption with authentication Tag* (or POET hereafter), which is based on an OPERM-CCA-secure family of on-line ciphers, called *Pipelineable On-line Encryption* (POE). POE and POET consist of an ECB layer that is protected by two wrapping layers of chaining with an ϵ-AXU family of hash functions. The property of pipelineability distinguishes POET from previous CCA-secure on-line ciphers (e.g.,

[1] By robust, we mean resistance against both nonce-misuse and decryption-misuse.

TC3 [48]), which are inherently sequential. Thus, it significantly increases the throughput on multi-core systems with integrated AES-NI, and also allows to utilize single-core processors more efficiently. In addition, we propose a generalized version, called POET-m, where m reflects the level of parallelism. POET-m follows a slightly more complex structure than POET. For $m \geq 2$, it can lead to increased software and hardware performance on high-end systems, but only at the cost of using the inverse of the universal hash function family. Note that our recommended version of POET is identical to POET-1.

We define POE and POET in a generic way, allowing the user to choose well-suitable instances for the cipher and the hash function. For concreteness, we propose three instances which use the AES-128 as block cipher and (1) Galois-Field multiplication, (2) four-round AES-128, or (3) full AES-128 for universal hashing.

Outline

The remainder of this work is structured as follows. In Chapter 2 we give a brief overview over our design goals and the distinguishing features of POET. Next, Chapter 4 recalls the necessary preliminaries about universal hash functions, on-line ciphers, and AE schemes that are used in the subsequent parts of this work. In Chapter 5 we define the relevant security notions used in our work, and in Chapter 6, the specification of POET. Chapters 8 and 8.3 are devoted to the security analysis. Next, Chapter 9 provides details on implementational aspects and performance evaluation of POET. Chapter 10 contains our design rationale. Finally, chapters 11,12, and 13 contain acknowledgements, and the obligational statements regarding intellectual property and consent.

Chapter 2

Features

Length-Preserving Encryption. POET processes messages of arbitrary length in a length-preserving manner, i.e., it encrypts m-bit plaintexts to m-bit ciphertexts, without appending any padding. This functionality is especially useful for (battery-powered) resource-constrained devices, where the transmission of bits is costly.

On-Line. POET provides on-line encryption and decryption, i.e., it can process the i-th input block before the $(i+1)$-th block has been read.

Authentication of Associated Data of Arbitrary Length. POET allows to authenticate associated data (or header, hereafter) of arbitrary length, including the empty string. Since the result of the header-processing step is required as an input parameter for the tag-generation process, POET appends the public message number nonce and pads the given header with a standard 10* padding. Thus, the POET approach renders the entire header into a nonce.

In theory, POET could employ any secure MAC to process the header. Among the variety of existing constructions, we *borrow* the provably secure PMAC design which allows to process the header blocks in arbitrary and parallelizable order to reduce the latency on multi-core CPUs.

Support For Intermediate Tags. POET offers built-in support for intermediate tags when messages already contain some well-formed redundancy, e.g., fixed constants or non-cryptographic checksums. Therefore, POET is well-suited for low-latency environments, such as OTNs, where messages usually consist of multiple TCP/IP packages with integrated (although small) checksums. Note that non-cryptographic intermediate tags lack the level of security of cryptographic authentication tags.

Variable Tag Lengths. While we recommend tags of the block cipher's state size n, POET also provides limited support for truncated tags. Network protocols – such as TLS 1.x [17, 18, 19] or IPSec [1, 33] – usually employ authentication tags of 96 bits. For messages whose length is a multiple of n, POET provides full flexibility to choose tag sizes.

In the other case, tags can still be truncated but only with the requirement that tag and final message block should sum up to at least n bits. Note that this complies with the TLS and IPSec protocol suites.

Incremental Updates For The Header. POET supports incrementality for the header, i.e., the tag can be efficiently recomputed with only $\ell + 1$ block-cipher calls when ℓ header blocks change.

Performance. Our recommended instance of POET is the AES and/or Galois Field multiplication. Therefore, POET can benefit greatly from the available native instruction sets of current processors. PMAC provides POET with a maximum of parallelism when the header is processed. For the message encryption and decryption, POET requires only a single block-cipher and two hash-function calls per message block. The non-sequential design of POET allows to efficiently process subsequent message blocks exploiting the CPU pipeline and multi-threading techniques. An optimized implementation of POET by Bogdanov et al. [12] can already process a single message at a speed of about 3.9, and multiple messages at a speed of about 2.1 clock cycles per byte per core.

Comparison to GCM. Like GCM, POET is an efficient OAE scheme. However, in contrast to GCM, POET supports resistance against nonce and decryption misuse wheraes the security of GCM totally falls apart when a nonce is used at least twice. Finally, POET is a CCA-secure on-line cipher wheras GCM is only CPA-secure.

Chapter 3

Security Goals

The security claims for POET are given in Table 3.1.

	Security in Bits			Maximal # of Blocks		
Goals	**GF**	**AES-4**	**Full AES**	**GF**	**AES-4**	**Full AES**
Plaintext confidentiality	128	128	128	$\ll 2^{64}$	$\ll 2^{56}$	$\ll 2^{64}$
Plaintext integrity	128	128	128	$\ll 2^{64}$	$\ll 2^{56}$	$\ll 2^{64}$
Associated data integrity	128	128	128	$\ll 2^{64}$	$\ll 2^{56}$	$\ll 2^{64}$
Nonce integrity	128	128	128	$\ll 2^{64}$	$\ll 2^{56}$	$\ll 2^{64}$

Table 3.1.: Security claims (left) and maximum number of blocks under a single key (right) for our recommended versions of POET. All versions use the AES as block cipher. The ϵ-AXU family of hash functions is given by a Galois-Field multiplication in $GF(2^{128})$ for "GF", by four-round AES for "AES-4", and the full-round AES for "Full AES".

POET does not intent to support secret message numbers, i.e., the length of secret message numbers is 0 bits. Our three recommended instantiations of POET all use the AES-128 as block cipher. Hence, the block size for message blocks, header blocks, and nonce is 128 bits. The recommended tag size for all recommended instantiations is 128 bits.

POET is designed to provide robustness against nonce misuse, i.e., POET maintains full integrity and confidentiality, except for leaking collisions of the longest common prefix of messages. Furthermore, it provides robustness against decryption misuse, where it maintains on-line confidentiality.

For all instantiations of POET, we assume that the legitimate key holder does not approach about $2^{-\epsilon/2}$ blocks encrypted under a single key.

Chapter 4

Preliminaries

This section introduces the general notions that are used throughout this work. Table 4.1 summarizes the most frequently used identifiers.

Identifier	Description
C	Ciphertext
E/E^{-1}	Cipher (encryption function)/Inverse cipher
F	Function, mostly universal hash function
H	Header (= associated data)
K	Cipher key
L	Key for header processing
L_M	Key for processing the last message block
L_T	Key for tag generation
L_F^{top},L_F^{bot}	Keys for the ϵ-AXU family of hash functions F
M	Plaintext message
N	Public message number (= initial value/nonce)
SK	User-given secret key
T	Authentication tag
τ	Header tag
n	Block length in bits
k	Key length in bits
$\|X\|$	Length of X in bits
X_i	i-th block of a value X
X^i	The i-th element of a set
$X \mid\mid Y$	Concatenation of two values X and Y
$X \mid\mid 10^*$	Value X with a single '1'-bit appended, and then padded with zeros until its length is a multiple of n
$\mathcal{X}$	Set or family $\mathcal{X}$
$X \xleftarrow{\$} \mathcal{X}$	X is a uniformly at random chosen sample from $\mathcal{X}$.

Table 4.1.: Notions used throughout this paper.

In general, we write uppercase letters to denote functions, parameters, or values (e.g., M, C, E), lowercase letters to denote lengths (e.g., n, k), and calligraphic uppercase letters to represent sets or families of functions (e.g., $\mathcal{F}$).

4.1. Universal Hash Functions

For POET we make extensive use of the well-studied properties of universal hash-function families. In this section, we start with recalling the relevant standard definitions from the literature by Carter and Wegman [15, 55]; thereupon, we restate the theorem related to these functions by Boesgaard et al. [11], and their composition by Stinson [52, 53].

Definition 4.1 (ϵ-Almost-(XOR-)Universal Hash Functions). *Suppose, $\mathcal{F} = \{F : \{0,1\}^m \to \{0,1\}^n\}$ is a family of hash functions. $\mathcal{F}$ is called* ϵ-almost- universal *(ϵ-AU) iff for all $X, X' \in \{0,1\}^m$, $X \neq X'$:*

$$\Pr[F \xleftarrow{\$} \mathcal{F} : \ F(X) = F(X')] \leq \epsilon.$$

$\mathcal{F}$ is called ϵ-almost-XOR-universal *(ϵ-AXU) iff for all $X, X' \in \{0,1\}^m$, $Y \in \{0,1\}^n$, $X \neq X'$:*

$$\Pr[F \xleftarrow{\$} \mathcal{F} : \ F(X) \oplus F(X') = Y] \leq \epsilon.$$

The definition of strong universal hash functions is similar.

Definition 4.2 (Strong Universal Hash Functions). *Suppose $\mathcal{F} = \{F : \{0,1\}^m \to \{0,1\}^n\}$ is a family of hash functions. $\mathcal{F}$ is called* strong- universal *(SU) iff for all $X \in \{0,1\}^m, Y \in \{0,1\}^n$:*

$$\Pr[F \xleftarrow{\$} \mathcal{F} : \ F(X) = Y] \leq 1/2^n,$$

and for all $X, X' \in \{0,1\}^m$ with $X \neq X'$ and $Y, Y' \in \{0,1\}^n$:

$$\Pr[F \xleftarrow{\$} \mathcal{F} : \ F(X) = Y, F(X') = Y'] \leq 1/2^{2n}.$$

Boesgaard et al. showed in [11] that an ϵ-AXU family of hash functions can be reduced to a family of ϵ-AU hash functions by XORing an arbitrary value to the output:

Theorem 4.3 (Theorem 3 from [11]). *Let $\mathcal{F} = \{F : \{0,1\}^m \to \{0,1\}^n\}$ be a family of ϵ-AXU hash functions. Then, the family $\mathcal{F}' = \{F' : \{0,1\}^m \times \{0,1\}^n \to \{0,1\}^n\}$ with $F'(X,Y) = F(X) \oplus Y$ is ϵ-AU.*

The effects of composing two universal hash function instances were studied by Stinson in [52, 53].

Theorem 4.4 (Theorem 5.4 from [53]). *Let $\mathcal{F} = \{F : \{0,1\}^m \to \{0,1\}^n\}$ be an ϵ_1-AU family of hash functions and $\mathcal{G} = \{G : \{0,1\}^n \to \{0,1\}^\ell\}$ an ϵ_2-AU family hash functions. Then, there exists an ϵ-AU family of hash functions $\mathcal{H}$ with $\epsilon \leq \epsilon_1 + \epsilon_2$ and $|H| = |\mathcal{F}| \times |\mathcal{G}|$.*

4.2. Block Ciphers

A block cipher is a keyed family of n-bit permutations $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$, which takes a k-bit key K and an n-bit message M, and outputs an n-bit ciphertext C. We denote $\texttt{Block}(k,n)$ as the set of all (k,n)-bit block ciphers for $n > 0$. For any $E \in \texttt{Block}(k,n)$ and a fixed key $K \in \{0,1\}^k$, the encryption of a message M is given by $E_K(M)$, and the decryption is defined as the inverse function, i.e., $E_K^{-1}(M)$. For any key $K \in \{0,1\}^k$, it applies that $E_K^{-1}(E_K(M)) = M$.

We define the IND-SPRP-security of a block cipher E by the success probability of an adversary trying to differentiate between the block cipher and an n-bit random permutation $\pi(\cdot)$.

Definition 4.5 (IND-SPRP-Security). *Let $E \in \texttt{Block}(k,n)$ denote a block cipher and E^{-1} its inverse. Let Perm_n be the set of all n-bit permutations. The IND-SPRP advantage of $\mathcal{A}$ against E is then defined by*

$$\mathbf{Adv}^{\text{IND-SPRP}}_{E,E^{-1}}(\mathcal{A}) \leq \left|\Pr\left[\mathcal{A}^{E(\cdot),E^{-1}(\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\pi(\cdot),\pi^{-1}(\cdot)} \Rightarrow 1\right]\right|,$$

where the probabilities are taken over $K \xleftarrow{\$} \{0,1\}^k$ and $\pi \xleftarrow{\$} \mathsf{Perm}_n$. We define $\mathbf{Adv}^{\text{IND-SPRP}}_{E,E^{-1}}(q,t)$ as the maximum advantage over all IND-SPRP-adversaries $\mathcal{A}$ on E that run in time at most t and make at most q queries to the available oracles.

4.3. On-Line Ciphers

Definition 4.6 (On-Line Cipher). *Let $\Gamma : \{0,1\}^k \times (\{0,1\}^n)^* \to (\{0,1\}^n)^*$ denote a keyed family of n-bit permutations, which takes a k-bit key K and a message M of an arbitrary number of n-bit blocks, and outputs a ciphertext C consisting of the same number of n-bit blocks as M. We call Γ an* on-line cipher *iff the encryption of message block M_i, for all $i \in [1, |M|/n]$, depends only on the blocks $M_1, \ldots, M_i$.*

Usually, a secure cipher that processes messages of arbitrary length should behave like a random permutation. It is easy to see that on-line ciphers are in conflict with this security property since the encryption of message block M_i does not depend on M_{i+1}. The on-line behavior implies that two messages M and M', which share an m-block common prefix, will always be encrypted to two ciphertexts C and C', which also share an m-block common prefix. Hence, we define an on-line cipher Γ to be secure if and only if no ciphertext reveals any further information about a plaintext than its length and the *longest common prefix* with previous messages. We recall the formal definition of the length of the longest common prefix of a message from [21].

Definition 4.7 (Length of Longest Common Prefix). *For integers $n, \ell, d \geq 1$, let $\mathcal{D}_n^d = (\{0,1\}^n)^d$ denote the set of all strings that consist of exactly d blocks of n bits each. Further, let $\mathcal{D}_n^* = \bigcup_{d \geq 0} \mathcal{D}_n^d$ denote the set which consists of all possible n-bit strings and $\mathcal{D}_{\ell,n} = \bigcup_{0 \leq d \leq \ell} \mathcal{D}_n^d$ the set of all possible strings which consist of 0 to ℓ n-bit blocks. For arbitrary $P \in \mathcal{D}_n^d$, let P_i denote the i-th block for all $i \in 1, \ldots, d$. For $P, R \in \mathcal{D}_n^*$, we define the* length of the longest common prefix of n-bit blocks *of P and R by*

$$\mathsf{LLCP}_n(P, R) = \max_i \{\forall j \in 1, \ldots, i : \ P_j = R_j\}.$$

For a non-empty set $\mathcal{Q}$ of strings in D_n^*, we define $\mathsf{LLCP}_n(\mathcal{Q}, P)$ by

$$\max_{q \in \mathcal{Q}} \{\mathsf{LLCP}_n(q, P)\}.$$

For any two ℓ-block inputs M and M' with $M \neq M'$, that share an exactly m-block common prefix $M_1 \mid\mid \ldots \mid\mid M_m$, the corresponding outputs $C = P(M)$ and $C' = P(M')$ satisfy $C_i = C'_i$ for all $i \in [1, m]$ and $m \leq \ell$, where P denotes an on-line permutation. However, it applies that $C_{m+1} \neq C'_{m+1}$ and all further blocks C_i and C'_i, with $i \in [m+2, \ell]$, to be *independent*. This behavior is defined by on-line permutations. We recall their definition in the following.

Definition 4.8 (On-Line Permutation). *Let $F_i : (\{0,1\}^n)^i \to \{0,1\}^n$ be a family of indexed n-bit permutations, i.e., for a fixed index $j \in (\{0,1\}^n)^{i-1}$ it applies that $F_i(j, \cdot)$ is a permutation. We define an n-bit on-line permutation $P : (\{0,1\}^n)^\ell \to (\{0,1\}^n)^\ell$ as a composition of ℓ permutations $F_1 \cup F_2 \cup \ldots \cup F_\ell$. An ℓ-block message $M = (M_1, \ldots, M_\ell)$ is mapped to an ℓ-block output $C = (C_1, \ldots, C_\ell)$ by*

$$C_i = F_i(M_1 \mid\mid \ldots \mid\mid M_{i-1}, M_i), \quad \forall i \in [1, \ell].$$

We denote by OPerm_n the set of all n-bit on-line permutations. Note, that one can efficiently implement a random on-line permutation by lazy sampling.

4.4. (On-Line) Authenticated Encryption Schemes

Definition 4.9 (Authenticated Encryption Scheme With Associated Data). *An authenticated encryption scheme is a triple $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, with a key-generation procedure $\mathcal{K}$ that generates a key K, an encryption algorithm $\mathcal{E}_K(H, M)$, and a decryption algorithm $\mathcal{D}_K(H, C, T)$. H denotes the associated data (or header), M the message, T the authentication tag, and C the ciphertext, with $H, M, C \in \{0,1\}^*$ and $T \in \{0,1\}^t$,*

where $t > 1$ is an integer. We write

$$(C, T) \leftarrow \mathcal{E}_K(H, M) \quad \textit{and}$$
$$M \,|\, \perp \leftarrow \mathcal{D}_K(H, C, T)$$

to state that $\mathcal{E}$ always outputs a ciphertext C and the authentication tag T for the tuple (H, M), and $\mathcal{D}$ outputs the decryption of (H, C) iff the given tag is valid or $\perp$ otherwise. The correctness condition applies that $\mathcal{D}_K(\mathcal{E}_K(H, M)) = M$ to hold for each triple (K, H, M).

Note that this definition implies that the nonce N is part of the header.

We call an authenticated encryption scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ an *on-line authenticated encryption scheme* if $\mathcal{E}$ encrypts plaintexts in an on-line manner.

Chapter 5

Security Notions

This section recalls the common security notions for AE schemes and on-line AE schemes. Authenticated encryption aims at ensuring privacy and authenticity of encrypted messages both at the same time. Therefore, this section recalls the security notions which consider either or both aspects. The literature provides various notions and relations for deterministic [47] and nonce-based AE schemes [5, 6, 32, 44, 46]. We consider the common CCA3 notion by Rogaway and Shrimpton [47] and recall the related IND-CPA and INT-CTXT notions for privacy and integrity which are covered by CCA3. Thereupon, we point out the differences between on-line and off-line encryption by recalling the OCCA3 notion for on-line AE schemes, and the OPERM-CCA notion for privacy of on-line ciphers. Therefore, we follow the approach from [21] and provide a game for each notion that illustrates the interaction of the respective adversaries with their oracles.

Remark. Note that we always consider nonce-ignoring adversaries which are allowed to use a nonce multiple times similar to the security notions of integrity for authenticated encryption schemes in [21].

5.1. General Security Notions for AE Schemes

Definition 5.1 (CCA3-Security). *Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an authenticated encryption scheme as defined in Definition 4.9. Then, the CCA3-advantage of a computationally bounded adversary $\mathcal{A}$ is defined as*

$$\mathbf{Adv}_{\Pi}^{CCA3}(\mathcal{A}) = \left| \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(\cdot,\cdot),\mathcal{D}_K(\cdot,\cdot,\cdot)} \Rightarrow 1 \right] - \Pr\left[\mathcal{A}^{\$(\cdot,\cdot),\perp(\cdot,\cdot,\cdot)} \Rightarrow 1 \right] \right|.$$

The CCA3 notion states that $\mathcal{A}$ has access to an oracle $\mathcal{O}$, which provides $\mathcal{A}$ with an encryption and a decryption functions. At the beginning, $\mathcal{O}$ tosses a fair coin; depending on the result of the coin toss, $\mathcal{O}$ uses the *real* encryption $\mathcal{E}_K(\cdot,\cdot)$ and decryption $\mathcal{D}_K(\cdot,\cdot,\cdot)$ functions, or a *random* function $\$(\cdot,\cdot)$ for the encryption and a $\perp$ function for $\perp(\cdot,\cdot,\cdot)$,

which returns $\perp$ on every input, for the decryption queries of $\mathcal{A}$. Wlog., we assume that $\mathcal{A}$ never asks a query to which it already knows the answer. The goal of $\mathcal{A}$ in this scenario is to determine the result of the coin toss, i.e., to distinguish between the real encryptions with Π and random one.

Definition 5.2 (IND-CPA-Security). *Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an authenticated encryption scheme as defined in Definition 4.9. Then, the IND-CPA-advantage of a computationally bounded adversary $\mathcal{A}$ for Π is defined as*

$$\mathbf{Adv}_{\Pi}^{\mathsf{IND\text{-}CPA}}(\mathcal{A}) \leq \left| \Pr\left[K \xleftarrow{\$} \mathcal{K} : \ \mathcal{A}^{\mathcal{E}(\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(\cdot,\cdot)} \Rightarrow 1\right] \right|.$$

We define $\mathbf{Adv}_{\Pi}^{\mathsf{IND\text{-}CPA}}(q, \ell, t)$ as the maximum advantage over all IND-CPA-adversaries $\mathcal{A}$ on Π that run in time at most t, and make at most q queries of total length ℓ to the available oracles.

Privacy and Integrity Notions. Let $\mathcal{A}$ be a computationally bounded IND-CPA-adversary with access to an oracle $\mathcal{O}$, which responds either with real encryptions using POET_{E_K} or a random permutation π, as given in Definition 5.2. In the beginning, the oracle tosses a fair coin to obtain a bit b. Thereupon, $\mathcal{A}$ can query messages to $\mathcal{O}$. Depending on b, $\mathcal{A}$ obtains either "real" encryptions for the messages it sends, or just the "random" outputs. Hence, the challenge for $\mathcal{A}$ is to guess b.

Definition 5.3 (INT-CTXT-Security). *Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an authenticated encryption scheme as defined in Definition 4.9. Then, the INT-CTXT-advantage of a computationally bounded adversary $\mathcal{A}$ for Π is given by the success probability of winning the game $G_{\mathsf{INT\text{-}CTXT}}$ that is defined in Figure 5.1. Thus, we obtain*

$$\mathbf{Adv}_{\Pi}^{\mathsf{INT\text{-}CTXT}}(\mathcal{A}) \leq \Pr\left[\mathcal{A}^{G_{\mathsf{INT\text{-}CTXT}}} \Rightarrow 1\right].$$

We define $\mathbf{Adv}_{\Pi}^{\mathsf{INT\text{-}CTXT}}(q, \ell, t)$ as the maximum advantage over all INT-CTXT-adversaries $\mathcal{A}$ on Π that run in time at most t, and make at most q queries of total length ℓ to the available oracles.

```
1  Initialize()
2  K ←$ K; Q ← ∅;

3  Finalize()
4  return win;
```

```
10 Encrypt(H, M)
11 (C, T) ← E_K(H, M);
12 Q ← Q ∪ {(H, C, T)};
13 return (C, T);
```

```
20 Verify(H, C, T)
21 M ← D_K(H, C);
22 if ((H, C, T) ∉ Q and M ≠ ⊥) then
23    win ← true;
24 return (M ≠ ⊥);
```

Figure 5.1.: The INT-CTXT game $G_{\mathsf{INT\text{-}CTXT}}$ for an authenticated encryption scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. $\mathcal{Q}$ denotes the query history of $\mathcal{A}$.

Relation to Privacy and Integrity Notions. Bellare and Namprempre showed in [5] that the CCA3 advantage of an adversary on an AE scheme Π can be upper bounded by the sum of the maximal advantage of an adversary on the integrity of Π, and the maximal advantage of a chosen-plaintext adversary on the privacy of Π. Fleischmann et al. generalized this relation in [21] (cf. Theorem 5.4). We illustrate this notion simply by rewriting Definition 5.1 as

$$\left|\Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(\cdot,\cdot),\mathcal{D}_K(\cdot,\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(\cdot,\cdot),\perp(\cdot,\cdot,\cdot)} \Rightarrow 1\right]\right. \tag{5.1}$$

$$\left. + \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(\cdot,\cdot),\perp(\cdot,\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(\cdot,\cdot),\perp(\cdot,\cdot,\cdot)} \Rightarrow 1\right]\right|. \tag{5.2}$$

Equation (5.1) refers to the INT-CTXT advantage and Equation 5.2 to the IND-CPA-advantage of $\mathcal{A}$ on Π.

Theorem 5.4 (Theorem 1 in [21]). *Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an authenticated encryption scheme, with a header space $\mathcal{H}$, a message space $\mathcal{M}$, and a tag space $\mathcal{T}$. Then, the CCA3-advantage over all adversaries $\mathcal{A}$ that run in time at most t, ask at most q queries of a total length at most ℓ to the available oracles, can be upper bounded by*

$$\mathbf{Adv}_{\Pi}^{CCA3}(q,t,\ell) \leq \mathbf{Adv}_{\Pi}^{IND\text{-}CPA}(q,t,\ell) + \mathbf{Adv}_{\Pi}^{INT\text{-}CTXT}(q,t,\ell).$$

5.2. Security Notions for On-Line AE Schemes

Definition 5.5 (OCCA3-Security). *Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an on-line authenticated encryption scheme as defined in Definition 4.9. Then, the OCCA3-advantage of an adversary $\mathcal{A}$ is upper bounded by*

$$\mathbf{Adv}_{\Pi}^{OCCA3}(\mathcal{A}) \leq \mathbf{Adv}_{\Pi}^{OPERM\text{-}CPA}(q,\ell,t) + \mathbf{Adv}_{\Pi}^{INT\text{-}CTXT}(q,\ell,t).$$

The OCCA3-advantage of Π, $\mathbf{Adv}_{\Pi}^{OCCA3}(q,\ell,t)$, is then defined by the maximum advantage of all OCCA3-adversaries $\mathcal{A}$ that run in time at most t, and make at most q queries of total length ℓ to the available oracles.

Based on the definition above, an on-line authenticated encryption scheme Π is OCCA3-secure if it provides both OPERM-CPA-security *and* INT-CTXT-security. We borrow the formal OPERM-CCA-notion from Bellare et al. [3, 4], which specifies the maximal advantage of an adversary $\mathcal{A}$ with access to both encryption and decryption oracles to distinguish between the output of a on-line cipher Γ under a randomly chosen key K and that of a random permutation.

Definition 5.6 (OPERM-CCA-Security). *Let K be a k-bit key, P a random on-line permutation, and $\Gamma : \{0,1\}^k \times (\{0,1\}^n)^* \to (\{0,1\}^n)^*$ an on-line cipher. Then, we define the OPERM-CCA-advantage of an adversary $\mathcal{A}$ by*

$$\mathbf{Adv}_{\Gamma}^{\mathsf{OPERM\text{-}CCA}}(\mathcal{A}) = \left|\Pr\left[\mathcal{A}^{\Gamma_K(\cdot),\Gamma_K^{-1}(\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{P(\cdot),P^{-1}(\cdot)} \Rightarrow 1\right]\right|,$$

where the probabilities are taken over $K \xleftarrow{\$} \mathcal{K}$ and $P \xleftarrow{\$} \mathsf{OPerm}_n$. Further, we define $\mathbf{Adv}_{\Gamma}^{\mathsf{OPERM\text{-}CCA}}(q,\ell,t)$ as the maximum advantage over all OPERM-CCA-adversaries $\mathcal{A}$ that run in time at most t, and make at most q queries of total length ℓ to the available oracles.

In [5] Bellare and Namprempre also showed that IND-CCA-security implies *non-malleable chosen-ciphertext*-security (NM-CCA). Hence, it is easy to derive that OPERM-CCA implies "weak" non- malleability, i.e., an adversary that manipulates the i-th ciphertext block cannot distinguish between the $(i+1)$, $(i+2)$, ... ciphertext blocks of Γ and random. Thus, it suffices to show the OPERM-CCA-security of POET to achieve non-malleability.

Note that an OPERM-CPA-adversary $\mathcal{A}$ on an on-line cipher Γ can always be used by an OPERM-CCA-adversary $\mathcal{A}$' on Γ which inherits the advantage of $\mathcal{A}$. Hence, an upper bound for the OPERM-CCA-advantage of Γ is always same as an upper bound for the OPERM-CPA-advantage of Γ.

Chapter 6

Specification

This chapter defines the POET family of on-line AE schemes. From a top-level point of view, POET consists of three layers:

1. The top-row layer applies an ϵ-AXU function F_t to the previous chaining values and computes the XOR of the output and the message block M_i.
2. The middle layer performs an ECB encryption.
3. The bottom-row layer applies another ϵ-AXU function F_b to the previous chaining value of bottom-row and computes the XOR between the output and the encrypted blocks. The result denotes the ciphertext block C_i.

Thus, the two chaining values X and Y are updated by applying the ϵ-AXU functions F_t at the top and F_b at the bottom and an XOR before and after the encryption layer, respectively. The chaining in POET ensures that each message block depends on all previous blocks, which provides our desired weak non-malleability property (cf. Section 5.2). A schematic illustration of the encryption process of POET is given in Figure 6.1.

POET takes advantage of several well-suited practices from previous modern AE schemes: the masking process and the middle ECB layer follow the secure XEX approach [45], which provides security against chosen-plaintext and chosen-ciphertext adversaries. The processing of the header is based on the PMAC design [10] (see Figure 6.2), which is both fast and provably secure. Finally, the processing of the final message block and the tag-generation of POET follows the idea of McOE [22], which provides length-preserving encryption/decryption.

In the remainder of this chapter, we first provide a formal definition of POET. Next, we describe the individual steps of the **key generation**, **header** and **message processing**, **tag generation** and **verification**. Prior, we define three auxiliary functions, that are used for length-preserving encryption:

- **GetMSB**(X, b) returns the b most significant bits of X.
- **GetLSB**(X, b) returns the b least significant bits of X.
- **Split**(X, b) returns a tuple (X^α, X^β) where X^α contains the b most significant bits of X and X^β the $|X| - b$ least significant bits of X. Hence, $X^\alpha \mid\mid X^\beta = X$.

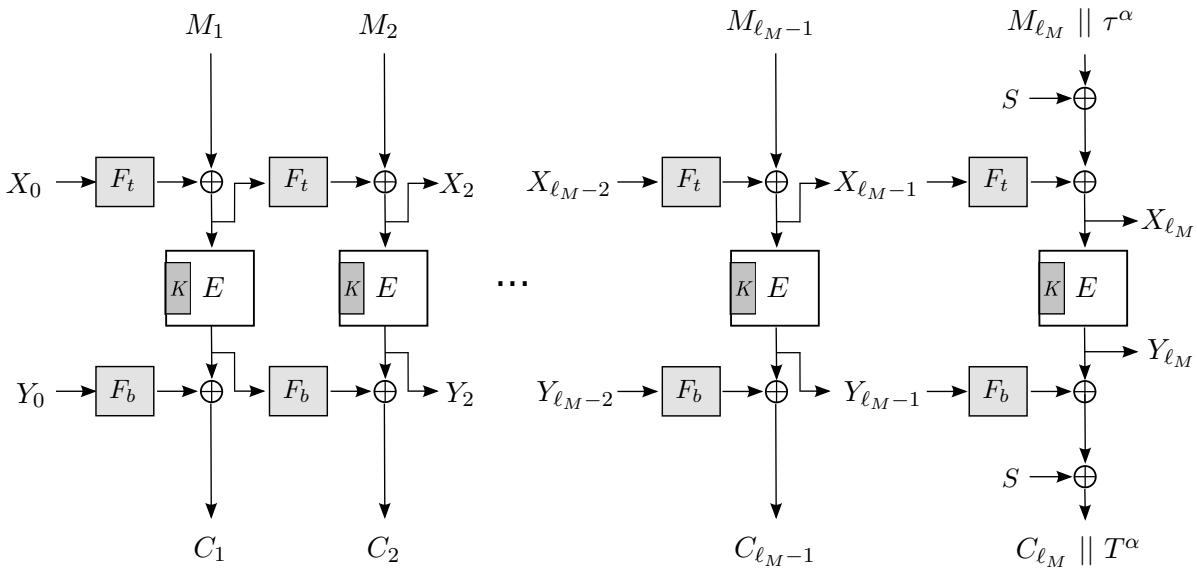

Figure 6.1.: Schematic illustration of the encryption process with POET for an (ℓ_M)-block message $M = M_1, \ldots, M_{\ell_M}$, where S denotes the encrypted message length, i.e., $S = E_K(|M|)$, F is an ϵ-AXU family of hash functions, and τ^α is taken from the most significant bits of the header processing to pad the final message block. Note that the functions F_t and F_b use the keys L_F^{top} and L_F^{bot}, respectively.

6.1. Definition of POET

Definition 6.1 (POET). *Let $m, n, k \geq 1$ be three integers. Let $\mathsf{POET} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an AE scheme as defined in Definition 4.9, $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ a block cipher and $F : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a family of keyed ϵ-AXU hash functions. Furthermore, let H be the header (including the public message number N appended to its end), M the message, T the authentication tag, and C the ciphertext, with $H, M, C \in \{0,1\}^*$ and $T \in \{0,1\}^n$. Then, $\mathcal{E}$ is given by procedure **EncryptAndAuthenticate**, $\mathcal{D}$ by procedure **DecryptAndVerify**, and $\mathcal{K}$ by procedure **GenerateKeys**, as shown in Algorithms 6.1 and 6.2, respectively.*

Algorithm 6.1 **EncryptAndAuthenticate** and **DecryptAndVerify**.

EncryptAndAuthenticate(H, M)	**DecryptAndVerify**(H, C, T)
101: $\ell_M \leftarrow \lceil |M|/n \rceil$	201: $\ell_C \leftarrow \lceil |C|/n \rceil$
102: $\tau \leftarrow$ **ProcessHeader**(H)	202: $\tau \leftarrow$ **ProcessHeader**(H)
103: $(C, X_{\ell_M}, Y_{\ell_M}) \leftarrow$ **Encrypt**(M, τ)	203: $(M, X_{\ell_C}, Y_{\ell_C}) \leftarrow$ **Decrypt**(C, T, τ)
104: $(C_{\ell_M}, T^\alpha) \leftarrow$ **Split**$(C_{\ell_M}, |M_{\ell_M}|)$	204: $(M_{\ell_C}, \tau') \leftarrow$ **Split**$(M_{\ell_C}, |C_{\ell_C}|)$
105: $T^\beta \leftarrow$ **GenerateTag**$(\tau, X_{\ell_M}, Y_{\ell_M})$	205: **if VerifyTag**$(T, X_{\ell_C}, Y_{\ell_C}, \tau, \tau')$ **then**
106: $T \leftarrow T^\alpha \,||\, T^\beta$	206: **return** M
107: **return** $(C_1 \,||\, \ldots \,||\, C_{\ell_M}, T)$	207: **end if**
	208: **return** $\perp$

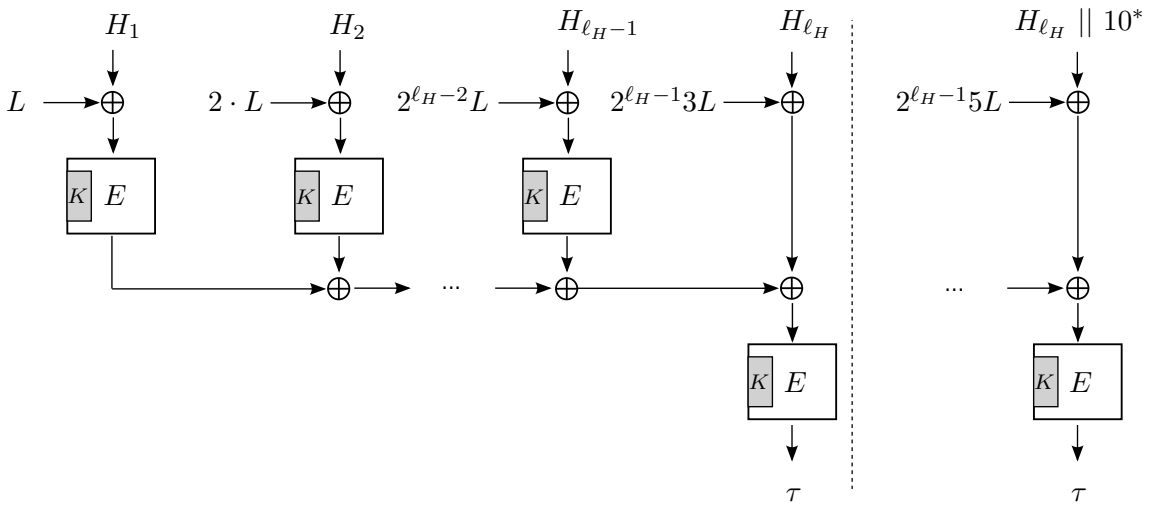

Figure 6.2.: The **ProcessHeader** procedure of POET. The right block depicts the processing of the final block in the case when the header length is not a multiple of the block size.

Key Generation. POET requires in total five pairwise independent k-bit keys used as follows:

- One key K for the block cipher.
- One masking key L for processing the header.
- Two keys L_F^{top} and L_F^{bot} for the keyed family of hash functions F.
- One masking key L_T to generate the authentication tag.

The key generation follows the idea from [30]. Thus, the user supplies a k-bit secret key SK. The further keys are then generated by encrypting distinct constants $const_0$, $const_1$, etc. For simplicity, we recommend $const_i = i$. Therefore, under the assumption that E is a secure Pseudorandom Permutation (PRP), we can ensure to obtain pairwise independent keys for the block cipher invocation and the masking.

Algorithm 6.2 The procedures **GenerateKeys** and **ProcessHeader**.

GenerateKeys(SK)
301: $K \leftarrow E_{SK}(const_0)$
302: $L \leftarrow E_{SK}(const_1)$
303: $L_F^{top} \leftarrow E_{SK}(const_2)$
304: $L_F^{bot} \leftarrow E_{SK}(const_3)$
305: $L_T \leftarrow E_{SK}(const_4)$
306: **return** $(K, L, L_F^{top}, L_F^{bot}, L_T)$

ProcessHeader(H)
401: $\ell_H \leftarrow \lceil |H|/n \rceil$, $\Sigma \leftarrow 0^n$
402: **for** $i \leftarrow 1, \ldots, \ell_H - 1$ **do**
403: $\quad \Sigma \leftarrow \Sigma \oplus E_K(H_i \oplus 2^{i-1}L)$
404: **end for**
405: **if** $|H_{\ell_H}| = n$ **then**
406: $\quad \tau \leftarrow E_K(H_{\ell_H} \oplus 2^{\ell_H-1}3L \oplus \Sigma)$
407: **else**
408: $\quad H_{\ell_H} \leftarrow H_{\ell_H} \,||\, 10^*$
409: $\quad \tau \leftarrow E_K(H_{\ell_H} \oplus 2^{\ell_H-1}5L \oplus \Sigma)$
410: **end if**
411: **return** τ

Header Processing. The header H denotes the associated data of a message, and an n-bit nonce N appended to its end. Hence, one can also interpret the entire header as a nonce.

In cases when the header length $|H|$ is smaller than the block size n or not a multiple of n, we apply the common 10*-padding, i.e., we append a single '1'-bit to the header followed by as many '0'-bits as necessary s.t. the length of the padded header becomes a multiple of n, and consists of at least one block. This is a mandatory requirement for POET since it generates an intermediate tag τ which is later used to generate the authentication tag. Thus, if the size of the user-given header is zero, POET masks one block of the form $1 \,||\, 0^{n-1}$ and then encrypts the result to generate the value τ.

POET processes the header in a PMAC-like fashion [10]. First, each block is masked by XORing a distinct multiple of L. Note that all multiplications are Galois-Field Multiplications in $GF(2^{128})$. Each masked header block, except for the last one, is then encrypted by E, and all outputs are XORed together, including the last masked block. The XOR sum is then encrypted again by E to generate an intermediate tag τ. The procedure **ProcessHeader** is shown in Algorithm 6.2. Note that the procedure can be fully parallelized.

Algorithm 6.3 The procedures **Encrypt** and **Decrypt**.

Encrypt(M, τ)	**Decrypt**(C, T, τ)
501: $\ell_M \leftarrow \lceil \lvert M\rvert /n \rceil$, $X_0 \leftarrow Y_0 \leftarrow \tau$	601: $\ell_C \leftarrow \lceil \lvert C\rvert /n \rceil$, $X_0 \leftarrow Y_0 \leftarrow \tau$
502: **for** $i \leftarrow 1, \ldots, \ell_M - 1$ **do**	602: **for** $i \leftarrow 1, \ldots, \ell_C - 1$ **do**
503: $\quad X_i \leftarrow F_t(X_{i-1}) \oplus M_i$	603: $\quad Y_i \leftarrow F_b(Y_{i-1}) \oplus C_i$
504: $\quad Y_i \leftarrow E_K(X_i)$	604: $\quad X_i \leftarrow E_K^{-1}(Y_i)$
505: $\quad C_i \leftarrow F_b(Y_{i-1}) \oplus Y_i$	605: $\quad M_i \leftarrow F_t(X_{i-1}) \oplus X_i$
506: **end for**	606: **end for**
507: $S \leftarrow E_K(\lvert M\rvert)$	607: $S \leftarrow E_K(\lvert C\rvert)$
508: $\tau^\alpha \leftarrow$ **GetMSB**$(\tau, n - \lvert M_{\ell_M}\rvert)$	608: $T^\alpha \leftarrow$ **GetMSB**$(T, n - \lvert C_{\ell_C}\rvert)$
509: $M^*_{\ell_M} \leftarrow (M_{\ell_M} \,||\, \tau^\alpha)$	609: $C^*_{\ell_C} \leftarrow (C_{\ell_C} \,||\, T^\alpha)$
510: $X_{\ell_M} \leftarrow F_t(X_{\ell_M - 1}) \oplus M^*_{\ell_M} \oplus S$	610: $Y_{\ell_C} \leftarrow F_b(Y_{\ell_C - 1}) \oplus C^*_{\ell_C} \oplus S$
511: $Y_{\ell_M} \leftarrow E_K(X_{\ell_M})$	611: $X_{\ell_C} \leftarrow E_K^{-1}(Y_{\ell_C})$
512: $C_{\ell_M} \leftarrow F_b(Y_{\ell_M - 1}) \oplus Y_{\ell_M} \oplus S$	612: $M_{\ell_C} \leftarrow F_t(X_{\ell_C - 1}) \oplus X_{\ell_C} \oplus S$
513: $C \leftarrow (C_1 \,||\, \ldots \,||\, C_{\ell_M})$	613: $M \leftarrow (M_1 \,||\, \ldots \,||\, M_{\ell_C})$
514: **return** $(C, X_{\ell_M}, Y_{\ell_M})$	614: **return** $(M, X_{\ell_C}, Y_{\ell_C})$

Encryption/Decryption. The workflow of the message-processing is shown in Algorithm 6.3 and Figure 6.1. For each message block M_i for $1 \leq i \leq \ell_M - 1$, the following process is applied: update the top- and bottom-row chaining values X_{i-1} and Y_{i-1} by applying the ϵ-AXU functions $F(L_F^{top}, X_{i-1})$ and $F(L_F^{bot}, Y_{i-1})$, respectively. Then, XOR the output of $F(L_F^{top}, X_{i-1})$ with the current message block M_i to derive X_i. The value X_i is then used twice; (1) as an input to the block cipher E and (2) as the new chaining value in the top row. The output $Y_i = E_K(X_i)$ is also used twice; (1) as the new chaining value for the bottom-row and (2) is XORed with the updated chaining value $F(L_F^{bot}, Y_{i-1})$ to generate the current ciphertext block C_i. The decryption process is defined similarly in procedure **Decrypt**. For simplicity, we write $F_t(\cdot)$ and $F_b(\cdot)$ instead of $F(L_F^{top}, \cdot)$ and $F(L_F^{bot}, \cdot)$ hereafter.

To process the final message block M_{ℓ_M}, we first separate header and message processing, then we encrypt the length of the message and XOR the result S to the final message

block M_{ℓ_M}. The result of $M_{\ell_M} \oplus S$ is then XORed with $F_t(X_{\ell_M-1})$ to produce X_{ℓ_M}. The value X_{ℓ_M} is again used twice; (1) as input to the block cipher call and (2) as chaining input to the tag generation step. The output $Y_{\ell_M} = E_K(X_{\ell_M})$ is also used twice; (1) as the new bottom-row chaining value for the tag generation and (2) is XORed with the updated chaining value $F_b(Y_{\ell_M-1})$. Thereupon, to produce the final ciphertext block, the result of the XOR operation is XORed again with the encrypted message length S.

For messages whose length is not a multiple of the block size, we employ a slightly more complicated procedure for the final block. Though, to avoid an overhead when transmitting the message, POET borrows the provably secure *tag-splitting* technique from McOE [21]. This means that messages are never padded; instead, the final message block M_{ℓ_M} is filled up with the most significant bits of the intermediate tag τ:

$$M^*_{\ell_M} \leftarrow M_{\ell_M} \;||\; \mathbf{GetMSB}(\tau, n - |M_{\ell_M}|),$$

where n denotes the block length. $M^*_{\ell_M}$ is then encrypted as described above for the final message block. $C^*_{\ell_M}$ is then split, where its $|M_{\ell_M}|$ most significant bits are used as the final bits of the ciphertext and the remaining bits as the $n - |M_{\ell_M}|$ most significant bits of the tag, T^α:

$$C_{\ell_M} \leftarrow \mathbf{GetMSB}(C^*_{\ell_M}, |M_{\ell_M}|), \quad T^\alpha \leftarrow \mathbf{GetMSB}(C^*_{\ell_M}, n - |M_{\ell_M}|).$$

The remaining bits of the tag are produced as shown next.

Algorithm 6.4 The procedures **GenerateTag** and **VerifyTag**.

GenerateTag$(\tau, X_{\ell_M}, Y_{\ell_M})$	**VerifyTag**$(T, X_{\ell_C}, Y_{\ell_C}, \tau, \tau')$
701: $X_{\ell_M+1} \leftarrow F_t(X_{\ell_M}) \oplus \tau \oplus L_T$	801: $X_{\ell_C+1} \leftarrow F_t(X_{\ell_C}) \oplus \tau \oplus L_T$
702: $Y_{\ell_M+1} \leftarrow E_K(X_{\ell_M+1})$	802: $Y_{\ell_C+1} \leftarrow E_K(X_{\ell_C+1})$
703: $C^*_{\ell_M+1} \leftarrow F_b(Y_{\ell_M}) \oplus Y_{\ell_M+1} \oplus L_T$	803: $C^*_{\ell_C+1} \leftarrow F_b(Y_{\ell_C}) \oplus Y_{\ell_C+1} \oplus L_T$
704: $(T^\beta, Z) \leftarrow \mathbf{Split}(C^*_{\ell_M+1}, \|M_{\ell_M}\|)$	804: $(T', Z) \leftarrow \mathbf{Split}(C^*_{\ell_C+1}, n - \|\tau'\|)$
705: **return** T^β	805: $T^\beta \leftarrow \mathbf{GetLSB}(T, n - \|\tau'\|)$
	806: $\tau^\alpha \leftarrow \mathbf{GetMSB}(\tau, \|\tau'\|)$
	807: **return** $\tau^\alpha = \tau'$ **and** $T' = T^\beta$

Authentication/Verification. To generate or to verify the authentication tag, POET processes the intermediate tag τ similarly to a message block M_i (cf. Algorithm 6.4 and Figure 6.3). The only difference is given by masking τ with an independent key L_T. When the length of the message is a multiple of n, the entire output $C^*_{\ell_M+1}$ (cf. Line 703) is used as a tag, which is transmitted together with the ciphertext when authenticating; for verification, $C^*_{\ell_C+1}$ is compared to the tag which was transmitted with the ciphertext.

When the message length is not a multiple of n, we already obtained the first $n - |M_{\ell_M}|$ bits of the tag (T^α) from the encryption of the final message block (cf. Line 104 of Algorithm 6.1). The remaining $|M_{\ell_M}|$ bits of the tag (T^β) are taken from the $|M_{\ell_M}|$ most significant bits of $C^*_{\ell_M+1}$ (cf. Line 704 of Algorithm 6.4); the rest of $C^*_{\ell_M+1}$ is discarded. The concatenation of $T^\alpha \;||\; T^\beta$ is simply used as a tag for authentication.

The verification consists of two steps: first, the $n - |M_{\ell_C}|$ least significant bits of $M^*_{\ell_C}$ are compared with the $n - |M_{\ell_C}|$ most significant bits of τ. Thereupon, the $|M_{\ell_C}|$ most

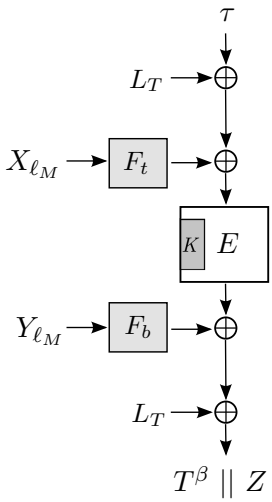

Figure 6.3.: Schematic illustration of the tag-generation procedure in POET.

significant bits of $C^*_{\ell_C+1}$ are compared to the $|M_{\ell_C}|$ least significant bits of T. If both checks are valid, the decrypted ciphertext is output; otherwise, the decryption fails (cf. lines 205 to 208 of Algorithm 6.1).

6.2. Instantiations for the ϵ-AXU Family of Hash Functions

We highly recommend to instantiate POET with AES-128 as a block cipher. For the ϵ-AXU families of hash functions F, we propose three different instantiations in the following:

1. POET with Galois-Field multiplications in $GF(2^{128})$,
2. POET with 4-round AES, and
3. POET with full-round AES.

POET with Galois-Field Multiplications. We recommend multiplications in $GF(2^{128})$, similar to the multiplication in AES-GCM [37] as universal hash function with an $\epsilon \approx 2^{-128}$. The family of hash functions F is then defined by $F_t(X) = X \cdot L_F^{top}$ or $F_b(X) = X \cdot L_F^{bot}$, depending on whether it is applied to the top or the bottom row.

When using multiplications in $GF(2^{128})$, one has to consider the risk of weak keys. As stressed by Saarinen in [49], $2^{128} - 1$ is not prime, so it produces some smooth-order multiplicative groups. Thus, one can explore a weak key with a probability about 2^{-96}. To avoid the risk of having weak multiplication keys (one for processing the header and two hash-function keys for processing the message), we propose to perform a checking on the keys L, L_F^{top}, and L_F^{bot} right after their generation phase. For each weak key, we choose a fresh unique constant $const_i$ with $1 \leq i \leq 3$, depending on which key is weak, re-generate the corresponding key, and check it again. This procedure can be repeated until none of the keys is weak. In addition, one can add a test function to assure that all keys are pairwise independent, and none of them represents a multiple of another one. Since this additional security measurement must be applied only at the time of key setup, and since only a small fraction of keys are weak, the effort for this can be considered negligible in

the long-term view.

POET with Four-Round AES. When trying to minimize the implementation footprint, it may be desirable to have an encryption scheme based on only a single primitive. Furthermore, as mentioned before, maximizing the throughput is often critical. Therefore, POET with four-round AES as family of keyed hash functions may be an excellent choice for restricted devices and/or devices with integrated AES-NI. In particular, the key schedule of the keyed hash function needs to be called only once for a given key.

The drawback of this solution would be a slightly lower number of message blocks that can be processed under the same key. As shown by Daemen et al. in [16], the four-round AES is a family of ϵ-AXU – under the reasonable assumption that all four round keys are independent – where ϵ can be upper bounded by

$$\epsilon \leq 1.88 \cdot 2^{-114} \approx 2^{-113}.$$

POET with Full-Round AES. As a strongly conservative variant, we propose to use full AES-128 as a family of hash functions. Under the common PRF assumption – where we assume that AES is indistinguishable from a random 128-bit permutation, this construction yields $\epsilon \approx 2^{-128}$.

6.3. Recommended Parameter Sets

Primary Recommendation: AES-POET-GF128 (A-POET-GF)

- Block cipher: AES-128.
- Hash function: One Galois-Field multiplication in $GF(2^{128})$ under the same polynomial of GCM: $f(x) = 1 + x + x^2 + x^7 + x^{128}$.
- Sizes: 128-bit key, 128-bit nonce, 128-bit state, 128-bit tag.

Secondary Recommendation: AES-POET-AES4 (A-POET-AES4)

- Block cipher: AES-128.
- Hash function: First Four Rounds of AES-128.
- Sizes: 128-bit key, 128-bit nonce, 128-bit state, 128-bit tag.

Tertiary Recommendation: AES-POET-AES128 (A-POET-AES128)

- Block cipher: AES-128.
- Hash function: AES-128.
- Sizes: 128-bit key, 128-bit nonce, 128-bit state, 128-bit tag.

6.4. Specification of POE

The encryption and decryption functions of POET– when considered without processing associated data and authentication – define a self-contained family of fast and secure on-line ciphers, called POE. While we concentrate on authenticated encryption in this

work, we can profit from considering the encryption process in an isolated fashion for our later security discussion of POET. Therefore, we briefly define the POE family of on-line ciphers. Note, POE is only defined for messages whose length is a multiple of n. The key-generation for POE is similar to POET (Algorithm 6.2), except the steps in lines 302 and 305 are neglected since POE considers neither associated data nor authentication.

Definition 6.2 (POE). *Let $k, n \geq 1$ be two integers, $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a block cipher, and $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a family of keyed ϵ-AXU hash functions. Further, let $K, L_F^{top}, L_F^{bot} \in \{0,1\}^k$ denote pairwise independent keys. Then, the encryption of POE and its inverse are defined by the procedures **Encrypt** and **Decrypt** as shown in Algorithm 6.5.*

Algorithm 6.5 The procedures **Encrypt** and **Decrypt** for POE.

Encrypt(M)	**Decrypt**(C)
101: $\ell_M \leftarrow \lvert M\rvert/n$, $X_0 \leftarrow Y_0 \leftarrow 1$	201: $\ell_C \leftarrow \lvert C\rvert/n$, $X_0 \leftarrow Y_0 \leftarrow 1$
102: **for** $i \leftarrow 1, \ldots, \ell_M$ **do**	202: **for** $i \leftarrow 1, \ldots, \ell_C$ **do**
103: $\quad X_i \leftarrow F_t(X_{i-1}) \oplus M_i$	203: $\quad Y_i \leftarrow F_b(Y_{i-1}) \oplus C_i$
104: $\quad Y_i \leftarrow E_K(X_i)$	204: $\quad X_i \leftarrow E_K^{-1}(Y_i)$
105: $\quad C_i \leftarrow F_b(Y_{i-1}) \oplus Y_i$	205: $\quad M_i \leftarrow F_t(X_{i-1}) \oplus X_i$
106: **end for**	206: **end for**
107: **return** $C \leftarrow (C_1 \,\|\, \ldots \,\|\, C_{\ell_M})$	207: **return** $M \leftarrow (M_1 \,\|\, \ldots \,\|\, M_{\ell_C})$

Chapter 7

The Parallel Version POET-m

This chapter defines a generalization of the POET family of on-line AE schemes – called POET-m, where the m arises from an m-periodic structure in the encryption and decryption. When using two instances of the ϵ-AXU function F, i.e., F_t and F_b (cf. Figure 7.1), POET-1 is identical to POET. Nevertheless, we claim that POET-m has similar security as POET. Therefore, the security proof of POET-m will soon be published on ePrint. Like POET, POET-m consists of three layers:

1. The top-row layer masks the incoming message blocks M_i and applies an ϵ-AXU hash function F.
2. The middle layer performs ECB encryption.
3. The bottom-row layer applies F to the encrypted blocks, and masks the outputs again, before they are uses as ciphertext blocks C_i.

Additionally, POET-m has two internal chaining values, X and Y, which are updated by an XOR with the values before and after the encryption layer, respectively. As for POET, the chaining ensures that each message block depends on all previous blocks.

A schematic illustration of the encryption process of POET-m is given in Figure 7.1. It is easy to see that each layer allows $m-1$ subsequent blocks to be processed in parallel. For the m-th block, the chaining values X and Y are updated with the hash functions in order to prevent cyclic patterns, and the procedure repeats for the following m blocks.

Given this structure, POET-m also makes use of the XEX approach, the PMAC design, and the idea of tag splitting to allow length-preserving encryption/decryption.

In the remainder of this chapter, we first provide a formal definition of POET-m. Thereupon, we describe the individual steps of the **key generation**, **header** and **message processing**, **tag generation** and **verification**. We use the same auxiliary functions as introduced for POET in Chapter 6.

Definition 7.1 (POET-m). *Let $m, n, k \geq 1$, be three integers. Let POET$-m = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an AE scheme, $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ a block cipher and $F : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a family of keyed, invertible ϵ-AXU hash functions. Furthermore, let H be the header (including the public message number N appended to its end), M the message, T the authentication tag, and C the ciphertext, with $H, M, C \in \{0,1\}^*$ and $T \in \{0,1\}^n$. Then, $\mathcal{E}$*

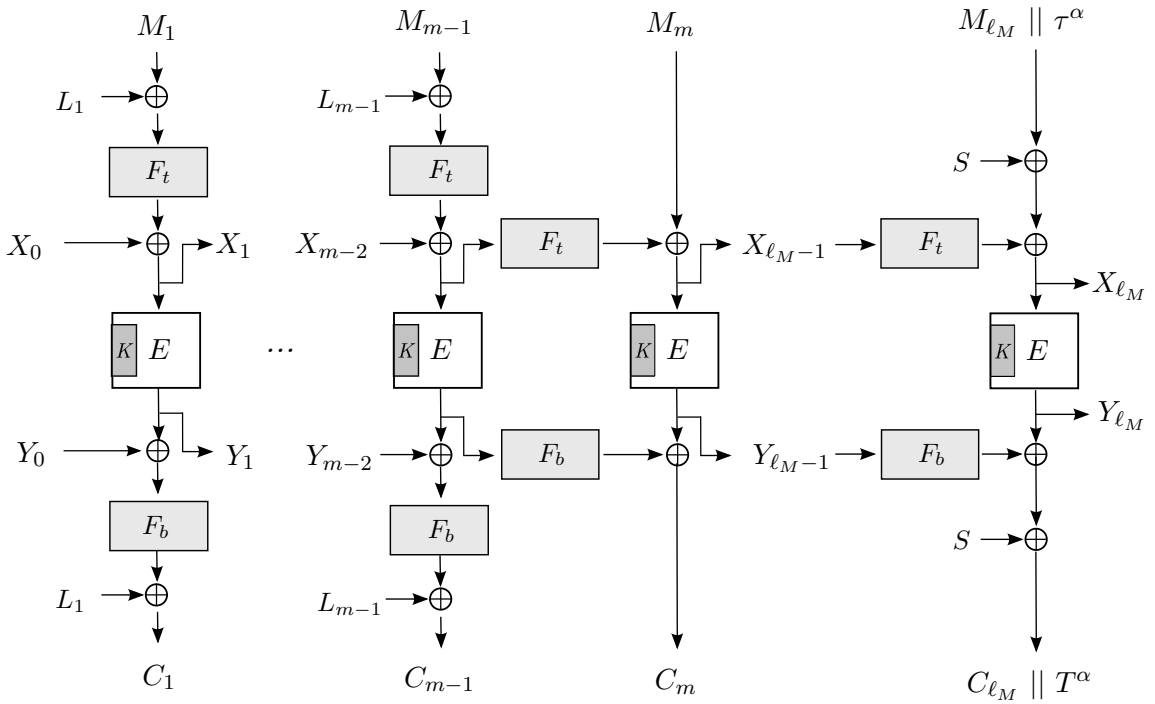

Figure 7.1.: Schematic illustration of the encryption process with POET-m for an $(m+1)$-block message $M = M_1, \ldots, M_{\ell_M}$. Note that the message blocks $M_1, \ldots, M_{m-1}$ can be processed in parallel. S denotes the encrypted message length, i.e., $S = E_K(|M|)$, $L_1, \ldots, L_{m-1}$ are the masking keys, F is an ϵ-AXU family of hash functions, and τ^α is taken from the most significant bits of the header processing to pad the final message block. Note that F_t and F_b use the keys L_F^{top} and L_F^{bot}. In comparison to POET, one can use $F_t = F_b$ i.e., $F_t(L_F^{top}, \cdot) = F_b(L_F^{bot}, \cdot) = F(L_F, \cdot)$ for some fixed key L_F (derived from the secret key SK) for POET-m.

*is given by procedure **EncryptAndAuthenticate**, $\mathcal{D}$ by procedure **DecryptAndVerify**, and $\mathcal{K}$ by procedure **GenerateKeys**, as shown in Algorithms 7.1 and 7.2.*

Algorithm 7.1 EncryptAndAuthenticate and **DecryptAndVerify**.

EncryptAndAuthenticate(H, M, m)	**DecryptAndVerify**(H, C, T, m)
101: $\ell_M \leftarrow \lceil M/n \rceil$	201: $\ell_C \leftarrow \lceil C/n \rceil$
102: $\tau \leftarrow$ **ProcessHeader**(H)	202: $\tau \leftarrow$ **ProcessHeader**(H)
103: $(C, X_{\ell_M}, Y_{\ell_M}) \leftarrow$ **Encrypt**(M, τ, m)	203: $(M, X_{\ell_C}, Y_{\ell_C}) \leftarrow$ **Decrypt**(C, T, τ, m)
104: $(C_{\ell_M}, T^\alpha) \leftarrow$ **Split**$(C_{\ell_M}, \lvert M_{\ell_M} \rvert)$	204: $(M_{\ell_C}, \tau') \leftarrow$ **Split**$(M_{\ell_C}, \lvert C_{\ell_C} \rvert)$
105: $T^\beta \leftarrow$ **GenerateTag**$(\tau, X_{\ell_M}, Y_{\ell_M})$	205: **if VerifyTag**$(T, X_{\ell_C}, Y_{\ell_C}, \tau, \tau')$ **then**
106: $T \leftarrow T^\alpha \,||\, T^\beta$	206: **return** M
107: **return** $(C_1 \,||\, \ldots \,||\, C_{\ell_M}, T)$	207: **end if**
	208: **return** $\perp$

Key Generation. Since we consider $F_t = F_b$, POET-m requires in total $m + 4$ pairwise independent k-bit keys as generated similar to POET, and used as follows:

- One key K for the block cipher.

- One masking key L for processing the header.
- One key L_F for the keyed family of hash functions F.
- $m-1$ masking keys $L_1, \dots L_{m-1}$ for encryption and decryption.
- One masking key L_T for generating the authentication tag.

The key generation is given in Algorithm 7.2. For the sake of simplicity, we write F in short for $F(L_F, \cdot)$.

Algorithm 7.2 The procedures **GenerateKeys** and **ProcessHeader**.

GenerateKeys(SK)
301: $K \leftarrow E_{SK}(const_0)$
302: $L \leftarrow E_{SK}(const_1)$
303: $L_F \leftarrow E_{SK}(const_2)$
304: $L_T \leftarrow E_{SK}(const_4)$
305: **for** $i \leftarrow 1, \dots, m-1$ **do**
306: $\quad L_i \leftarrow E_{SK}(const_{i+4})$
307: **end for**
308: **return** $(K, L, L_F, L_T, L_1, \dots, L_{m-1})$

ProcessHeader(H)
401: $\ell_H \leftarrow \lceil |H|/n \rceil$, $\Sigma \leftarrow 0^n$
402: **for** $i \leftarrow 1, \dots, \ell_H - 1$ **do**
403: $\quad \Sigma \leftarrow \Sigma \oplus E_K(H_i \oplus 2^{i-1}L)$
404: **end for**
405: **if** $|H_{\ell_H}| = n$ **then**
406: $\quad \tau \leftarrow E_K(H_{\ell_H} \oplus 2^{\ell_H - 1}3L \oplus \Sigma)$
407: **else**
408: $\quad H_{\ell_H} \leftarrow H_{\ell_H} \,||\, 10^*$
409: $\quad \tau \leftarrow E_K(H_{\ell_H} \oplus 2^{\ell_H - 1}5L \oplus \Sigma)$
410: **end if**
411: **return** τ

Header Processing. As one can easily see from Algorithm 7.2, the header processing is identical to that of POET. Therefore, we do not provide an extra figure.

Encryption/Decryption. The message-processing workflow is shown in Algorithm 7.3 and Figure 7.1. Let m denote the number of message blocks processed within one period. Each message block M_i for $1 \leq i \leq \ell_M - 1$ with $i \bmod m \not\equiv 0$ is masked by XORing the corresponding secret masking key L_i and the application of the ϵ-AXU function F. The output is then XORed using the previous chaining value X_{i-1} resulting in X_i, which is then used twice: (1) as input for the current block cipher invocation and (2) as the new chaining value in the top row. The output of the block-cipher call $Y_i = E_K(X_i)$ is then used twice: (1) as the new chaining value in the bottom-row and (2) is XORed with the previous chaining value Y_{i-1} to produce the input to the next invocation of F. The output of F is then XORed with the corresponding masking key L_i to compute the ciphertext block C_i. As for POET, the values X_0 and Y_0 are given by the intermediate tag τ.

The message blocks M_i with $i \bmod m \equiv 0$ are processed similar to an intermediate message block of POET. Thus, the chaining values X_{i-1} and Y_{i-1} are updated using F. The value $X_i = F(X_{i-1}) \oplus M_i$ is then used twice; (1) as input to E and (2) as the new top-row chaining value. The output $Y_i = E_K(X_i)$ is also used twice: (1) as the new chaining value for bottom-row and (2) is XORed with $F(Y_{i-1})$ to create the ciphertext block C_i. The decryption process is defined similarly in procedure **Decrypt**. The processing of the last message block (either with or without tag splitting) is similar to that of POET, except for the fact that POET-m can use the same ϵ-AXU function F in the top and bottom row.

Algorithm 7.3 The procedures **Encrypt** and **Decrypt**.

Encrypt(M, τ, m)

501: $\ell_M \leftarrow \lceil |M|/n \rceil$, $X_0 \leftarrow Y_0 \leftarrow \tau$
502: **for** $i \leftarrow 1, \ldots, \ell_M - 1$ **do**
503: $\quad j \leftarrow i \bmod m$
504: $\quad$ **if** $j \neq 0$ **then**
505: $\quad\quad X_i \leftarrow F(M_i \oplus L_j) \oplus X_{i-1}$
506: $\quad\quad Y_i \leftarrow E_K(X_i)$
507: $\quad\quad C_i \leftarrow F(Y_{i-1} \oplus Y_i) \oplus L_j$
508: $\quad$ **else**
509: $\quad\quad X_i \leftarrow F(X_{i-1}) \oplus M_i$
510: $\quad\quad Y_i \leftarrow E_K(X_i)$
511: $\quad\quad C_i \leftarrow F(Y_{i-1}) \oplus Y_i$
512: $\quad$ **end if**
513: **end for**
514: $S \leftarrow E_K(|M|)$
515: $\tau^\alpha \leftarrow$ **GetMSB**$(\tau, n - |M_{\ell_M}|)$
516: $M^*_{\ell_M} \leftarrow (M_{\ell_M} \mid\mid \tau^\alpha)$
517: $X_{\ell_M} \leftarrow F(X_{\ell_M - 1}) \oplus M^*_{\ell_M} \oplus S$
518: $Y_{\ell_M} \leftarrow E_K(X_{\ell_M})$
519: $C_{\ell_M} \leftarrow F(Y_{\ell_M - 1}) \oplus Y_{\ell_M} \oplus S$
520: $C \leftarrow (C_1 \mid\mid \ldots \mid\mid C_{\ell_M})$
521: **return** $(C, X_{\ell_M}, Y_{\ell_M})$

Decrypt(C, T, τ, m)

601: $\ell_C \leftarrow \lceil |C|/n \rceil$, $X_0 \leftarrow Y_0 \leftarrow \tau$
602: **for** $i \leftarrow 1, \ldots, \ell_C - 1$ **do**
603: $\quad j \leftarrow i \bmod m$
604: $\quad$ **if** $j \neq 0$ **then**
605: $\quad\quad Y_i \leftarrow F^{-1}(C_i \oplus L_j) \oplus Y_{i-1}$
606: $\quad\quad X_i \leftarrow E_K^{-1}(Y_i)$
607: $\quad\quad M_i \leftarrow F^{-1}(X_{i-1} \oplus X_i) \oplus L_j$
608: $\quad$ **else**
609: $\quad\quad Y_i \leftarrow F(Y_{i-1}) \oplus C_i$
610: $\quad\quad X_i \leftarrow E_K^{-1}(Y_i)$
611: $\quad\quad M_i \leftarrow F(X_{i-1}) \oplus X_i$
612: $\quad$ **end if**
613: **end for**
614: $S \leftarrow E_K(|C|)$
615: $T^\alpha \leftarrow$ **GetMSB**$(T, n - |C_{\ell_C}|)$
616: $C^*_{\ell_C} \leftarrow (C_{\ell_C} \mid\mid T^\alpha)$
617: $Y_{\ell_C} \leftarrow F(Y_{\ell_C - 1}) \oplus C^*_{\ell_C} \oplus S$
618: $X_{\ell_C} \leftarrow E_K^{-1}(Y_{\ell_C})$
619: $M_{\ell_C} \leftarrow F(X_{\ell_C - 1}) \oplus X_{\ell_C} \oplus S$
620: $M \leftarrow (M_1 \mid\mid \ldots \mid\mid M_{\ell_C})$
621: **return** $(M, X_{\ell_C}, Y_{\ell_C})$

Algorithm 7.4 The procedures **GenerateTag** and **VerifyTag**.

GenerateTag$(\tau, X_{\ell_M}, Y_{\ell_M})$

701: $X_{\ell_M + 1} \leftarrow F(X_{\ell_M}) \oplus \tau \oplus L_T$
702: $Y_{\ell_M + 1} \leftarrow E_K(X_{\ell_M + 1})$
703: $C^*_{\ell_M + 1} \leftarrow F(Y_{\ell_M}) \oplus Y_{\ell_M + 1} \oplus L_T$
704: $(T^\beta, Z) \leftarrow$ **Split**$(C^*_{\ell_M + 1}, |M_{\ell_M}|)$
705: **return** T^β

VerifyTag$(T, X_{\ell_C}, Y_{\ell_C}, \tau, \tau')$

801: $X_{\ell_C + 1} \leftarrow F(X_{\ell_C}) \oplus \tau \oplus L_T$
802: $Y_{\ell_C + 1} \leftarrow E_K(X_{\ell_C + 1})$
803: $C^*_{\ell_C + 1} \leftarrow F(Y_{\ell_C}) \oplus Y_{\ell_C + 1} \oplus L_T$
804: $(T', Z) \leftarrow$ **Split**$(C^*_{\ell_C + 1}, n - |\tau'|)$
805: $T^\beta \leftarrow$ **GetLSB**$(T, n - |\tau'|)$
806: $\tau^\alpha \leftarrow$ **GetMSB**$(\tau, |\tau'|)$
807: **return** $\tau^\alpha = \tau'$ **and** $T' = T^\beta$

Authentication/Verification. For the generation and verification of the authentication tag, POET-m applies a similar strategy as POET, with the only difference that POET-m can use the same ϵ-AXU function F for top and bottom row (cf. Algorithm 7.4). Therefore, we do not provide an extra figure.

Chapter 8

Security Analysis

This chapter analyzes the security analysis of POET. First, we provide a proof for the OPERM-CCA-security of POE. Thereupon, we prove the OPERM-CCA- security for POET. Finally, we prove the INT-CTXT-security of POET.

8.1. OPERM-CCA-Security of POE

Theorem 8.1 (OPERM-CCA-Security of POE). *Let $E \in \texttt{Block}(k,n)$ be a block cipher and $F : \{0,1\}^n \to \{0,1\}^n$ be an ϵ-AXU family of hash functions and F^{-1} its inverse. Then, it holds that*

$$\mathbf{Adv}^{\textit{OPERM-CCA}}_{\textit{POE}_{E,E^{-1}}}(q,\ell,t) \leq \ell^2 \cdot \epsilon + \frac{\ell^2}{2^n - \ell} + \mathbf{Adv}^{\textit{IND-SPRP}}_{E,E^{-1}}(\ell, \mathcal{O}(t)).$$

Proof. Let $\mathcal{A}$ be an OPERM-CCA-adversary with access to an oracle $\mathcal{O}$, which responds either with real encryptions/decryptions using $\mathsf{POET}_{E_K}(M)/\mathsf{POET}_{E_K^{-1}}(C)$ or random encryptions/decryptions using $P(M)$ or $P^{-1}(C)$ as in Definition 5.6. In the beginning, the oracle tosses a fair coin to obtain a bit b. Thereupon, $\mathcal{A}$ can query messages to $\mathcal{O}$. Depending on b, $\mathcal{A}$ obtains either "real" encryptions/decryptions for the messages it sends to $\mathcal{O}$, or just "random" outputs. Hence, the task for $\mathcal{A}$ is to guess b.

$\mathcal{A}$ asks at most q queries of a total length of at most ℓ blocks and stores each query together with the corresponding response from the oracle as tuples (M^i, C^i) in a query history $\mathcal{Q}$. Note that we assume wlog. that $\mathcal{A}$ will not make queries to which it already knows the answer. It is easy to see that we can upper bound Equation (5.6) as (cf. [21], Sec. 4)

$$\left|\Pr\left[\mathcal{A}^{\mathsf{POE}_E,\mathsf{POE}^{-1}_{E^{-1}}} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\mathsf{POE}_\pi,\mathsf{POE}^{-1}_{\pi^{-1}}} \Rightarrow 1\right]\right| \tag{8.1}$$

$$+ \left|\Pr\left[\mathcal{A}^{\mathsf{POE}_\pi,\mathsf{POE}^{-1}_{\pi^{-1}}} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{P(\cdot),P^{-1}(\cdot)} \Rightarrow 1\right]\right|, \tag{8.2}$$

where $\pi : \{0,1\}^n \to \{0,1\}^n$ denotes an n-bit random permutation that was chosen at random from the set of all n-bit random permutations, and π^{-1} denotes its inverse.

The difference in Equation (8.1) can be upper bounded by the IND-SPRP-advantage of $\mathcal{A}$ to distinguish E from a random permutation

$$\mathbf{Adv}^{\mathsf{IND\text{-}SPRP}}_{E,E^{-1}}(\ell, \mathrm{O}(t)).$$

Proof Idea for the Remainder. It remains to study the difference in (8.2), which refers to the advantage of $\mathcal{A}$ to distinguish POE instantiated with a random permutation π from $P(\cdot), P^{-1}(\cdot)$. From the structure of POE, we can identify two cases: (1) collisions between internal values of POE (COLL), or (2) no collisions occur (NOCOLLWIN). From the law of total probability follows that we can rewrite (8.2) as

$$\begin{aligned} &\Pr\left[\mathrm{COLL}\right] \cdot \Pr\left[\mathrm{COLLWIN}\right] + \Pr\left[\neg\,\mathrm{COLL}\right] \cdot \Pr\left[\mathrm{NOCOLLWIN}\right] \\ \leq\ &\Pr\left[\mathrm{COLL}\right] + \Pr\left[\mathrm{NOCOLLWIN}\right], \end{aligned}$$

with

$$\Pr\left[\mathrm{NOCOLLWIN}\right] = \left|\Pr\left[\mathcal{A}^{\mathsf{POE}_\pi, \mathsf{POE}^{-1}_{\pi^{-1}}} \Rightarrow 1 \,|\, \neg\,\mathrm{COLL}\right] - \Pr\left[\mathcal{A}^{P(\cdot), P^{-1}(\cdot)} \Rightarrow 1\right]\right|. \quad (8.3)$$

COLL. In this case, $\mathcal{A}$ tries to distinguish POE from random by exploiting some collision between internal values. Since π is random permutation, any "fresh" (i.e., not previously queried) input to $\pi(\cdot)$ or its inverse $\pi^{-1}(\cdot)$ produces a random output. This implies for the internal values of POE:

- For any fresh X_i, the result of $Y_i = \pi(X_i)$ is also random in encryption direction, and so are the resulting ciphertext outputs $C_i = Y_i \oplus F_b(Y_{i-1})$.
- For any fresh Y_i in decryption direction, the result of $X_i = \pi^{-1}(Y_i)$ is also random, and so are the resulting decrypted message blocks $M_i = X_i \oplus F_t(X_{i-1})$.

It is easy to see that there are two possible subcases: an internal collision in the top row ($\mathrm{COLL}^{\mathrm{enc}}$), or an internal collision in the bottom row ($\mathrm{COLL}^{\mathrm{dec}}$). COLL then represents the event that either (or both) of these subcases occurred:

$$\mathrm{COLL} = \mathrm{COLL}^{\mathrm{enc}} \vee \mathrm{COLL}^{\mathrm{dec}}.$$

Due to the symmetric structure of POE, it applies that $\Pr[\mathrm{COLL}^{\mathrm{enc}}] = \Pr[\mathrm{COLL}^{\mathrm{dec}}]$. The individual probabilities for the events $\mathrm{COLL}^{\mathrm{enc}}$ and NOCOLLWIN are upper bounded by $\frac{\ell^2}{2} \cdot \epsilon$. The proof is given in Lemma B.1 in Appendix B.

NOCOLLWIN. The probability for the event NOCOLLWIN can be upper bounded $\frac{\ell^2}{2^n - \ell}$. The proof is given in Lemma B.2 in Appendix B.

Our claim follows from summing up the individual terms. □

8.2. OPERM-CCA-Security of POET

From the OPERM-CCA-security bound for POE we can now easily derive the respective advantage for POET.

Theorem 8.2 (OPERM-CCA-Security of POET). *Let $E \in \texttt{Block}(k,n)$ Then, it applies that*

$$\mathbf{Adv}^{\mathsf{OPERM\text{-}CCA}}_{\mathsf{POET}_{E,E^{-1}}}(q,\ell,t) \leq \frac{\ell^2}{2^n} + \ell^2 \cdot \epsilon + 2 \cdot \max\left\{\ell \cdot q \cdot \epsilon, \frac{q^2}{2^n - q}\right\} + \frac{(\ell+2q)^2}{2^n - (\ell+2q)} + \mathbf{Adv}^{\mathsf{IND\text{-}SPRP}}_{E,E^{-1}}(\ell+2q, O(t)).$$

Proof. The proof follows the ideas of Theorem 8.1. We can write Equation (5.6) as

$$\left|\Pr\left[\mathcal{A}^{\mathsf{POET}_E,\mathsf{POET}^{-1}_{E^{-1}}} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\mathsf{POET}_\pi,\mathsf{POET}^{-1}_{\pi^{-1}}} \Rightarrow 1\right]\right| \tag{8.4}$$

$$+ \left|\Pr\left[\mathcal{A}^{\mathsf{POET}_\pi,\mathsf{POET}^{-1}_{\pi^{-1}}} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{P(\cdot),P^{-1}(\cdot)} \Rightarrow 1\right]\right|, \tag{8.5}$$

where $\pi : \{0,1\}^n \to \{0,1\}^n$ denotes an n-bit random permutation that was chosen at random from the set of all n-bit random permutations, and π^{-1} denotes its inverse.

The difference in Equation (8.4) can be upper bounded by the $\mathsf{IND\text{-}SPRP}$-advantage of $\mathcal{A}$ to distinguish E from a random permutation π

$$\mathbf{Adv}^{\mathsf{IND\text{-}SPRP}}_{E,E^{-1}}(\ell+2q, \mathrm{O}(t)).$$

The additional term $2q$ results from the fact that the tag-generation of POET requires – compared to POE– two additional calls to the block cipher for the encryption of the message length and the tag generation.

Proof Idea for the Remainder. It remains to study the difference in (8.5), which refers to the advantage of $\mathcal{A}$ to distinguish POET instantiated with a random permutation π from $P(\cdot), P^{-1}(\cdot)$. From the structure of POET we can identify two cases: (1) collisions between internal values of POET (COLL), or (2) no collisions occur (NOCOLLWIN). From the law of total probability follows that we can rewrite (8.5) as

$$\begin{aligned} &\Pr[\mathrm{COLL}] \cdot \Pr[\mathrm{COLLWIN}] + \Pr[\neg\,\mathrm{COLL}] \cdot \Pr[\mathrm{NOCOLLWIN}] \\ \leq\ &\Pr[\mathrm{COLL}] + \Pr[\mathrm{NOCOLLWIN}], \end{aligned}$$

with

$$\Pr[\mathrm{NOCOLLWIN}] = \left|\Pr\left[\mathcal{A}^{\mathsf{POET}_\pi,\mathsf{POET}^{-1}_{\pi^{-1}}} \Rightarrow 1 \,|\, \neg\,\mathrm{COLL}\right] - \Pr\left[\mathcal{A}^{P(\cdot),P^{-1}(\cdot)} \Rightarrow 1\right]\right|. \tag{8.6}$$

COLL. In this case, $\mathcal{A}$ tries to distinguish POE from random by exploiting some collision between internal values. Since π is random permutation, any "fresh" (i.e., not previously queried) input to $\pi(\cdot)$ or its inverse $\pi^{-1}(\cdot)$ produces a random output. This implies for the internal values of POE:

- For any fresh X_i, the result of $Y_i = \pi(X_i)$ is also random in encryption direction, and so are the resulting ciphertext outputs $C_i = Y_i \oplus F_b(Y_{i-1})$.

- For any fresh Y_i in decryption direction, the result of $X_i = \pi^{-1}(Y_i)$ is also random, and so are the resulting decrypted message blocks $M_i = X_i \oplus F_t(X_{i-1})$.

From the definition of POET we can derive the following subcases:

1. $\mathbf{COLL^{ad}}$: $\mathcal{A}$ found a collision for two distinct headers $H \neq H'$: $\mathbf{ProcessHeader}(H) = \mathbf{ProcessHeader}(H')$.
2. $\mathbf{COLL^{enc}}$: For two distinct tuples (X_{i-1}, M_i) and (X'_{j-1}, M'_j) in one or two encryption query results $(M, C), (M', C') \in \mathcal{Q}$, the resulting top-row chaining values $X_i = X'_j$ collide.
3. $\mathbf{COLL^{dec}}$: For two distinct tuples (Y_{i-1}, C_i) and (Y'_{j-1}, C'_j) from one or two decryption query results $(M, C), (M', C') \in \mathcal{Q}$, the resulting bottom-row chaining values $Y_i = Y'_j$ collide.
4. $\mathbf{COLL^{lmb}}$: For two distinct tuples (X_{i-1}, M_i) and (X'_{j-1}, M'_j) in one or two encryption query results (M, C), $(M', C') \in \mathcal{Q}$, the resulting top-row chaining values $X_i = X'_j$ collide, when M_i and/or M'_j are the last blocks of M and M', respectively.
5. $\mathbf{COLL^{lcb}}$: For two distinct tuples (Y_{i-1}, C_i) and (Y'_{j-1}, C'_j) in one or two decryption query results (M, C), $(M', C') \in \mathcal{Q}$, the resulting top-row chaining values $Y_i = Y'_j$ collide, when C_i and/or C'_j are the last blocks of C and C', respectively.

Moreover, we define a compound event COLL which reflects the event that either or multiple events of $\mathrm{COLL}^{\mathrm{ad}}$, $\mathrm{COLL}^{\mathrm{enc}}$, $\mathrm{COLL}^{\mathrm{dec}}$, $\mathrm{COLL}^{\mathrm{lmb}}$, $\mathrm{COLL}^{\mathrm{lcb}}$ occurred:

$$\mathrm{COLL} = \mathrm{COLL}^{\mathrm{ad}} \vee \mathrm{COLL}^{\mathrm{enc}} \vee \mathrm{COLL}^{\mathrm{dec}} \vee \mathrm{COLL}^{\mathrm{lmb}} \vee \mathrm{COLL}^{\mathrm{lcb}}. \tag{8.7}$$

Note that Pr [NOCOLLWIN] reflects the success probability of $\mathcal{A}$ to distinguish $\mathsf{POET}_{E,E^{-1}}$ from a random OPERM. Furthermore, due to the symmetric structure of POET, we have $\Pr[\mathrm{COLL}^{\mathrm{enc}}] = \Pr[\mathrm{COLL}^{\mathrm{dec}}]$ and $\Pr[\mathrm{COLL}^{\mathrm{lmb}}] = \Pr[\mathrm{COLL}^{\mathrm{lcb}}]$.

We can upper bound the probabilities of the remaining events as follows:

- $\Pr[\mathbf{COLL^{ad}}] = \ell^2/2^n$. The proof is given by Theorem 8.3.
- $\Pr[\mathbf{COLL^{enc}}] = \frac{\ell^2}{2} \cdot \epsilon$. The proof is given in Lemma B.1 in Appendix B.
- $\Pr[\mathbf{COLL^{lmb}}] = \max\left\{\ell \cdot q \cdot \epsilon, \frac{q^2}{2^n - q}\right\}$. The proof is given in Lemma B.3 in Appendix B.
- $\Pr[\mathbf{NOCOLLWIN}] = \frac{(\ell+2q)^2}{2^n-(\ell+2q)}$. The proof is given in Lemma B.2 in Appendix B. Note that the analysis of the event NOCOLLWIN is similar to the NOCOLLWIN event analyzed in Chapter 8.1, except for the fact that for POET one has to consider $\ell + 2q$ blocks since the tag generation requires two additional block-cipher calls per query.

Our claim follows from summing up the individual terms. □

Since our header-processing step is similar to PMAC, we recall the security bound from [45].

Theorem 8.3 (Security of PMAC [45]/Upper Bound for $\mathbf{COLL^{ad}}$). *Let $\mathcal{A}$ be an adversary which asks at most q queries of total length at most ℓ to to an oracle which uses either PMAC or a uniformly at random chosen PRF. Then, the maximal advantage of $\mathcal{A}$*

to distinguish PMAC from a PRF is upper bounded by

$$\mathbf{Adv}^{\mathrm{PRF}}_{PMAC[Perm(n)]}(\mathcal{A}) \leq \ell^2/2^n.$$

8.3. INT-CTXT-Security Analysis of POET

Lemma 8.4 (INT-CTXT-Security of POET). *Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a POET scheme as defined in Definition 6.1. Then, it applies*

$$\mathbf{Adv}^{\mathit{INT\text{-}CTXT}}_{\Pi}(q, \ell, t) \leq \frac{\ell^2 + q}{2^n} + \frac{q}{2^{n/2} - q} + \mathbf{Adv}^{\mathit{OPERM\text{-}CCA}}_{\mathit{POE}_{E,E^{-1}}}(q, \ell, t) + \mathbf{Adv}^{\mathit{IND\text{-}SPRP}}_{E,E^{-1}}(\ell + 2q, O(t)).$$

Proof. We define $\mathcal{A}$ as an INT-CTXT-adversary with access to an oracle $\mathcal{O}$, as in Game $G_{\mathsf{INT\text{-}CTXT}}$ (cf. Figure 5.1). $\mathcal{A}$ can query encryption and decryption messages to $\mathcal{O}$. Similar to our OPERM-CCA proof, we say that $\mathcal{A}$ stores each query it asks to $\mathcal{O}$ together with the oracle response as a tuple (H, M, C, T) in a query history $\mathcal{Q}$, and $\mathcal{A}$ does not ask queries to which it already knows the answer. $\mathcal{A}$ wins if it can predict the correct authentication tag T for a query that was not previously asked, i.e., that is not in $\mathcal{Q}$.

We assume that POET is an OPERM, which implies that the generated tags T are random values. Hence, the task for $\mathcal{A}$ is to find a forgery (H, C, T) with $(H, C, *, T) \notin \mathcal{Q}$ and $\mathcal{D}(H, C, T) \neq \perp$.

For the remainder, we replace the block cipher E by a random permutation π. It is easy to see that the advantage for the adversary from this replacement can be upper bounded by

$$\mathbf{Adv}^{\mathsf{IND\text{-}SPRP}}_{E,E^{-1}}(\ell + 2q, \mathrm{O}(t)).$$

Next, we analyze the possible cases that an adversary can win, depending on the length of C_{ℓ_C} as well as on the *freshness* of the values H, C, and T.

H is fresh. When H is fresh, since the header processing step of POET is secure up to the birthday bound, τ will also be a fresh random value. Therefore, we can upper bound the success probability of $\mathcal{A}$ to find a forgery by

$$\frac{\ell^2 + q}{2^n}.$$

Since we already considered the case of fresh header, for the remainder of this proof we assume that H is old, i.e., it was queried before by $\mathcal{A}$. We distinguish between two different scenarios depending on whether the last message block is full or not.

Scenario 1: ($|C_{\ell_C}| = n$). For the case when the last message block is full, we have to consider the following two mutually exclusive cases depending on whether C is a fresh value or C is old and T is fresh.

Case (1): C is fresh. Since C is fresh and POET is an OPERM, the probability that the decryption of T yields the correct value τ is 2^{-n}. Therefore, in this case, we can upper bound the success probability of $\mathcal{A}$ to find a forgery by

$$\mathbf{Adv}^{\mathsf{OPERM\text{-}CCA}}_{\mathsf{POE}_{\pi,\pi^{-1}}}(q,\ell,t) \leq \frac{q}{2^n}.$$

Case (2): C is old and T is fresh. Since POET maps each tuple (H, C) uniquely to one value $T = \pi(F_t(X_{\ell_C}) \oplus \tau \oplus L_T) \oplus F_b(Y_{\ell_C}) \oplus L_T$, there can not exist a second value $T' \neq T$ which is also valid. Hence, the success probability for $\mathcal{A}$ in this case is 0.

Scenario 2: ($|C_{\ell_C}| \neq n$). In this part we consider the INT-CTXT-security of POET for messages of arbitrary length. Again, we consider two mutually exclusive cases as the above.

Case (1): C is fresh. For this case we already know that X_{ℓ_M} and Y_{ℓ_M} must be fresh; otherwise, $\mathcal{A}$ could have found an internal collision that violates the OPERM-CCA security of POET. The probability of this event can be upper bounded by the OPERM-CCA advantage:

$$\mathbf{Adv}^{\mathsf{OPERM\text{-}CCA}}_{\mathsf{POE}_{\pi,\pi^{-1}}}(q,\ell,t).$$

In the following, we consider the probability that the condition $\tau^\alpha = \tau'$ from Line 807 of procedure **VerifyTag** (see Section 6) holds. Since $Y_{\ell_C} = F_b(Y_{\ell_C-1}) \oplus C_{\ell_C} \mid\mid T^\alpha \oplus S$ is fresh, X_{ℓ_C} will also be a fresh random value. $\mathcal{A}$ now has to find a collision in the $n - |M_{\ell_C}|$ least significant bits of $M_{\ell_C} \mid\mid \tau^\alpha$, i.e., it has to find a collision of the form

$$F_t(X_{\ell_C-1}) \oplus X_{\ell_C} \oplus S = F_t(X'_{\ell_{C'}-1}) \oplus X'_{\ell_{C'}} \oplus S'.$$

Since it holds that $X_{\ell_C} \neq X'_{\ell_{C'}}$, it is easy to see that when $S = S'$ the success probability of collision is 0. Thus, we consider the case when $S \neq S'$. Then, the success probability of $\mathcal{A}$ to obtain $\tau^\alpha = \tau'$ for a single query can be upper bounded by

$$\Pr_\alpha = \max_{M_{\ell_C}} \{\Pr[F_t(X_{\ell_C-1}) \oplus X_{\ell_C} \oplus S = (M_{\ell_C} \mid\mid \tau^\alpha)]\} \leq \frac{1}{2^{(n-|M_{\ell_C}|)} - q}$$

with $X_{\ell_C} = \pi^{-1}(Y_{\ell_C})$. For q queries, the probability for $\tau^\alpha = \tau'$ is then at most

$$\frac{q}{2^{(n-|M_{\ell_C}|)} - q}.$$

We can use a similar argument to upper bound the probability of $T^\beta = T'$.

$$\Pr_\beta = \max_Z \left\{\Pr[F_b(Y_{\ell_C}) \oplus Y_{\ell_C+1} \oplus L_T = (T^\beta \mid\mid Z)]\right\} \leq \frac{1}{2^{|M_{\ell_C}|} - q}$$

with $Y_{\ell_C+1} = \pi(X_{\ell_C+1})$. Then, the probability for q queries is at most

$$\frac{q}{2^{|M_{\ell_C}|} - q}.$$

Note that the (total) success probability of this case depends on the length $|C_{\ell_C}|$. Therefore, we can distinguish between the three following subcases:

Subcase (2.1): $|C_{\ell_C}| < n/2$**.** In this case, we can upper bound $\Pr_\alpha$ by $\frac{1}{2^{n/2}-q}$ and $\Pr_\beta$ by 1. Hence, the total success probability for q queries is at most

$$\Pr_\alpha \cdot \Pr_\beta \leq \frac{q}{2^{n/2} - q}.$$

Subcase (2.2): $|C_{\ell_C}| = n/2$**.** In this case, we can upper bound $\Pr_\alpha$ by $\frac{1}{2^{n/2}-q}$ and $\Pr_\beta$ by $\frac{1}{2^{n/2}-q}$. Hence, the total success probability for q queries is at most

$$\Pr_\alpha \cdot \Pr_\beta \leq \frac{q^2}{2^n - q \cdot 2^{n/2+1} + q^2}.$$

Subcase (2.3): $|C_{\ell_C}| > n/2$**.** In this case, we can upper bound $\Pr_\alpha$ by 1 and $\Pr_\beta$ by $\frac{1}{2^{n/2+1}-q}$. Hence, the total success probability for q queries is at most

$$\Pr_\alpha \cdot \Pr_\beta \leq \frac{q}{2^{n/2+1} - q}.$$

Since all three subcases are mutually exclusive, we can upper bound the success probability for $q \leq 2^{n/2-2}$ queries by

$$\max\left\{\frac{q}{2^{n/2} - q}, \frac{q^2}{2^n - q \cdot 2^{n/2+1} + q^2}, \frac{q}{2^{n/2+1} - q}\right\} \leq \frac{q}{2^{n/2} - q}.$$

Case (2): C is old and T is fresh. Since POET maps each tuple (H, C) uniquely to one tag, there cannot exist a second value $T' \neq T$ which is valid. Hence, the success probability of $\mathcal{A}$ in this case is 0.

Due to the fact that Case (1) and (2) are mutually exclusive, we can upper bound the success probability by

$$\frac{q}{2^{n/2} - q}.$$

Since scenario 1 and scenario 2 are mutually exclusive, we can upper bound the success probability for q queries by

$$\begin{aligned}&\mathbf{Adv}^{\mathsf{OPERM\text{-}CCA}}_{\mathsf{POE}_{\pi,\pi^{-1}}}(q,\ell,t) + \max\left\{\frac{q}{2^n}, \frac{q}{2^{n/2} - q}\right\} \\ \leq\ &\mathbf{Adv}^{\mathsf{OPERM\text{-}CCA}}_{\mathsf{POE}_{\pi,\pi^{-1}}}(q,\ell,t) + \frac{q}{2^{n/2} - q}.\end{aligned}$$

Our claim follows from summing up the individual terms. □

Chapter 9

Implementation Aspects

POE and POET can be efficiently implemented in both soft- and hardware. Though, our reference implementation of POET is not supposed to be optimized for the large variety of supported platforms. However, in the majority of cases, the block cipher will be AES, and the hash function either Galois-Field multiplication or round-reduced AES. Therefore, the present section recalls the state-of-the-art for implementations of the AES and Galois-Field multiplications on 64-, 32-, and 8-bit processors as well as with and without native instructions (NIs). Then, we derive estimations for optimized versions from the existing figures. An optimized implementation for POET will then be available at mid of April 2014.

9.1. AES Implementations

Implementations without NI. Using standard optimization techniques (such as combined shift-and-mask instructions, scaled-index loads, etc.), Bernstein and Schwabe showed in 2008 [8] that the AES can be implemented for a 64-bit architecture without native instructions to run at about 10 clock cycles per byte (cpb). In 2009, Käsper and Schwabe [20] improved these results with a bitsliced implementation of the AES that exploited the *Streaming SIMD Extension instructions* (`SSE1-SSE5`). Their implementations achieved about 7.59 cpb on an Intel Core2 Q6600 (Kentsfield) and about 6.92 cpb on a Intel i7-920 CPU (Bloomfield). For 32-bit processors, Bernstein and Schwabe achieved a throughput of about 14.13 cpb on a Pentium 4 f12 (Willamette). For an example of a mobile 32-bit processor, we can use the figures for the ARM Cortex-A8 by Krovetz and Rogaway [36].

Concerning off-the-shelf 8-bit processors, Osvik et al. [39] showed at the FSE 2010, that the AES can be implemented at speeds of around 124.6 cpb on an AVR Atmel. Considering slightly modified devices, Tillich and Herbst [54] proposed at CT-RSA 2008 an enhancement for 8-bit AVR cores, which allowed to perform an AES encryption at a speed of about 78.7 cpb (1,259 cycles/block) on an Atmel ATmega128, with additional costs of about 1,100 gates. Table 9.1 summarizes the performance figures for the AES.

Platform	CPU	Cbp/Core	Ref.
64 bit	Intel i7-920 (Bloomfield)	6.92	[20]
	Intel Core 2 Q9550 (Yorkfield)	7.59	[20]
	Intel Core 2 Q6600 (Kentsfield)	9.32	[20]
32 bit	Intel Pentium 4 f12, x86 (Willamette)	14.13	[8]
	ARM Cortex-A8 (OpenSSL)	25.40	[36]
8 bit	Atmel AVR ATmega128 (extended)	78.70	[54]
	Atmel AVR AT90USB646	124.60	[39]

Table 9.1.: Speed of existing software implementations for one encryption of the AES (without key schedule) for common platforms. Cpb = cycles per byte.

Implementations with NI. In 2010, Intel [24] introduced native instructions for the AES encryption and decryption, which are nowadays supported by all modern Intel (Sandy Bridge, Ivy Bridge, Haswell series) and AMD (Bulldozer, Piledriver, and Jaguar series) processors. The *AES New Instruction Set* contains six constant-time CPU instructions: `aesenc` (one-round encryption), `aesenclast` (last-round encryption), `aesdec` (single-round decryption), `aesdeclast` (last-round decryption), `aesimc` (inverse MixColumns), and `aeskeygenassist` for faster key scheduling. On recent processors, they achieved a throughput of 1-2 and a latency of 7-8 cycles.

9.2. Implementations of Galois-Field Multiplication

Implementations without NI. Käsper and Schwabe showed in [20] that a multiplication in $GF(2^{128})$ can be efficiently implemented in software with table lookups at a speed of about three cpb on 64-bit platforms. For 32-bit implementations, one can use, e.g., the `mpFq` library by Gaudry and Thomé [14].

Implementations with NI. In 2010, Intel [26] integrated the `pclmulqdq` instruction into its Westmere processor series for multiplication in $GF(2^{128})$ to increase the performance of AES-GCM. The `pclmulqdq` instruction set takes two 128-bit factors and a byte as input and performs carry-less multiplication of one 64-bit half of each operand; the additional byte parameter then determines which halves are used.

Two additional methods for multiplications in $GF(2^{128})$ optimize the speed of `pclmulqdq` on Haswell processors. The method by Jankowski and Laurent [31] is limited to GCM; the method by Gueron [25] also targets but is not limited to GCM. The latter allows to perform a multiplication at about 2.4 cpb when processing a single message.

9.3. Performance of POET

Implementations without NI. From the existing implementations, we can derive estimations for optimized implementations of POET, using the AES as a block cipher and either Galois- Field multiplication or four-round AES for universal hashing, on different platforms without NI. Therefore, we sum up the costs for one AES encryption and two

multiplications, or for 18 rounds of AES, respectively. For either version, we add an overhead of one cpb for the chaining XOR operations.

For 64-bit platforms, we can use the figures from Käspar and Schwabe [20], which give a first estimate that POET-1 (identical to POET) with $GF(2^{128})$ is supposed to run at a speed of about $3+7+3+1 = 14$ cpb on platforms with native processor instructions. For the version with four-round AES, we expect POET to also run at about $1.8 \cdot 7+1 \approx 14$ cpb.

For 32-bit implementations, we can use the figures from Krovetz and Rogaway [36] to estimate the performance on an ARM Cortex-A8. Therefore, we approximate the costs for an AES encryption with 25.4 cpb from their figures for AES-CTR. To approximate the costs of a multiplication in $GF(2^{128})$, we use the difference of their performance figures for GCM and CTR, which yields about $50.8 - 25.4 \approx 25.4$ cpb. Therefore, we have an upper bound of $25.4+2 \cdot 25.4+1 \approx 78$ cpb for POET with AES and GF, and $1.8 \cdot 25.4+1 \approx 46$ cpb with four-round AES for a mobile 32-bit CPU. However, there clearly exist various more powerful 32-bit CPUs. For 8-bit implementations, POET with four-round AES will be in favor to have a single primitive. We expect POET to run at about 250 cpb on a non-modified 8-bit Atmel AVR CPU.

		Cpb/Core	
Platform	**CPU**	**AES + GF**	**AES + AES-4**
64-bit (with NI)	Intel Haswell	3.92 [12]	≈ 3
64-bit (without NI)	Intel Bloomfield	≈ 14	≈ 14
32-bit (without NI)	ARM Cortex-A8	≈ 78	≈ 46
8-bit	Atmel AVR ATmega128	–	≈ 250

Table 9.2.: Estimated speeds of software implementations of POET for processing a single message of ≥ 2048 bytes on common platforms. Cpb = cycles per byte.

Optimization Potential of POET. Clearly, optimized implementations can profit from the fully parallel design of PMAC for the header-processing step. But, the encryption and decryption steps of POET can also benefit from several aspects of pipelining and/or parallelism.

Galois-Field multiplications can be fully parallelized with an approach that was pointed out by [35]. For instance, consider four subsequent blocks of a message, $M_1 \,||\, \ldots \,||\, M_4$. With Galois-Field multiplications, the input for the second block-cipher call in POET is given by $K^2 + KM_1 + M_2$. Instead of sequentially multiplying with K, adding M_3, multiplying with K and adding M_4, one can compute the whole process in parallel:

- For the third block-cipher call: $K \cdot (K^2 + KM_1 + M_2) + M_3$.
- For the fourth block-cipher call: $K^2 \cdot (K^2 + KM_1 + M_2) + KM_3 + M_4$.

This approach increases the total number of multiplications, but decreases the latency. Given c cores and c subsequent message blocks to process, this approach reduces the latency of POET from c hash-function calls to $\mathcal{O}(\log c)$.

For instance, the multiplications can be performed in parallel for a set of eight subsequent message blocks, which are XORed using the existing 128-bit XOR instructions. The encryption allows each block to be processed independently from the other. On Haswell

processors, the AES round function has a latency of 7-8 cycles and an inverse throughput of a single cycle. As pointed out by Andreeva et al. [2], one can efficiently utilize the CPU-pipeline by applying the first AES round to each message block in the set, then applying the second round to each block, etc. In addition, one can profit from the available *Intel Advanced Vector Extensions* (`AVX`) on Haswell processors. After encryption, the same strategy as the above can be used to reduce the latency of multiplications in the bottom row.

Implementations with NI. In a recent study, Bogdanov et al. [12] compared several AE schemes, including POET (the version published at the FSE 2014), COPA, COBRA, and McOE-G. For an outlook of the performance of POET with NI, Table 9.3 summarizes the figures by Bogdanov et al. The authors showed that POET with the AES as cipher and Galois-Field multiplication as a family of hash functions runs at about 3.92 clock cycles per byte when encrypting a single message, and at about 2.1 clock cycles per byte when processing multiple messages in parallel. While their study considered only POET with Galois-Field multiplications, we expect POET with four-round AES to run slightly faster. Further, the authors showed that a recent implementation of COPA (requiring 20 AES rounds per message block) can run in 1.76 clock cycles per byte for a single message. Thus, we expect an optimized implementation of POET instantiated with four-round AES (hence, 18 AES rounds per block) to run at about < 3 cpb when encrypting in the single-message scenario, and at about < 2 cpb in the multi-message scenario.

	Message Length (bytes)				
Mode	**128**	**256**	**512**	**1024**	**2048**
Single message	4.61	4.24	4.13	4.02	3.92
Multiple messages	2.37	2.27	2.17	2.12	2.10
Speed-up for multi. messages	×1.95	×1.87	×1.90	×1.90	×1.87

Table 9.3.: Speed measurements in cycles per bytes of implementations of POET with NI on a single core of a 64-bit Intel Core i5-4300U at 1900 MHz, by Bogdanov et al. [12].

9.4. Hardware Implementations

Hardware implementations of POET can profit from the wide range of existing AES implementations. Galois-Field multiplications are likewise well-suited for hardware implementations, as McGrew and Viega already pointed out for GCM in [37]. Paar [40] summarized four methods with different trade-offs of speed and area, which are recalled in Table 9.4.

Method	**Time**	**Area**
Parallel	1	$\mathcal{O}(q^2)$
Digit Serial [51]	q/D	$\mathcal{O}(q \cdot D)$
Bit Serial	q	$\mathcal{O}(q)$
Super Serial [38]	$q \cdot E$	$\mathcal{O}(q/E)$

Table 9.4.: Time-area trade-offs of different Galois-Field multiplication methods for $GF(2^q)$, from [37].

One can see that a high-speed implementation can compute one multiplication in a single clock, although, at the costs of a squared number of gates. Hence, an implementation which makes use of the single-digit serial or parallel multiplier will be useful. Moreover, POET with four-round AES allows hardware implementations to share components for cipher and hashing, reducing the required area.

Chapter 10

Design Rationale

Key Generation. The key generation follows the idea from [30]. Thus, the user supplies a k-bit secret key SK. The further keys are then generated by encrypting distinct constants using the AES under SK. Since the AES is a well-studied and secure block cipher (a secure PRP), we can ensure to obtain pairwise independent keys, which is crucial for our security analysis (cf. Chapter 8). Note that, when using multiplications in $GF(2^{128})$, one has to consider the risk of weak keys, which can occur with a probability of 2^{-96} [49]. Therefore, one can check for weak keys during the key generation and to re-generate the corresponding key using a new distinct constant, until none of the keys is weak.

Three-Layered Structure. The basic idea behind this symmetric design is given by the secure XEX approach [45], where a block cipher is wrapped by XORing secret values. For POET/POET-m, these values are given by X_{i-1} (top row) and Y_{i-1} (bottom row), which are given as output of F_t and F_b, i.e., two instances of the ϵ-AXU hash function family F using independent keys. Thus, we have shown that POET inherits the resistance against chosen-plaintext- and chosen-ciphertext-adversaries. Moreover, based on the chaining, we show that POET satisfies OPERM-CCA (cf. Chapter 8), which on the other hand implies decryption misuse.

Header Processing. Since POET – beyond other things – is designed to provide a high performance (without neglecting any security properties), we decided to use the fast and provably secure PMAC design [10] for processing the header.

Tag Splitting. Since two of the application fields of POET are low-latency and restricted environments, we want to keep the overhead as small as possible. Therefore, POET inherits the provably secure tag-splitting approach from McOE [22], which provides length-preserving encryption/decryption.

Standard Primitives. Our recommended instances of POET are the AES as block cipher and Galois- Field multiplications for hashing to exploit available native instruction sets of current processors. Since both are well-studied, widely-deployed, standard primitives, POET becomes easy to analyze.

Key Lengths. We recommend POE and POET for the use of 128-bit keys. It is straightforward to enhance POE and POET with a block cipher with longer keys, such as AES-256. However, the benefit of longer keys depends on the use case.

For attacks settings, where an adversary has unlimited oracle access, the security of POET is still upper bounded by the birthday bound of the state size. On the other hand, in settings where the oracle renews the key after $\ell \ll 2^{n/2}$ blocks were processed, POET may benefit from increased key lengths.

Absence of Hidden Weaknesses. We, the designers of POE, POET, and POET-m, declare that we have not hidden any weaknesses in this cipher.

Chapter 11

Acknowledgments

We thank all reviewers of the FSE 2014 for their helpful comments. Furthermore, this work did benefit from the fruitful discussions at the Seminar on Symmetric Cryptography at Schloss Dagstuhl in January 2014, especially from the contributions of Daniel J. Bernstein and Tetsu Iwata. Finally, we would like to thank Jian Guo, Jérémy Jean, Thomas Peyrin, and Lei Wang who pointed out a mismatch between the specified and the analyzed version of POET in the pre-proceedings of the FSE 2014 [27].

Chapter 12

Intellectual Property

To the best of our knowledge, neither POE, POET, POET-m nor any of their instantiations is encumbered by any patents. We have not, and will not, apply for patents on any part of our design or anything in this document, and we are unaware of any other patents or patent filings that cover this work. The example source code is in the public domain and can be freely used. If any of this information changes, the submitters will promptly (and within at most one month) announce these changes on the `crypto-competitions` mailing list.

We make this submission to the CAESAR hash function competition solely as individuals. Our respective employers neither endorse nor condemn this submission.

Chapter 13

Consent

The submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

Bibliography

[1] D. Eastlake 3rd. Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 4305 (Proposed Standard), December 2005. Obsoleted by RFC 4835.

[2] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. Parallelizable and Authenticated Online Ciphers. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT (1)*, volume 8269 of *Lecture Notes in Computer Science*, pages 424–443. Springer, 2013.

[3] Mihir Bellare, Alexandra Boldyreva, Lars R. Knudsen, and Chanathip Namprempre. Online Ciphers and the Hash-CBC Construction. In *CRYPTO*, pages 292–309, 2001.

[4] Mihir Bellare, Alexandra Boldyreva, Lars R. Knudsen, and Chanathip Namprempre. On-line Ciphers and the Hash-CBC Constructions. *J. Cryptology*, 25(4):640–679, 2012.

[5] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In Tatsuaki Okamoto, editor, *ASIACRYPT*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000.

[6] Mihir Bellare and Phillip Rogaway. Encode-Then-Encipher Encryption: How to Exploit Nonces or Redundancy in Plaintexts for Efficient Cryptography. In *ASIACRYPT*, pages 317–330, 2000.

[7] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX Mode of Operation. In *FSE*, pages 389–407, 2004.

[8] Daniel J. Bernstein and Peter Schwabe. New AES Software Speed Records. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *INDOCRYPT*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.

[9] Guido Bertoni, Joan Daemen, Michael 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*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer, 2011.

[10] John Black and Phillip Rogaway. A Block-Cipher Mode of Operation for Parallelizable Message Authentication. In *EUROCRYPT*, pages 384–397, 2002.

[11] Martin Boesgaard, Thomas Christensen, and Erik Zenner. Badger - A Fast and Provably Secure MAC. In John Ioannidis, Angelos D. Keromytis, and Moti Yung,

editors, *ACNS*, volume 3531 of *Lecture Notes in Computer Science*, pages 176–191, 2005.

[12] Andrey Bogdanov, Martin M. Lauridsen, and Elmar Tischhauser. AES-Based Authenticated Encryption Modes in Parallel High-Performance Software. Cryptology ePrint Archive, Report 2014/186, 2014. `http://eprint.iacr.org/`.

[13] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In *MOBICOM*, pages 180–189, 2001.

[14] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster Multiplication in GF(2)[x]. In Alfred J. van der Poorten and Andreas Stein, editors, *ANTS*, volume 5011 of *Lecture Notes in Computer Science*, pages 153–166. Springer, 2008.

[15] Larry Carter and Mark N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[16] Joan Daemen, Mario Lamberger, Norbert Pramstaller, Vincent Rijmen, and Frederik Vercauteren. Computational Aspects of the Expected Differential Probability of 4-Round AES and AES-Like Ciphers. *Computing*, 85(1-2):85–104, 2009.

[17] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.

[18] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176.

[19] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[20] Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *CHES*, pages 1–17, 2009.

[21] Ewan Fleischmann, Christian Forler, and Stefan Lucks. McOE: A Family of Almost Foolproof On-Line Authenticated Encryption Schemes. In *FSE*, pages 196–215, 2012.

[22] Ewan Fleischmann, Christian Forler, Stefan Lucks, and Jakob Wenzel. McOE: A Foolproof On-Line Authenticated Encryption Scheme. Cryptology ePrint Archive, Report 2011/644, 2011. `http://eprint.iacr.org/`.

[23] Pierre-Alain Fouque, Antoine Joux, Gwenaëlle Martinet, and Frédéric Valette. Authenticated On-Line Encryption. In *Selected Areas in Cryptography*, pages 145–159, 2003.

[24] Shay Gueron. Intel® Advanced Encryption Standard (AES) Instructions Set - Rev 3.01. Intel White Paper. Technical report, Intel corporation, September 2012.

[25] Shay Gueron. AES-GCM Software Performance on the Current High End CPUs as a Performance Baseline for CAESAR Competition. In *Directions in Authenticated Ciphers - DIAC 2013*, 2013.

[26] Shay Gueron and Michael E. Kounavis. Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode - Rev 2.01. Intel White Paper. Technical report, Intel corporation, September 2012.

[27] Jian Guo, Jérémy Jean, Thomas Peyrin, and Wang Lei. Breaking POET Authentication with a Single Query. Cryptology ePrint Archive, Report 2014/197, 2014. `http://eprint.iacr.org/`.

[28] George Hotz. Console Hacking 2010 - PS3 Epic Fail. 27th Chaos Communications Congress, 2010. `http://tinyurl.com/39yds8h`.

[29] ITU-T. Interfaces for the Optical Transport Network (OTN). Recommendation G.709/Y.1331, International Telecommunication Union, Geneva, December 2009.

[30] Tetsu Iwata and Kaoru Kurosawa. OMAC: One-Key CBC MAC. In Thomas Johansson, editor, *FSE*, volume 2887 of *Lecture Notes in Computer Science*, pages 129–153. Springer, 2003.

[31] Krzysztof Jankowski and Pierre Laurent. Packed AES-GCM Algorithm Suitable for AES/PCLMULQDQ Instructions. *IEEE Trans. Computers*, 60(1):135–138, 2011.

[32] Jonathan Katz and Moti Yung. Unforgeable Encryption and Chosen Ciphertext Secure Modes of Operation. In *FSE*, pages 284–299, 2000.

[33] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), December 2005.

[34] Tadayoshi Kohno. Attacking and Repairing the WinZip Encryption Scheme. In *ACM Conference on Computer and Communications Security*, pages 72–81, 2004.

[35] Tadayoshi Kohno, John Viega, and Doug Whiting. The CWC-AES Dual-use Mode. Internet Draft. Technical report, 05 2003. Work in progress. Available at `http://tools.ietf.org/html/draft-irtf-cfrg-cwc-00`.

[36] Ted Krovetz and Phillip Rogaway. The Software Performance of Authenticated-Encryption Modes. In *FSE*, pages 306–327, 2011.

[37] David McGrew and John Viega. The Galois/Counter Mode of Operation (GCM). *Submission to NIST. http://csrc.nist.gov/CryptoToolkit/modes/-proposedmodes/gcm/gcm-spec.pdf*, 2004.

[38] Gerardo Orlando and Christof Paar. A Super-Serial Galois Fields Multiplier for FPGAs and its Application to Public-Key Algorithms. In *FCCM*, pages 232–239. IEEE Computer Society, 1999.

[39] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast Software AES Encryption. In Seokhie Hong and Tetsu Iwata, editors, *FSE*, volume 6147 of *Lecture Notes in Computer Science*, pages 75–93. Springer, 2010.

[40] Christof Paar. Implementation Options for Finite Field Arithmetic for Elliptic Curve Cryptosystems. In *Invited presentation at the 3rd Workshop on Elliptic Curve Cryptography (ECC'99)*, 1999.

[41] J. Postel. User Datagram Protocol. RFC 768 (INTERNET STANDARD), August 1980.

[42] J. Postel. Internet Protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864.

[43] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[44] Phillip Rogaway. Authenticated-Encryption with Associated-Data. In *ACM Conference on Computer and Communications Security*, pages 98–107, 2002.

[45] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of*

Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[46] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: a block-cipher mode of operation for efficient authenticated encryption. In *ACM Conference on Computer and Communications Security*, pages 196–205, 2001.

[47] Phillip Rogaway and Thomas Shrimpton. Deterministic Authenticated-Encryption: A Provable-Security Treatment of the Key-Wrap Problem. Cryptology ePrint Archive, Report 2006/221. (Full Version), 2006. `http://eprint.iacr.org/`.

[48] Phillip Rogaway and Haibin Zhang. Online Ciphers from Tweakable Blockciphers. In Aggelos Kiayias, editor, *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, volume 6558 of *Lecture Notes in Computer Science*, pages 237–249. Springer, 2011.

[49] Markku-Juhani Olavi Saarinen. Cycling Attacks on GCM, GHASH and Other Polynomial MACs and Hashes. In Anne Canteaut, editor, *FSE*, volume 7549 of *Lecture Notes in Computer Science*, pages 216–225. Springer, 2012.

[50] Todd Sabin. Vulnerability in Windows NT's SYSKEY encryption. *BindView Security Advisory*, 1999. Available at `http://marc.info/?l=ntbugtraq&m=94537191024690&w=4`.

[51] Leilei Song and Keshab K. Parhi. Efficient Finite Field Serial/Parallel Multiplication. In *ASAP*, pages 72–. IEEE Computer Society, 1996.

[52] Douglas R. Stinson. Universal Hashing and Authentication Codes. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 74–85. Springer, 1991.

[53] Douglas R. Stinson. Universal Hashing and Authentication Codes. *Des. Codes Cryptography*, 4(4):369–380, 1994.

[54] Stefan Tillich and Christoph Herbst. Boosting AES Performance on a Tiny Processor Core. In Tal Malkin, editor, *CT-RSA*, volume 4964 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2008.

[55] Mark N. Wegman and J. Lawrence Carter. New Hash Functions and Their Use in Authentication and Set Equality. *JCSSBM*, 22(3):265–279, June 1981.

Appendix A

Test Vectors for POET

A.1. Galois-Field Multiplication

SK:	01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10	(16 octets)
K:	db f1 84 11 2e b9 11 16 59 71 2b af cf f2 ab 24	(16 octets)
L:	9a 7a 06 19 aa c2 9e 6c 1f 2b 5c 47 53 d5 88 f3	(16 octets)
L_F^{top}:	14 2c 51 c9 af 2c f1 d9 2e 89 37 c4 fb c1 8d 7a	(16 octets)
L_F^{bot}:	b2 12 da 69 51 71 7f 95 59 07 ca d6 bc fe 08 6a	(16 octets)
L_T:	b8 84 cc fd 8f b1 5e 99 15 db 6a d7 6b 29 6e 4d	(16 octets)

H:		(0 octets)
τ:	68 5b 56 f2 eb e0 25 fb b1 a6 09 40 9a af d7 69	(16 octets)
M:	00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff	(16 octets)
C:	f4 b0 19 7a 24 7f f0 7d 67 b0 48 fe 89 eb 29 08	(16 octets)
T:	f1 fd 30 54 5a bf f3 32 50 3e 57 00 58 40 e2 a8	(16 octets)

H:	00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff	(24 octets)
	de ad be ef de af ba be	
τ:	25 89 8a 27 ef f0 fc 67 3a ab 06 87 45 0a 18 f4	(16 octets)
M:		(0 octets)
C:		(0 octets)
T:	2c dd 53 c3 a9 e7 40 f9 53 bb 1e fa 95 c3 42 1b	(16 octets)

SK: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff (16 octets)
K: fd e4 fb ae 4a 09 e0 20 ef f7 22 96 9f 83 83 2b (16 octets)
L: 84 d4 c9 c0 8b 4f 48 28 61 e3 a9 c6 c3 5b c4 d9 (16 octets)
L_F^{top}: 1d f9 27 37 45 13 bf d4 9f 43 6b d7 3f 32 52 85 (16 octets)
L_F^{bot}: da ef 4f f7 e1 3d 46 a6 db cb 1c 02 4e 72 53 87 (16 octets)
L_T: 06 e8 65 b7 a0 36 2d 2f c1 a1 56 3b c2 e3 05 84 (16 octets)

H: 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 (16 octets)
τ: df 4f a1 ff e9 c7 50 93 06 b6 69 cb b2 3f f4 bc (16 octets)

M: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f (56 octets)
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
de ad be ef de af ba be
C: 24 8b 2f d2 e7 1a 8e 9d 45 b4 da ac 1e e7 ea e2 (56 octets)
af 3e 53 6e 7e 4b 16 0f dc 5d a9 1d f7 05 05 bc
9b 96 3a 84 8a 03 50 38 83 2d 86 f4 15 37 52 82
8e c1 7b 11 6f 11 9b f8
T: b7 5f b8 31 bb 86 d7 cf 99 51 42 ec 58 95 10 60 (16 octets)

H: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff (24 octets)
de ad be ef de af ba be
τ: 89 fc f9 14 0d 5f aa b2 a5 36 ff b2 9f ed 5c 92 (16 octets)

M: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f (52 octets)
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
fe fe ba be
C: ee 8d 9b 61 e8 58 ab 75 dc e1 02 ad a5 bb a9 55 (52 octets)
f9 c7 c3 17 0d 11 29 e6 a7 1b 58 56 0a eb 2c eb
0c 71 9b 25 e4 37 b8 56 6b e5 ea fe f1 42 16 fe
1f 82 d3 8f
T: 3f 49 c1 3c 48 d3 6e 9a c4 dd 0f 12 2c b2 27 c7 (16 octets)

A.2. Four-Round AES

SK:	01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10	(16 octets)
K:	db f1 84 11 2e b9 11 16 59 71 2b af cf f2 ab 24	(16 octets)
L:	9a 7a 06 19 aa c2 9e 6c 1f 2b 5c 47 53 d5 88 f3	(16 octets)
L_F^{top}:	14 2c 51 c9 af 2c f1 d9 2e 89 37 c4 fb c1 8d 7a	(16 octets)
L_F^{bot}:	b2 12 da 69 51 71 7f 95 59 07 ca d6 bc fe 08 6a	(16 octets)
L_T:	b8 84 cc fd 8f b1 5e 99 15 db 6a d7 6b 29 6e 4d	(16 octets)

H:		(0 octets)
τ:	68 5b 56 f2 eb e0 25 fb b1 a6 09 40 9a af d7 69	(16 octets)
M:	00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff	(16 octets)
C:	0e ce b2 9d be 27 e7 9a 3a 99 b6 46 76 ef e5 db	(16 octets)
T:	c2 77 f1 6f 1d a8 07 d9 52 55 04 93 d3 a0 ca 84	(16 octets)

H:	00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff	(24 octets)
	de ad be ef de af ba be	
τ:	25 89 8a 27 ef f0 fc 67 3a ab 06 87 45 0a 18 f4	(16 octets)
M:		(0 octets)
C:		(0 octets)
T:	25 89 8a 27 ef f0 fc 67 3a ab 06 87 45 0a 18 f4	(16 octets)

SK: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff (16 octets)
K: fd e4 fb ae 4a 09 e0 20 ef f7 22 96 9f 83 83 2b (16 octets)
L: 84 d4 c9 c0 8b 4f 48 28 61 e3 a9 c6 c3 5b c4 d9 (16 octets)
L_F^{top}: 1d f9 27 37 45 13 bf d4 9f 43 6b d7 3f 32 52 85 (16 octets)
L_F^{bot}: da ef 4f f7 e1 3d 46 a6 db cb 1c 02 4e 72 53 87 (16 octets)
L_T: 06 e8 65 b7 a0 36 2d 2f c1 a1 56 3b c2 e3 05 84 (16 octets)

H: 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 (16 octets)
τ: df 4f a1 ff e9 c7 50 93 06 b6 69 cb b2 3f f4 bc (16 octets)

M: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f (56 octets)
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
de ad be ef de af ba be
C: 4b 95 70 55 1f de fe 5c b5 cd ae a1 d4 cb c5 91 (56 octets)
1d e6 9a a3 32 f1 0f 7d 6f 1d 2d ae 76 1a c4 ad
d7 3b 5e 83 35 38 01 90 bf d7 c6 41 76 4b 52 e7
e3 d8 58 79 e7 07 f7 16
T: 24 17 f6 b8 e0 13 7e ef 46 9a ea aa 8d c3 e4 a4 (16 octets)

H: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff (24 octets)
de ad be ef de af ba be
τ: 89 fc f9 14 0d 5f aa b2 a5 36 ff b2 9f ed 5c 92 (16 octets)

M: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f (52 octets)
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
fe fe ba be
C: ee 8d 9b 61 e8 58 ab 75 dc e1 02 ad a5 bb a9 55 (52 octets)
f9 c7 c3 17 0d 11 29 e6 a7 1b 58 56 0a eb 2c eb
0c 71 9b 25 e4 37 b8 56 6b e5 ea fe f1 42 16 fe
1f 82 d3 8f
T: 3f 49 c1 3c 48 d3 6e 9a c4 dd 0f 12 2c b2 27 c7 (16 octets)

A.3. Full-Round AES

SK:	01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10	(16 octets)
K:	db f1 84 11 2e b9 11 16 59 71 2b af cf f2 ab 24	(16 octets)
L:	9a 7a 06 19 aa c2 9e 6c 1f 2b 5c 47 53 d5 88 f3	(16 octets)
L_F^{top}:	14 2c 51 c9 af 2c f1 d9 2e 89 37 c4 fb c1 8d 7a	(16 octets)
L_F^{bot}:	b2 12 da 69 51 71 7f 95 59 07 ca d6 bc fe 08 6a	(16 octets)
L_T:	b8 84 cc fd 8f b1 5e 99 15 db 6a d7 6b 29 6e 4d	(16 octets)

H:		(0 octets)
τ:	68 5b 56 f2 eb e0 25 fb b1 a6 09 40 9a af d7 69	(16 octets)
M:	00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff	(16 octets)
C:	04 f6 a0 60 59 a5 b4 d5 ce ea 84 bc 0a b1 42 c5	(16 octets)
T:	02 88 30 4b a1 14 78 17 12 eb 79 df 56 0e 95 14	(16 octets)

H:	00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff	(24 octets)
	de ad be ef de af ba be	
τ:	25 89 8a 27 ef f0 fc 67 3a ab 06 87 45 0a 18 f4	(16 octets)
M:		(0 octets)
C:		(0 octets)
T:	62 50 b8 fa ed 20 99 46 58 f8 bc 1e 24 35 3f 20	(16 octets)

SK: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff (16 octets)
K: fd e4 fb ae 4a 09 e0 20 ef f7 22 96 9f 83 83 2b (16 octets)
L: 84 d4 c9 c0 8b 4f 48 28 61 e3 a9 c6 c3 5b c4 d9 (16 octets)
L_F^{top}: 1d f9 27 37 45 13 bf d4 9f 43 6b d7 3f 32 52 85 (16 octets)
L_F^{bot}: da ef 4f f7 e1 3d 46 a6 db cb 1c 02 4e 72 53 87 (16 octets)
L_T: 06 e8 65 b7 a0 36 2d 2f c1 a1 56 3b c2 e3 05 84 (16 octets)

H: 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 (16 octets)
τ: df 4f a1 ff e9 c7 50 93 06 b6 69 cb b2 3f f4 bc (16 octets)

M: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f (56 octets)
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
de ad be ef de af ba be
C: f1 de 86 3e e9 2c 8b e7 e4 97 cc 87 1e 17 07 12 (56 octets)
79 98 da a8 59 65 5f a4 a2 82 3b 44 d6 a9 b8 59
74 90 21 cc 38 35 ee 7e a9 ab 4f 67 8b ca 4c 5d
0d 10 38 51 88 64 6a 6b
T: 84 e9 af a7 e9 0b 26 dc b7 55 46 92 83 d6 04 60 (16 octets)

H: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff (24 octets)
de ad be ef de af ba be
τ: 89 fc f9 14 0d 5f aa b2 a5 36 ff b2 9f ed 5c 92 (16 octets)

M: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f (52 octets)
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
fe fe ba be
C: 11 23 cf 29 91 fd ca be 49 01 ef 80 99 88 26 ea (52 octets)
1b 28 4a de d9 d7 2e 69 6d 22 84 1d 47 0e 66 aa
91 8a ac 03 c6 b8 a4 bf e8 21 00 10 f2 1c 92 a3
78 17 40 ec
T: e6 ad ec ce 21 a1 89 62 4d 73 f4 b0 9d 2f 0c 3f (16 octets)

Appendix B

Proof of the OPERM-CCA-Security of POE

B.1. Upper Bound for COLL^{enc}

Lemma B.1 (COLL^{enc}). *Let M_i, M'_j denote the i-th and j-th block of one or two encryption queries $M, M' \in \mathcal{Q}$, and X_i, X'_j the internal top-row chaining values as defined in Algorithm 6.5. Let $COLL^{enc}$ be the event that $X_i = X'_j$ for two distinct tuples (X_{i-1}, M_i) and (X'_{j-1}, M'_j), with $i, j \geq 1$. Then, the probability of $COLL^{enc}$ is upper bounded by*

$$\Pr[COLL^{enc}] \leq \frac{\ell^2}{2} \cdot \epsilon.$$

Proof. The adversary has full control over the message blocks M_i and M'_j, which can refer to blocks in two messages as well as to different blocks in the same message. In encryption direction, the adversary never sees the values X_i and X'_j that serve as input to the encryption layer.

Case (1): $X_{i-1} = X'_{j-1}$. This case can happen when M and M' share a common prefix up to the $(i-1)$-th message block; otherwise, $\mathcal{A}$ would have already found a collision at this point and we would have given the adversary the attack before. Hence, the advantage for $\mathcal{A}$ is 0 in the latter case. In the former case, from $(X_{i-1}, M_i) \neq (X'_{j-1}, M'_j)$ must follow that $M_i \neq M'_j$. Since X_i and X'_j are computed by $F_t(X_{i-1}) \oplus M_i$ and $F_t(X'_{j-1}) \oplus M'_j$, it must hold for a collision that

$$F_t(X_{i-1}) \oplus M_i = F_t(X'_{j-1}) \oplus M'_j,$$

with $F_t(X_{i-1}) = F_t(X'_{j-1})$. It is trivial that this condition can never hold, and the advantage for $\mathcal{A}$ is 0 in this case.

Case (2): $X_{i-1} \neq X'_{j-1}$**.** Since $F_t(\cdot)$ is an ϵ-AXU family of hash functions, we can derive a family of hash functions $F'_t(\cdot,\cdot)$ as

$$F'_t(x,m) := F_t(x) \oplus m,$$

which is ϵ-AU according to Theorem 4.3. For a collision of the form $X_i = X'_j$, it must hold that

$$F'_t(X_{i-1}, M_i) = F'_t(X'_{j-1}, M'_j).$$

for distinct inputs (X_{i-1}, M_i) and (X'_{j-1}, M'_j). Hence, the probability of $\mathrm{COLL}^{\mathrm{enc}}$ to happen can be upper bounded by

$$\Pr[\mathrm{COLL}^{\mathrm{enc}}] = \frac{\ell(\ell-1)}{2} \cdot \epsilon \leq \frac{\ell^2}{2}\epsilon.$$

□

B.2. Upper Bound for NOCOLLWIN

Lemma B.2 (NOCOLLWIN). *Let NOCOLLWIN be the event as defined in Equation* (8.3) *in the OPERM-CCA proof of POE. Then, the probability of NOCOLLWIN can be upper bounded by*

$$\Pr[\mathit{NOCOLLWIN}] \leq \frac{\ell^2}{2^n - \ell}.$$

Proof. Here, we regard the case that $\mathcal{A}$ shall distinguish POE $(\mathsf{POE}_\pi, \mathsf{POE}^{-1}_{\pi^{-1}})$ from a random OPERM when no internal collisions occur. Prior, we can generalize that each distinct query pair (M,C), $(M',C') \in \mathcal{Q}$ shares a common prefix. We denote the by i the length of their longest common prefix:

$$i = \mathsf{LLCP}_n(M,M') = \max_h \left\{ \forall j \in 0,\ldots,h : \; M_j = M'_j \right\}.$$

In the following, we study the difference in the behavior of POE and P for three subcases, and derive the advantage of $\mathcal{A}$ to distinguish between POE and P for each of them.

Case (2.1): Message Blocks in the Common Prefix. The input and output behaviors of $(\mathsf{POE}_\pi, \mathsf{POE}^{-1}_{\pi^{-1}})$ and a random OPERM are identical for the common prefix. Hence, the advantage for $\mathcal{A}$ in this case is 0.

Case (2.2): Message Block directly after the Common Prefix. Since M and M' share an i-block longest common prefix, it must hold that $X_i = X'_i$ and $Y_i = Y'_i$. For an encryption query with the inputs $M_{i+1} \neq M'_{i+1}$, it must hold that

$$X_{i+1} = F_t(X_i) \oplus M_{i+1} \neq F_t(X'_i) \oplus M'_{i+1} = X'_{i+1}.$$

Since $\pi(\cdot)$ is an SPRP, it must follow that

$$C_{i+1} \;=\; F_b(Y_i) \oplus \pi(X_{i+1}) \;\neq\; F_b(Y_i') \oplus \pi(X_{i+1}') \;=\; C_{i+1}'.$$

The analysis is similar in decryption direction. In the random case, P or P^{-1} are used with two different prefixes $(M_1 \;||\; \ldots \;||\; M_{i+1})$ and $(M_1' \;||\; \ldots \;||\; M_{i+1}')$ in encryption, or $(C_1 \;||\; \ldots \;||\; C_{i+1})$ and $(C_1' \;||\; \ldots \;||\; C_{i+1}')$ in decryption direction. Since P and P^{-1} are random OPERMs, $C_{i+1} \neq C_{i+1}'$ or $M_{i+1} \neq M_{i+1}'$ also must hold in this case, respectively. Hence, the behavior of $(\mathsf{POE}_\pi, \mathsf{POE}_{\pi^{-1}}^{-1})$ and a random OPERM is also identical for the message block after the common prefix, and the advantage for $\mathcal{A}$ to distinguish them is also 0 in this case.

Case 2.3: After the $(i+1)$-st Message Block. In the random case, each query output is chosen uniformly at random from the set $\{0,1\}^n$. However, in the real world, each output of either an encryption or a decryption query is chosen uniformly at random from the set $\{0,1\}^n \setminus \mathcal{Q}$. This means that in the real case, POE loses randomness with every query. Therefore, we can upper bound the probability of $\mathcal{A}$ to distinguish POE from a random OPERM by

$$\frac{\ell^2}{2^n - \ell}.$$

Our claim follows from summing up the individual terms. □

B.3. Upper Bound for COLL$^{\mathsf{lmb}}$

Lemma B.3 (COLL$^{\mathsf{lmb}}$). *Let M_i, M_j' denote the i-th and j-th block of one or two encryption queries $M, M' \in \mathcal{Q}$, and X_i, X_j' the internal top-row chaining values as defined in Algorithm 6.5. Further, let ℓ_M denote the number of blocks in M and $\ell_{M'}$ the number of blocks in M'. Let $COLL^{lmb}$ be the event that $X_i = X_j'$ for two distinct tuples (X_{i-1}, M_i) and (X_{j-1}', M_j'), with $i, j \geq 1$ and $i = \ell_M$. Then, the probability of $COLL^{lmb}$ is upper bounded by*

$$\Pr\left[COLL^{lmb}\right] \;\leq\; \max\left\{\ell \cdot q \cdot \epsilon, \frac{q^2}{2^n - q}\right\}.$$

Proof. The encryption and decryption of POET differs only from that of POE in the way how POET treats the last message block. To prove OPERM-CCA security for POET, we have to consider separately the probability for collisions of internal chaining values when M_i and/or M_j' refer to the last blocks of M and M', respectively. Here, we want to upper bound the probability of collision in the chaining values $X_i = X_j'$ for two distinct tuples (X_{i-1}, M_i) and (X_{j-1}', M_j'), with $i, j \geq 1$. Wlog., we say in the following that M_i is the last block of M, i.e., $i = \ell_M$.

POET pads a message whose length is not a multiple of n with the most significant bits of the intermediate tag τ: $M_{\ell_M} \;||\; \tau^\alpha$. Hence, we have to analyze the options that M_{ℓ_M} is a full block or M_{ℓ_M} is padded. For each of them, we have to analyze three mutually exclusive cases, depending on M_j':

1. M'_j is an intermediate block of M': $j < \ell_{M'}$.
2. M'_j is the last, unpadded block of M': $j = \ell_{M'}$ and $|M'_j| = n$.
3. M'_j is the last, padded block of M': $j = \ell_{M'}$ and $|M'_j| < n$.

Remark. Note that in each case, we regard the subcases $X_{i-1} = X'_{j-1}$ and $X_{i-1} \neq X'_{j-1}$, where $X_{i-1} = X'_{j-1}$ can result from a longest common prefix of M and M', or from a previous collision that $\mathcal{A}$ found before. Since the advantage for $\mathcal{A}$ is 0 when it found a collision before, we regard only the case when $X_{i-1} = X'_{j-1}$ results from a common prefix. Note that $(X_{i-1}, M_i) \neq (X'_{j-1}, M'_j)$ implies that $M_i \neq M'_j$; otherwise, M_i and M'_j would just extend the common prefix and the advantage for $\mathcal{A}$ would be 0 again.

Case (1): Without Tag-Splitting at M_{ℓ_M}. Here, the top-row chaining value X_{ℓ_M} is computed by $F_t(X_{\ell_M-1}) \oplus M_{\ell_M} \oplus S$. Depending on the above mentioned cases, X'_j is computed slightly differently. For a collision of the form $X_{\ell_M} = X'_j$, it must hold

$$F_t(X_{\ell_M-1}) \oplus M_{\ell_M} \oplus S = \begin{cases} F_t(X'_{j-1}) \oplus M'_j & \text{if } j < \ell_{M'}, \\ F_t(X'_{\ell_{M'}-1}) \oplus M'_{\ell_{M'}} \oplus S' & \text{if } j = \ell_{M'}, |M'_{\ell_{M'}}| = n, \\ F_t(X'_{\ell_{M'}-1}) \oplus (M'_{\ell_{M'}} \,||\, \tau'^{\alpha}) \oplus S' & \text{if } j = \ell_{M'}, |M'_{\ell_{M'}}| < n. \end{cases}$$

Note that these cover all possible cases for a collision in X_{ℓ_M} and X'_j when M_{ℓ_M} is not padded. Furthermore, for the sake of simplicity, we make $\mathcal{A}$ stronger than it is and give it full control over τ'^{α}.

Subcase (1.1): $X_{\ell_M-1} = X'_{j-1}$. Since $F_t(\cdot)$ is a permutation, it must hold $F_t(X_{i-1}) = F_t(X'_{\ell_{M'}-1})$. Hence, we can rearrange our equations from above and see that a collision in X_{ℓ_M} and X'_j requires that $\mathcal{A}$ must find

$$\begin{aligned} M_{\ell_M} \oplus M'_j &= S \text{ or} \\ M_{\ell_M} \oplus M'_{\ell_{M'}} &= S \oplus S' \text{ or} \\ M_{\ell_M} \oplus (M'_{\ell_{M'}} \,||\, \tau'^{\alpha}) &= S \oplus S'. \end{aligned}$$

In all cases, $\mathcal{A}$ has to choose M_{ℓ_M} and M'_j appropriately to match S or $S \oplus S'$. Furthermore, recall that this case implies $M_{\ell_M} \neq M'_j$, which rules out the trivial way of finding a collision for $S = S'$. Since S and S' are secret, the success probability for $\mathcal{A}$ in either case is at most $\frac{1}{2^n - q}$.

Subcase (1.2): $X_{\ell_M-1} \neq X'_{j-1}$. In this case it must hold that $F_t(X_{\ell_M-1}) \neq F_t(X'_{j-1})$. This time, we can rearrange our equations from above and see that a collision in X_{ℓ_M} and X'_j requires that $\mathcal{A}$ must find

$$\begin{aligned} F_t(X_{\ell_M-1}) \oplus F_t(X'_{j-1}) &= M_{\ell_M} \oplus M'_j \oplus S \text{ or} \\ F_t(X_{\ell_M-1}) \oplus F_t(X'_{\ell_{M'}-1}) &= M_{\ell_M} \oplus M'_{\ell_{M'}} \oplus S \oplus S' \text{ or} \\ F_t(X_{\ell_M-1}) \oplus F_t(X'_{\ell_{M'}-1}) &= M_{\ell_M} \oplus (M'_{\ell_{M'}} \,||\, \tau'^{\alpha}) \oplus S \oplus S'. \end{aligned}$$

Since $F_t(\cdot)$ is an ϵ-AXU family of hash functions, the success probability that $\mathcal{A}$ can choose M_{ℓ_M} and M'_j appropriately can be upper bounded by ϵ in either case.
Thus, we can upper bound the success probability of $\mathcal{A}$, asking at most q queries of a total length of ℓ blocks, for Case (1) by

$$\max\left\{\ell \cdot q \cdot \epsilon, \frac{q^2}{2^n - q}\right\}.$$

Case (2): With Tag-Splitting at M_{ℓ_M}. Now, the top-row chaining value X_{ℓ_M} is computed by $F_t(X_{\ell_M-1}) \oplus (M_{\ell_M} \mid\mid \tau^\alpha) \oplus S$. For a collision of the form $X_{\ell_M} = X'_j$, it must hold that

$$X_{\ell_M} = \begin{cases} F_t(X'_{j-1}) \oplus M'_j & \text{if } j < \ell_{M'}, \\ F_t(X'_{\ell_{M'}-1}) \oplus M'_{\ell_{M'}} \oplus S' & \text{if } j = \ell_{M'}, |M'_{\ell_{M'}}| = n, \\ F_t(X'_{\ell_{M'}-1}) \oplus (M'_{\ell_{M'}} \mid\mid \tau'^\alpha) \oplus S' & \text{if } j = \ell_{M'}, |M'_{\ell_{M'}}| < n. \end{cases}$$

Again, note that these cover all possible cases for a collision in X_{ℓ_M} and X'_j when M_{ℓ_M} is padded. For the sake of simplicity, we make $\mathcal{A}$ stronger than it is and give it full control over τ^α and τ'^α.

Subcase (2.1): $X_{\ell_M-1} = X'_{j-1}$. Since $F_t(\cdot)$ is a permutation, it must hold $F_t(X_{i-1}) = F_t(X'_{\ell_{M'}-1})$. Hence, we can rearrange our equations from above and see that a collision in X_{ℓ_M} and X'_j requires that $\mathcal{A}$ must find

$$\begin{aligned} (M_{\ell_M} \mid\mid \tau^\alpha) \oplus M'_j &= S \text{ or} \\ (M_{\ell_M} \mid\mid \tau^\alpha) \oplus M'_{\ell_{M'}} &= S \oplus S' \text{ or} \\ (M_{\ell_M} \mid\mid \tau^\alpha) \oplus (M'_{\ell_{M'}} \mid\mid \tau'^\alpha) &= S \oplus S'. \end{aligned}$$

In all cases, $\mathcal{A}$ has to choose M_{ℓ_M} and M'_j appropriately to match S or $S \oplus S'$. Furthermore, recall that this case implies $M_{\ell_M} \neq M'_j$, which rules out the trivial way of finding a collision for $S = S'$. Since S and S' are secret, the success probability for $\mathcal{A}$ in either case is at most $\frac{1}{2^n-q}$.

Subcase (2.2): $X_{\ell_M-1} \neq X'_{j-1}$. In this case, it must hold that $F_t(X_{\ell_M-1}) \neq F_t(X'_{j-1})$. This time, we can rearrange our equations from above and see that a collision in X_{ℓ_M} and X'_j requires that $\mathcal{A}$ must find

$$\begin{aligned} F_t(X_{\ell_M-1}) \oplus F_t(X'_{j-1}) &= (M_{\ell_M} \mid\mid \tau^\alpha) \oplus M'_j \oplus S \text{ or} \\ F_t(X_{\ell_M-1}) \oplus F_t(X'_{\ell_{M'}-1}) &= (M_{\ell_M} \mid\mid \tau^\alpha) \oplus M'_{\ell_{M'}} \oplus S \oplus S' \text{ or} \\ F_t(X_{\ell_M-1}) \oplus F_t(X'_{\ell_{M'}-1}) &= (M_{\ell_M} \mid\mid \tau^\alpha) \oplus (M'_{\ell_{M'}} \mid\mid \tau'^\alpha) \oplus S \oplus S'. \end{aligned}$$

Since $F_t(\cdot)$ is an ϵ-AXU family of hash functions, the success probability that $\mathcal{A}$ can choose M_{ℓ_M} and M'_j appropriately can be upper bounded by ϵ in either case.

Similar to Case (1), we can upper bound the success probability of $\mathcal{A}$, asking at most q queries of a total length of ℓ blocks, for Case (2) by

$$\max\left\{\ell \cdot q \cdot \epsilon, \frac{q^2}{2^n - q}\right\}.$$

Since Case (1) and Case (2) are mutually exclusive, the success probability of $\mathcal{A}$ for the event $\mathrm{COLL}^{\mathrm{lmb}}$ is given by the maximum of the success probabilities of the two cases. Thus, it holds that

$$\Pr\left[\mathrm{COLL}^{\mathrm{lmb}}\right] \leq \max\left\{\ell \cdot q \cdot \epsilon, \frac{q^2}{2^n - q}\right\}.$$

□