An extended abstract of this article was accepted for presentation at ESORICS 2020. This is the full version and available in the IACR eprint archive.

Encrypt-to-self: Securely Outsourcing Storage

Jeroen Pijnenburg[1] and Bertram Poettering[2]

[1] Royal Holloway, University of London, Egham, United Kingdom
jeroen.pijnenburg.2017@rhul.ac.uk

[2] IBM Research – Zurich, Rüschlikon, Switzerland
poe@zurich.ibm.com

Abstract. We put forward a symmetric encryption primitive tailored towards a specific application: outsourced storage. The setting assumes a memory-bounded computing device that inflates the amount of volatile or permanent memory available to it by letting other (untrusted) devices hold encryptions of information that they return on request. For instance, web servers typically hold for each of the client connections they manage a multitude of data, ranging from user preferences to technical information like database credentials. If the amount of data per session is considerable, busy servers sooner or later run out of memory. One admissible solution to this is to let the server *encrypt* the session data *to itself* and to let the client store the ciphertext, with the agreement that the client reproduce the ciphertext in each subsequent request (e.g., via a cookie) so that the session data can be recovered when required.

In this article we develop the cryptographic mechanism that should be used to achieve confidential and authentic data storage in the encrypt-to-self setting, i.e., where encryptor and decryptor coincide and constitute the only entity holding keys. We argue that standard authenticated encryption represents only a suboptimal solution for preserving confidentiality, as much as message authentication codes are suboptimal for preserving authenticity. The crucial observation is that such schemes instantaneously give up on *all* security promises in the moment the key is compromised. In contrast, data protected with our new primitive remains fully integrity protected and unmalleable. In the course of this paper we develop a formal model for encrypt-to-self systems, show that it solves the outsourced storage problem, propose surprisingly efficient provably secure constructions, and report on our implementations.

1 Introduction

We explore techniques that enable a computing device to securely outsource the storage of data. We start with motivating this area of research by describing three application scenarios where outsourcing storage might prove crucial.

Web Server. We come back to the example considered in the abstract, giving more details. While it is difficult to make general statements about the setup of a web server back-end, it is fair to say that the processing of HTTP requests routinely also includes extracting a session identifier from the HTTP header and fetching basic session-related information (e.g., the user's password, the date and time of the last login, the number of failed login attempts, but also other kinds of data not related to security) from a possibly remote SQL database. To avoid the inherent bottleneck induced by the transmission and processing of the database query, such data can be cached on the web server, the limits of this depending only on the amount of available working memory (RAM). For some types of web applications and a large number of web sessions served simultaneously, these memory-imposed limits might represent a serious restriction to efficiency. This article scouts techniques that allow the web server to securely outsource the storage of session information to the (untrusted) web clients.

Hardware Security Module. An HSM is a computing device that performs cryptographic and other security-related operations on behalf of the owning user. While such devices are internally built from

off-the-shelf CPUs and memory chips, a key concept of HSMs is that they are specially encapsulated to protect them against physical attacks, including various kinds of side channel analysis. One consequence of this tamper-proof shielding is that the memory capacity of an HSM can never be physically extended—unlike it would be the case for desktop computers—so that the amount of available working memory might constitute a relevant obstacle when the HSM is deployed in applications with requirements that increase over time (e.g., due to a growing user base). This article scouts techniques that allow the HSM to securely outsource the storage of any kind of valuable information to the (untrusted) embedding host system.

Smartcard. A smartcard, most prominently recognized in the form of a payment card or a mobile phone security token, is effectively a tiny computing device. While fairly potent configurations exist (with 32-bit CPUs and a couple of 100KBs of memory), as the costs associated with producing a smartcard scales roughly linearly with the amount of implemented physical memory, in order to be cost effective, mass-produced cards tend to come with only a small amount of memory. This article scouts techniques that allow smartcards to securely outsource the storage of valuable information to the infrastructure they connect to, e.g., a banking or mobile phone backbone, or a smartphone.

Trusted Platform Module. A TPM is a discrete security chip that is embedded into virtually all laptops and desktop PCs produced in the past decade. A TPM supports its host system by offering trusted cryptographic services and is typically relied upon by boot loaders and operating systems. TPMs are located conceptually between HSMs and smartcards, and as much as these they benefit from a secure option to outsource storage.

Outsourced Storage based on Symmetric Cryptography. If a computing device has access to some kind of external storage facility (a memory chip wired to it, a connected hard drive, cloud storage, etc.), then, intuitively, it can virtually extend the amount of memory available to it by outsourcing storage, i.e., by serializing data objects and communicating them to the storage facility which will reproduce them on request. In this article we focus on the case where neither the external storage facility nor the connection to it is considered trustworthy. More concretely, we assume that all infrastructure outside of the computing device itself is under control of an adversary that aims at reading or changing the data that is to be externally stored.[3] As a first approximation one might conclude that standard tools from the domain of symmetric encryption are sufficient to achieve security in this setting. Consider for instance the following approach based on authenticated encryption (AE, [12]): The computing device samples a fresh symmetric key; whenever it wants to store internal data on the outsourced storage, it encrypts and authenticates the data by invoking the AE encryption algorithm with its key and hands the resulting ciphertext over to the storage facility; to retrieve the data, it requests a copy of the ciphertext, and decrypts and verifies it. While this simple solution requires further tweaking to thwart replay attacks,[4] as long as the AE key remains private it can be used to protect confidentiality and integrity as expected.

Our Contribution: Secure Outsourced Storage w/ Key Leakage. While we confirm that standard cryptographic methods will securely solve the storage outsourcing problem if the used key material remains private, we argue that satisfactory solutions should go a step further by providing as much security as possible even if the latter assumption (that keys remain private) is not met. Indeed, different attacks against practical systems that lead to partial or full memory leakage continue to regularly emerge (including different types of side channel analysis against embedded systems,[5] cold-boot attacks against memory chips,[6] Meltdown/Spectre-like attacks against modern CPUs,[7][8] etc.), and it is commonly understood that the corruption model considered for cryptographic primitives should always be

[3] Certainly, the storage device can always decide to "fail" by not returning any data previously stored into it, leading to an attack on the *availability* of the computing device. We hence consider environments where this is either not a problem or where such an attack cannot be prevented anyway (independently of the storage technique). Note that this assumption holds for our three motivating scenarios.

[4] One option to strengthen the scheme against replay is to implement the AE primitive nonce-based [13], and using a strictly increasing nonce for encrypting and decrypting.

[5] https://en.wikipedia.org/wiki/Side-channel_attack

[6] https://en.wikipedia.org/wiki/Cold_boot_attack

[7] https://en.wikipedia.org/wiki/Meltdown_(security_vulnerability)

[8] https://en.wikipedia.org/wiki/Spectre_(security_vulnerability)

as strong as possible and affordable. For two-party symmetric encryption (e.g., AE) this strongest model necessarily excludes any type of user corruption[9] as the keys of both parties are identical: Once any party is corrupted, any past or future ciphertext can be decrypted and ciphertexts can be forged for any message, i.e., no form of confidentiality or authenticity remains. We point out, however, that for outsourced storage a stronger corruption model is both feasible and preferable. Clearly, like in the AE case, if the adversary obtains a copy of the used key material then all confidentiality guarantees are lost (the adversary can decrypt what the device can decrypt, that is, everything), but a similar reasoning with respect to integrity protection cannot be made. To see this, consider the encrypt-then-hash (EtH) solution where the computing device encrypts the outsourced data as described above, but in addition to having the ciphertext stored externally it internally registers a hash of it (computed with, say, SHA256). When the device decides to recover externally stored data, it requests a copy of the ciphertext, recomputes its hash value, and decrypts only if the hash value is consistent with the internally registered value. Note that even if the device is corrupted and its keys became public, all successfully decrypted ciphertexts are necessarily authentic.

The example just given shows that while no solution for secure storage outsourcing can do much about protecting data confidentiality against key leakage attacks, solutions can *fully* protect the integrity of the stored data in any case. Naive AE-based schemes do not provide this type of security, and the contribution of our work is to fill this gap and to explore corresponding constructions. Precisely, this article provides the following: (1) We identify the new *encrypt-to-self* (ETS) primitive as the right cryptographic tool to solve the outsourced storage problem and formalize its syntax and security properties. (2) We formalize notions more directly related to the outsourced storage problem and provably confirm that secure solutions based on ETS are indeed immediate. (3) We design provably secure constructions of ETS from established cryptographic primitives.[10] (4) We develop open-source implementations of our constructions that are optimized with respect to security and efficiency.

Related Work. While we are not aware of any former systematic treatment of the encrypt-to-self (ETS) primitive, a number of similar primitives or ad hoc constructions partially overlap with our results. We discuss these in the following, but emphasize that none of them provides general solutions to the ETS problem.

Memory Encryption in Modern CPUs. Recent desktop and server CPUs offer dedicated infrastructure for memory encryption,[11] with the main applications in cloud computing and Trusted Execution Environments (TEEs). Prominent TEE examples include Intel SGX[12] and ARM TrustZone[13] in which every memory access of the processes that are executed within a TEE (aka 'enclave') is conducted through a memory encryption engine (MEE). This effectively implements outsourced data storage, but with quite different access rules and patterns than in the ETS case. While we consider the (stateless) encryption of a message to a ciphertext and then a decryption of a ciphertext back to a message, MEEs are stateful systems that consider the protected physical memory area a single ciphertext that is constantly locally modified with each write operation [8].

Password Managers. A password manager can be seen as a database that stores security credentials in an encrypted form and requires e.g., a master password to be unlocked. Also this can be seen as an ETS instance, but the cryptographic design of password managers has a different focus than general outsourced storage. More concretely, the central challenge solved by good password managers is the password-based key derivation,[14] which typically involves invoking a time-expensive derivation function like PBKDF2 [9] or a memory-hard derivation function like ARGON2 [5]. Password-based key derivation is not considered in our treatment of the ETS primitive (we instead assume uniform keys).

Encryptment. A symmetric encryption option that recently emerged as a proposal to protect messages in instant messaging is Encryptment [7]. Its features go beyond regular authenticated encryption in that

[9] We use the terms 'key leakage', 'user corruption', and 'state corruption' synonymously.

[10] The above encrypt-then-hash (EtH) solution is secure in our models but requires two passes over the data. Our solutions are more efficient, getting along with just one pass.

[11] https://software.intel.com/en-us/blogs/2017/12/22/intel-releases-new-technology-specification-for-memory-encryption

[12] https://software.intel.com/en-us/sgx/details

[13] https://genode.org/documentation/articles/trustzone

[14] https://1password.com/files/1Password-White-Paper.pdf

the tags contained in ciphertexts act as (cryptographically strongly binding) commitments to the encoded messages. This committing feature was deemed helpful for the public resolution of cyber harassment cases by allowing affected parties to appeal to a judging authority by opening their ciphertexts by releasing their keys. On first sight this has nothing to do with our ETS setting (in which only one party holds a key, this key would never be deliberately shared, and a necessity of provably releasing message contents to anybody else is not considered). Interestingly, however, our constructions of ETS are very similar to those of [7]. The intuitive reason for this is that the ETS setting requires that ciphertexts remain unforgeable under key leakage, which somewhat aligns with the committing property of encryptment that is required to survive disclosing keys to a judge. Ultimately, however, the applications and thus security models of ETS and encryptment differ, and our constructions are actually more efficient than those in [7].[15]

Technical Approach. In addition to formalizing the security of the encrypt-to-self (ETS) primitive, in the course of this article we also propose efficient provably-secure constructions from standardized building blocks. As discussed above, the authenticity promises of ETS shall withstand adversaries that have knowledge of the key material. In this setting one cannot hope that standard secret-key authentication building blocks like MACs or universal hash functions will be of help, as generically they lose all security when the key is leaked. We instead employ, as they manifest *unkeyed* authentication primitives, cryptographic hash functions like SHA256. A first candidate construction, already hinted at above, would be the encrypt-then-hash (EtH) approach where the message is first encrypted (using any secret key scheme, e.g., AES-CTR) and the ciphertext is then hashed. Our constructions are more efficient than this by exploiting the structure of Merkle–Damgård (MD) hash functions and dual-use leveraging on the properties of their inner building block: the compression function (CF). Intuitively, for authentication we build on the collision resistance of the CF, and for confidentiality we build on a PRF-like property of the CF. More precisely, our message schedule for the CF is such that each invocation provides both confidentiality *and* integrity for the processed block. This effectively halves the computational costs in comparison to the EtH approach.

We believe that a cryptographic analysis is not complete without also implementing the construction under consideration. This is because only implementing a scheme will enforce making conscious decisions about all its details and building blocks, and these decisions may crucially affect the obtained security and efficiency. We thus realized three ready-to-use instances of the ETS primitive, based on the CFs of the top performing hash functions SHA256, SHA512, and BLAKE2. In fact, observations from implementing the schemes led to considerable feedback to the theoretical design which was updated correspondingly. One example for this is connected to memory alignment: Computations on modern CPUs experience noticeable efficiency penalties if memory accesses are not aligned to specific boundaries. Our constructions reflect this at two different levels: at the register level and at the cache level (64 bit alignment for register-oriented operations, and 256 bit alignment for bulk memory transfers[16]).

2 Preliminaries

2.1 Notation

All algorithms considered in this article may be randomized. We let $\mathbb{N} = \{0, 1, \ldots\}$ and $\mathbb{N}^+ = \{1, 2, \ldots\}$. For the Boolean constants True and False we either write `T` and `F`, respectively, or 1 and 0, respectively, depending on the context. An alphabet Σ is any finite set of symbols or characters. We denote with Σ^n the set of strings of length n and with $\Sigma^{\leq n}$ the strings of length up to (and including) n. In the practical parts of this article we assume that $|\Sigma| = 256$, i.e., that all strings are byte strings. We denote string concatenation with $\|$. If *var* is a string variable and *exp* evaluates to a string, we write $var \xleftarrow{\|} exp$ shorthand for $var \leftarrow var \,\|\, exp$. Further, if *exp* evaluates to a string, we write $var \,\|\, var' \leftarrow_n exp$ to denote splitting *exp* such that we assign the first n characters from *exp* to *var* and assign the remainder to var'. When we do not need the remainder, we write $var \leftarrow_n exp$ shorthand for $var \,\|\, dummy \leftarrow_n exp$

[15] This is the case for at least two reasons: (1) The ETS primitive does not need to be committing to the key, which is the case for encryptment. (2) Our message padding is more sophisticated than that of [7] and does not require the processing of a length field.

[16] The value 256 stems from the size of the cache lines of 1st level cache.

and discard *dummy*. In pseudocode, if S is a finite set, expression $\$(S)$ stands for picking an element of S uniformly at random. Associative arrays implement the 'dictionary' data structure: Once the instruction $A[\cdot] \leftarrow exp$ initialized all items of array A to the default value exp, with $A[idx] \leftarrow exp$ and $var \leftarrow A[idx]$ individual items indexed by expression idx can be updated or extracted.

2.2 Security Games

Security games are parameterized by an adversary, and consist of a main game body plus zero or more oracle specifications. The execution of a game starts with the main game body and terminates when a 'Stop with *exp*' instruction is reached, where the value of expression *exp* is taken as the outcome of the game. The adversary can query all oracles specified by the game, in any order and any number of times. If the outcome of a game G is Boolean, we write $\Pr[\mathrm{G}(\mathcal{A})]$ for the probability that an execution of G with adversary $\mathcal{A}$ results in True, where the probability is over the random coins drawn by the game and the adversary. We define macros for specific combinations of game-ending instructions: We write 'Win' for 'Stop with `T`' and 'Lose' for 'Stop with `F`', and further 'Reward *cond*' for 'If *cond*: Win', 'Promise *cond*' for 'If $\neg$*cond*: Win', 'Require *cond*' for 'If $\neg$*cond*: Lose'. These macros emphasize the specific semantics of game termination conditions. For instance, a game may terminate with 'Reward *cond*' in cases where the adversary arranged for a situation—indicated by *cond* resolving to True—that should be awarded a win (e.g., the crafting of a forgery in an authenticity game).

2.3 Handling of Algorithm Failures

Regarding the algorithms of cryptographic schemes, we assume that *any* such algorithm can fail. Here, by failure we mean that an algorithm doesn't generate output according to its syntax specification, but instead outputs some kind of error indicator (e.g., an AE decryption algorithm that rejects an unauthentic ciphertext or a randomized signature algorithm that doesn't have sufficiently many random bits to its disposal). Instead of encoding this explicitly in syntactical constraints which would clutter the notation, we assume that if an algorithm invokes another algorithm as a subroutine, and the latter fails, then also the former immediately fails.[17] We assume the same for game oracles: If an invoked scheme algorithm fails, then the oracle immediately aborts as well. Further, we assume that the adversary learns about this failure, i.e., the oracle will return the error indicator when it aborts. Note that this implies that if a scheme's algorithms leak vital information through error messages, then the scheme will not be secure in our models. (That is, our models are particularly robust.) We believe that our way to handle errors implicitly rather than explicitly contributes to obtaining definitions with clean and clear semantics.

2.4 Memory Alignment

For n a power of 2, we say an address of computer memory is n-byte aligned if it is a multiple of n bytes. We further say that a piece of data is n-byte aligned if the address of its first byte is n-byte aligned. A modern CPU accesses a single (aligned) word in memory at a time. Therefore, the CPU performs reads and writes to memory most efficiently when the data is aligned. For example, on a 64-bit machine, 8 bytes of data can be read or written with a single memory access if the first byte lies on an 8-byte boundary. However, if the data does not lie within one word in memory, the processor would need to access two memory words, which is considerably less efficient. Our scheme algorithms are designed such that when they need to move around data, they exclusively do this for aligned addresses. In practice, the preferred alignment value depends on the hardware used, so for generality in this article we refer to it abstractly as the memory alignment value mav. (A typical value would be mav = 8.)

[17] This approach to handling algorithm failures is taken from [11] and borrows from how modern programming languages handle 'exceptions', where any algorithm can raise (or 'throw') an exception, and if the caller does not explicitly 'catch' it, the caller is terminated as well and the exception is passed on to the next level. See Wikipedia: `Exception_handling_syntax` for exception handling syntaxes of many different programming languages.

2.5 Tweaking the Compression Functions of Hash Functions

The main NIST hash functions of the SHA2 family (FIPS 180-4, [10]) accomplish their task of hashing a message into a short string by strictly following the Merkle–Damgård framework: All inputs to their core building block —the compression function— are either directly taken from the message or from the chaining state. It has been recognized, however, that options to further contextualize or domain-separate the inputs of compression functions can be of advantage. Indeed, compression functions that are designed according to the alternative, more recent HAIFA framework [4] have a number of additional inputs, for instance an explicit salt input, that allow for weaving some extra bits of context information into the bulk hash operations. A concrete example for this is the compression function of the popular BLAKE2 hash function ([2, 14], a HAIFA design), which takes as an additional input a Boolean finalization flag that is to be set specifically when processing the very last (padded) block of a hash computation. The idea behind making the last invocation "special" is that this effectively thwarts length extension attacks: While conducting extension attacks against the SHA2 hash functions, where the compression functions do not natively support any such marking mechanism, is quite immediate,[18] similar attacks against BLAKE2 are impossible [6]. We note that, generally speaking, an ad hoc way of augmenting the input of a compression function by an additional small number of bits is to XOR predefined constants into the hashing state (e.g., before or while the compression function is executed), with the choice of constants depending on the added bits. For instance, if the finalization flag is set, the BLAKE2 compression function will flip all bits of one of its inputs, but beyond that operate as normal.

While textbook SHA2 does not support contextualizing compression function invocations via additional inputs, we observe that NIST, in order to solve an emerging domain-separation problem in the definition of their FIPS 180-4 standard, employed ad hoc modifications of some SHA2 functions that can be seen as (implicitly) retrofitting a one-bit additional input into the compression function. Concretely, the SHA512/t functions [10], that intuitively represent plain SHA512 truncated to $0 < t < 512$ bits, are carefully designed such that for any $t_1 \neq t_2$ the functions SHA512/t_1 and SHA512/t_2 are independent of each other.[19] The separation of the individual SHA512/t versions works as follows [10, Sec. 5.3.6]: First compute the SHA512 hash value of the string `"SHA512/xxx"` (where placeholder `xxx` is replaced by the decimal encoding of t), then XOR the byte value `0xa5` (binary: `0b10100101`) into every byte of the resulting chain state, then continue with regular SHA512 steps from that state on, truncating the final hash value to t bits. While the XORing step is ad hoc, it arguably represents a fairly robust domain separation method for SHA2.

Our constructions of the encrypt-to-self primitive rely on compression functions that are *tweaked* with a single bit, that is, that support one bit as an additional input. When we implement this based on SHA2 compression functions, we employ precisely the mechanism scouted by NIST: When the additional tweak bit is set, we XOR constant `0xa5` into all state bytes and continue operation as normal. Our BLAKE2 based construction, on the other hand, uses the already existing finalization bit.

3 Foundations of Encrypt-to-Self

The overall goal of this article is to provide a secure solution for outsourced storage. We identified the novel encrypt-to-self (ETS) primitive, which provides one-time secure encryption with authenticity guarantees that hold beyond key compromise, as the right tool to construct outsourced storage.[20] In this section we first formalize and study ETS, then formalize outsourced storage, and finally show how the former immediately implies the latter. This allows us to leave the outsourced storage topic aside in the remaining part of the paper and lets us instead fully focus on constructing and implementing ETS.

3.1 Syntax and Security of ETS

ETS consists of an encryption and a decryption algorithm, where the former translates a message to a *binding tag* and a ciphertext, and the latter recovers the message from the tag-ciphertext pair. For

[18] For instance, an adversary who doesn't know a value x but instead the values $H(x)$ and y, can compute $H(x \mid\mid y)$ by just continuing the iterative MD computation from chain value $H(x)$ on. Note this does not require inverting the compression function.

[19] In particular, for instance, SHA512/128(`"a"`) is not a prefix of SHA512/192(`"a"`).

[20] While ETS is novel, note that prior work explored the quite similar Encryptment primitive [7]. Encryptment is stronger than ETS, and less efficient to construct.

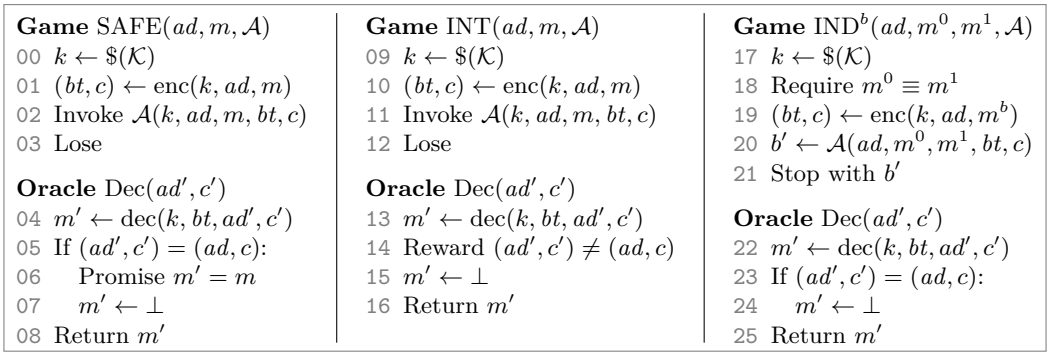

Fig. 1. Games for ETS. For the values ad', c' provided by the adversary we require that $ad' \in \mathcal{AD}, c' \in \mathcal{C}$. Assuming $\perp \notin \mathcal{M}$, we encode suppressed messages with $\perp$. For the meaning of instructions Stop with, Lose, Promise, Reward, and Require see Sec. 2.2.

versatility the two operations further support the processing of an associated-data input [12] which has to be identical for a successful decryption.

The task of the binding tag is to prevent forgery attacks: A user that holds an authentic copy of the binding tag will never accept any ciphertext they did not generate themselves, even if all their secrets become public. Note that while standard authenticated encryption (AE) does not provide this type of authentication, the encrypt-then-hash construction suggested in Sec. 1 does. In Sec. 4 we provide a considerably more efficient construction that uses a hash function's compression function as its core building block. Here, we define the generic syntax of ETS and formalize its security requirements.

Definition 1. *Let $\mathcal{AD}$ be an associated data space and let $\mathcal{M}$ be a message space. An encrypt-to-self* (ETS) *scheme for $\mathcal{AD}$ and $\mathcal{M}$ consists of algorithms* enc, dec*, a key space $\mathcal{K}$, a binding-tag space $\mathcal{B}t$, and a ciphertext space $\mathcal{C}$. The encryption algorithm* enc *takes a key $k \in \mathcal{K}$, associated data $ad \in \mathcal{AD}$ and a message $m \in \mathcal{M}$, and returns a binding tag $bt \in \mathcal{B}t$ and a ciphertext $c \in \mathcal{C}$. The decryption algorithm* dec *takes a key $k \in \mathcal{K}$, a binding tag $bt \in \mathcal{B}t$, associated data $ad \in \mathcal{AD}$ and a ciphertext $c \in \mathcal{C}$, and returns a message $m \in \mathcal{M}$. A shortcut notation for this API is*

$$\mathcal{K} \times \mathcal{AD} \times \mathcal{M} \to \text{enc} \to \mathcal{B}t \times \mathcal{C} \qquad \mathcal{K} \times \mathcal{B}t \times \mathcal{AD} \times \mathcal{C} \to \text{dec} \to \mathcal{M} \ .$$

Correctness and Security. We require of an ETS scheme that if a message m is processed to a tag-ciphertext pair with associated data ad, and a message m' is recovered from this pair using the same associated data ad, then the messages m, m' shall be identical. This is formalized via the SAFE game in Fig. 1.[21] In particular, observe that if the adversary queries Dec(ad, c) (for the authentic ad and c that it receives in line 02) and the dec procedure produces output m', the game promises that $m' = m$ (lines 05,06). Recall from Sec. 2.2 that this means the game stops with output T if $m' \neq m$. Intuitively, the scheme is *safe* if we can rely on $m' = m$, that is, if the maximum advantage $\mathbf{Adv}^{\text{safe}}(\mathcal{A}) := \max_{ad \in \mathcal{AD}, m \in \mathcal{M}} \Pr[\text{SAFE}(ad, m, \mathcal{A})]$ that can be attained by realistic adversaries $\mathcal{A}$ is negligible. The scheme is perfectly safe if $\mathbf{Adv}^{\text{safe}}(\mathcal{A}) = 0$ for all $\mathcal{A}$. We remark that the universal quantification over all pairs (ad, m) makes our advantage definition particularly robust.

Our security notions demand that the integrity of ciphertexts be protected (INT-CTXT), and that encryptions be indistinguishable in the presence of chosen-ciphertext attacks (IND-CCA). The notions are formalized via the INT and IND^0, IND^1 games in Fig. 1, where the latter two depend on some equivalence relation $\equiv \subseteq \mathcal{M} \times \mathcal{M}$ on the message space.[22] For consistency, in lines 07,15,24 we suppress the message

[21] The SAFETY term borrows from the Distributed Computing community. SAFETY should not be confused with a notion of security. Informally, safety properties require that "bad things" will not happen. (In the case of encryption, it would be a bad thing if the decryption of an encryption yielded the wrong message.) For an initial overview we refer to Wikipedia: `Safety_property` and for the details to [1].

[22] We use relation $\equiv$ (in line 18 of IND^b) to deal with certain restrictions that practical ETS schemes may feature. Concretely, our construction does not take effort to hide the length of encrypted messages, implying that indistinguishability is necessarily limited to same-length messages. In our formalization such a technical restriction can be expressed by defining $\equiv$ such that $m^0 \equiv m^1 \Leftrightarrow |m^0| = |m^1|$.

in all games if the adversary queries $\mathrm{Dec}(ad, c)$. This is crucial in the IND^b games, as otherwise the adversary would trivially learn which message was encrypted, but does not harm in the other games as the adversary already knows m. Recall from Sec. 2.3 that all algorithms can fail, and if they do, then the oracles immediately abort. This property is crucial in the INT game where the dec algorithm must fail for unauthentic input such that the oracle immediately aborts. Otherwise, the game will reward the adversary, that is the game stops with $\mathtt{T}$ (line 14). We say that a scheme provides *integrity* if the maximum advantage $\mathbf{Adv}^{\mathrm{int}}(\mathcal{A}) := \max_{ad\in\mathcal{AD},m\in\mathcal{M}} \Pr[\mathrm{INT}(ad, m, \mathcal{A})]$ that can be attained by realistic adversaries $\mathcal{A}$ is negligible, and that it provides *indistinguishability* if the same holds for the advantage $\mathbf{Adv}^{\mathrm{ind}}(\mathcal{A}) := \max_{ad\in\mathcal{AD},m^0,m^1\in\mathcal{M}} |\Pr[\mathrm{IND}^1(ad, m^0, m^1, \mathcal{A})] - \Pr[\mathrm{IND}^0(ad, m^0, m^1, \mathcal{A})]|$.

3.2 Sufficiency of ETS for Outsourced Storage

We define the syntax of an outsourced storage scheme. We model such a scheme as a set of stateful algorithms, where algorithm write is invoked to store data and algorithm read is invoked to retrieve it. We indicate the statefulness of the algorithms by appending the term $\langle st\rangle$ to their names, where st is the state variable.

Definition 2. *Let $\mathcal{M}$ be a message space. A storage outsourcing scheme for $\mathcal{M}$ consists of algorithms* gen, write, read*, a state space $\mathcal{ST}$, and a ciphertext space $\mathcal{C}$. The state generation algorithm* gen *takes no input and outputs an (initial) state $st \in \mathcal{ST}$. The storage algorithm* write *takes a state $st \in \mathcal{ST}$ and a message $m \in \mathcal{M}$, and outputs an (updated) state $st \in \mathcal{ST}$ and a ciphertext $c \in \mathcal{C}$. The retrieval algorithm* read *takes a state $st \in \mathcal{ST}$ and a ciphertext $c \in \mathcal{C}$, and outputs an updated state $st \in \mathcal{ST}$ and a message $m \in \mathcal{M}$. A shortcut notation for this API is*

$$\mathrm{gen} \to \mathcal{ST} \qquad \mathcal{M} \to \mathrm{write}\langle\mathcal{ST}\rangle \to \mathcal{C} \qquad \mathcal{C} \to \mathrm{read}\langle\mathcal{ST}\rangle \to \mathcal{M} \ .$$

Correctness and Security. We require of a storage outsourcing scheme that if a message m is processed to a ciphertext, and subsequently a message m' is recovered from this ciphertext, then the messages m, m' shall be identical. This is formalized via the SAFE game in Fig. 2. Observe boolean flag *is* ('in-sync') tracks whether the attack is active or passive. Initially $is = \mathtt{T}$, i.e., the attack is passive; however, once the adversary requests the reading of a ciphertext that is not the last created one, the game sets $is \leftarrow \mathtt{F}$ to flag the attack as active (line 11). For passive attacks the game promises that any m returned by the read procedure is the last one that was processed by the write procedure (line 13). Intuitively, the scheme is *safe* if the maximum advantage $\mathbf{Adv}^{\mathrm{safe}}(\mathcal{A}) := \Pr[\mathrm{SAFE}(\mathcal{A})]$ that can be attained by realistic adversaries $\mathcal{A}$ is negligible. The scheme is perfectly safe if $\mathbf{Adv}^{\mathrm{safe}}(\mathcal{A}) = 0$ for all $\mathcal{A}$.

Our security notions demand that the integrity of ciphertexts be protected (INT-CTXT), and that encryptions be indistinguishable in the presence of chosen-ciphertext attacks (IND-CCA). The notions are formalized via the INT and $\mathrm{IND}^0, \mathrm{IND}^1$ games in Fig. 2, where the latter two depend on some equivalence relation $\equiv \subseteq \mathcal{M} \times \mathcal{M}$ on the message space (see also Footnote 22). Recall from Sec. 2.3 that all algorithms can fail, and if they do, the oracles immediately abort. This property is crucial in the INT game where the read algorithm must fail for unauthentic input such that the adversary is not rewarded in the subsequent line in the Read oracle. For consistency we suppress the message in the Read oracle for passive attacks in all games if the adversary queries $\mathrm{Dec}(ad, c)$. This is crucial in the IND^b games, as otherwise the adversary would trivially learn which message was encrypted, but does not harm in the other games as the adversary already knows m for passive attacks. Furthermore, we remark the adversary is only allowed to query the Corrupt oracle if M contains at most 1 message, i.e., the ChWrite oracle was queried for $m^0 = m^1$. Otherwise, the adversary would be able to run the read procedure and trivially learn m. We say that a scheme provides *integrity* if the maximum advantage $\mathbf{Adv}^{\mathrm{int}}(\mathcal{A}) := \Pr[\mathrm{INT}(\mathcal{A})]$ that can be attained by realistic adversaries $\mathcal{A}$ is negligible, and that it provides *indistinguishability* if the same holds for the advantage $\mathbf{Adv}^{\mathrm{ind}}(\mathcal{A}) := |\Pr[\mathrm{IND}^1(\mathcal{A})] - \Pr[\mathrm{IND}^0(\mathcal{A})]|$.

Construction from ETS. Constructing secure outsourced storage from ETS is immediate: The write procedure samples a uniformly random key and runs the enc procedure of ETS to obtain a binding tag and ciphertext. It stores the binding tag (and key) in the state and returns the ciphertext. The read procedure gets the key and binding tag from the state, runs the dec procedure of ETS and returns the message. The details of this construction are in Fig. 3. The security argument is obvious.

Game SAFE($\mathcal{A}$)
00 $\mathrm{C}, \mathrm{M} \leftarrow \emptyset$
01 $is \leftarrow \mathtt{T}$
02 $st \leftarrow \mathrm{gen}$
03 Invoke $\mathcal{A}$
04 Lose

Oracle Write(m)
05 $c \leftarrow \mathrm{write}\langle st\rangle(m)$
06 If is:
07 $\quad \mathrm{C} \leftarrow \{c\}$
08 $\quad \mathrm{M} \leftarrow \{m\}$
09 Return c

Oracle Read(c)
10 $m \leftarrow \mathrm{read}\langle st\rangle(c)$
11 If $c \notin \mathrm{C}$: $is \leftarrow \mathtt{F}$
12 If is:
13 $\quad$ Promise $m \in \mathrm{M}$
14 $\quad m \leftarrow \bot$
15 Return m

Game INT($\mathcal{A}$)
16 $\mathrm{C} \leftarrow \emptyset$
17 $is \leftarrow \mathtt{T}$
18 $st \leftarrow \mathrm{gen}$
19 Invoke $\mathcal{A}$
20 Lose

Oracle Write(m)
21 $c \leftarrow \mathrm{write}\langle st\rangle(m)$
22 If is: $\mathrm{C} \leftarrow \{c\}$
23 Return c

Oracle Read(c)
24 $m \leftarrow \mathrm{read}\langle st\rangle(c)$
25 If $c \notin \mathrm{C}$: $is \leftarrow \mathtt{F}$
26 Promise is
27 If is: $m \leftarrow \bot$
28 Return m

Oracle Corrupt
29 Return st

Game IND^b($\mathcal{A}$)
30 $\mathrm{C}, \mathrm{M} \leftarrow \emptyset$
31 $is \leftarrow \mathtt{T}$
32 $st \leftarrow \mathrm{gen}$
33 $b' \leftarrow \mathcal{A}$
34 Stop with b'

Oracle ChWrite(m^0, m^1)
35 Require $m^0 \equiv m^1$
36 $c \leftarrow \mathrm{write}\langle st\rangle(m^b)$
37 If is:
38 $\quad \mathrm{C} \leftarrow \{c\}$
39 $\quad \mathrm{M} \leftarrow \{m^0, m^1\}$
40 Return c

Oracle Read(c)
41 $m \leftarrow \mathrm{read}\langle st\rangle(c)$
42 If $c \notin \mathrm{C}$: $is \leftarrow \mathtt{F}$
43 If is: $m \leftarrow \bot$
44 Return m

Oracle Corrupt
45 Require $|\mathrm{M}| \leq 1$
46 Return st

Fig. 2. Games for outsourced storage. For all values m, m^0, m^1, c provided by the adversary we require that $m, m^0, m^1 \in \mathcal{M}$ and $c \in \mathcal{C}$. Assuming $\bot \notin \mathcal{M}$, we encode suppressed messages with $\bot$. Boolean flag is ('in-sync') tracks whether the attack is active or passive. For the meaning of instructions Stop with, Lose, Promise, and Require see Sec. 2.2.

Proc gen
00 $\mathrm{S} \leftarrow \emptyset$
01 $st := \mathrm{S}$
02 Return st

Proc write$\langle st\rangle(m)$
03 $k \leftarrow \$(\mathcal{K})$
04 $(bt, c) \leftarrow \mathrm{enc}(k, \epsilon, m)$
05 $\mathrm{S} \leftarrow \{(k, bt)\}$
06 Return c

Proc read$\langle st\rangle(c)$
07 Require $\mathrm{S} \neq \emptyset$
08 $\{(k, bt)\} \leftarrow \mathrm{S}$
09 $m \leftarrow \mathrm{dec}(k, bt, \epsilon, c)$
10 Return m

Fig. 3. Construction for outsourced storage from ETS. If in line 07 the condition is not met, the read algorithm aborts with some error indicator. Recall from Sec. 2.3 that the read algorithm also aborts if the dec invocation in line 09 fails.

4 Construction of Encrypt-to-Self

We mentioned in Sec. 1 that a generic construction of ETS can be realized by combining standard symmetric encryption with a cryptographic hash function: one encrypts the message and computes the binding tag as the hash of the ciphertext. Here we provide a more efficient construction that builds on the compression function of a Merkle–Damgård hash function. To be more precise, our construction uses a tweakable compression function with tweak space $T = \{0, 1\}$, i.e., the domain of the compression function is extended by one bit (see Sec. 2.5). We provide a general definition below.

Definition 3. *For Σ an alphabet, $c, d \in \mathbb{N}^+$ with $c \leq d$, and a tweak space T, we define a tweakable compression function to be a function $F\colon \Sigma^d \times T \times \Sigma^c \to \Sigma^c$ that takes as input a block $B \in \Sigma^d$ from the data domain, a tweak $t \in T$ from the tweak space, and a string $C \in \Sigma^c$ from the chain domain, and outputs a string $C' \in \Sigma^c$ in the chain domain.*

We will write $F_t(B, C)$ as shorthand notation for $F(B, t, C)$. For practical tweakable compression functions the memory alignment value mav (see Sec. 2.4) will divide both c and d. When constructing an ETS scheme from F, because the compression function only takes fixed-size input, we need to map the (ad, m) input to a series of block–tweak pairs (B, t). We will refer to this mapping as the input *encoding*. We take a modular approach by fixing the encoding independently of the encryption engine, and detail the former in Sec. 4.1 and the latter in Sec. 4.2. Together they form an efficient construction of ETS.

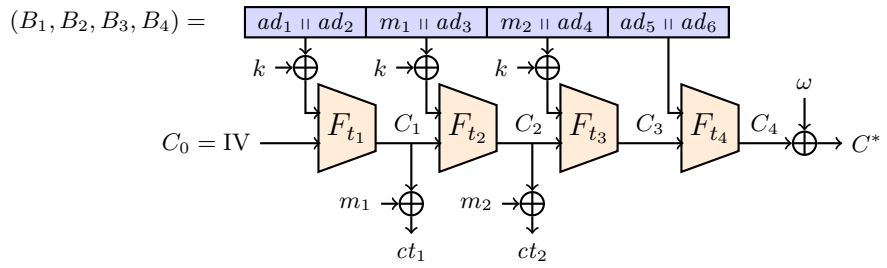

Fig. 4. Example for $\mathrm{enc}(k, ad, m)$ where $d = 2c$ and $ad = ad_1 \mid\mid \ldots \mid\mid ad_6$ and $m = m_1 \mid\mid m_2$ with $|ad| = 6c$ and $|m| = 2c$. For clarity we have made the blocks B_i, as they are output by the encoding function, explicit. Inspiration for this figure is drawn from `https://www.iacr.org/authors/tikz/`.

We first convey a rough overview of our ETS construction. In Fig. 4 we consider an example with block size d double the chaining value size c. We assume that key k is padded to size d. The first block B_1 only contains associated data and we XOR B_1 with the key k before we feed it into the compression function. From the second block, we start processing message data. We fill the first half of the block with message data m_1 and the second half with associated data ad_3, and XOR with the key. We also XOR m_1 with the current chaining value C_1, to generate a partial ciphertext ct_1. The same happens in the third block and we append ct_2 to the ciphertext. If there is associated data left after processing all message data we can load the entire block with associated data, which occurs in the fourth block. Note, we no longer need to XOR the key into the block after we have processed all message data, because at this point the input to the compression function will already be independent of the message m. After processing all blocks, we XOR an offset $\omega \in \{\omega_0, \omega_1\}$ with the chaining value, where ω_0, ω_1 are two distinct constants. The binding tag will be (a truncation of) the last chaining value C^*.[23] Note that the task of the encoding is not only to partition ad and m into blocks $B_1, B_2, \ldots$ as described, but also to derive tweak values $t_1, t_2, \ldots$ and the choice of the final offset ω in such a way that the overall encoding is injective.

4.1 Message Block Encoding

We turn to the technical component of our ETS construction that encodes the (ad, m) input into a series of output pairs (B, t) and the final offset value ω. For authenticity we require that the encoding is injective. For efficiency we require that the encoding is online (i.e., the input is read only once, left-to-right, and with small state), that the number of output pairs is as small as possible, and that the encoding preserves memory alignment (see Sec. 2.4). Syntactically, for the outputs we require that all $B \in \Sigma^d$, all $t \in T$, and $\omega \in \Omega$, where quantities c, d are those of the employed compression function, $T = \{0, 1\}$, and $\Omega \subseteq \Sigma^c$ is any two-element set. (Note that $|T| = 2$ allows us to use the tweaking approach from Sec. 2.5; further, in our implementations we use $\Omega = \{\omega_0, \omega_1\}$ where $\omega_0 = \texttt{0x00}^c$ and $\omega_1 = \texttt{0xa5}^c$.) Overall, the task we are facing is the following:

Task. Assume $|\Sigma| = 256$ and $\mathcal{AD} = \mathcal{M} = \Sigma^*$ and $T = \{0, 1\}$ and $|\Omega| = 2$. For $c, d \in \mathbb{N}^+$, $c < d$, find an injective encoding function $\mathrm{encode}\colon \mathcal{AD} \times \mathcal{M} \to (\Sigma^d \times T)^* \times \Omega$ that takes as input two finite strings and outputs a finite sequence of pairs $(B, t) \in \Sigma^d \times T$ and an offset $\omega \in \Omega$.

A detailed specification of our encoding (and decoding) function can be found in Fig. 6, but we present it here in text. Our construction does not use the decoding function, but we provide it anyway to show that the encoding function is indeed injective. Roughly, we encode as follows. We fill the first block with ad and for any subsequent block we load the message in the first part of the block and ad in the second part of the block. When we have processed all the message data, we load the full block with ad again. Clearly, we need to pad ad if it runs out before we have processed all message data. We do this by appending a special termination symbol $\diamond \in \Sigma$ to ad and then appending null bytes as needed. Similarly, we need to pad the message if the message length is not a multiple of c. Naturally, one might want to pad the message to a multiple of c. However, this is suboptimal: Consider the scenario where there are

[23] It will be crucial to fix ω_0, ω_1 such that they are distinct also after truncation.

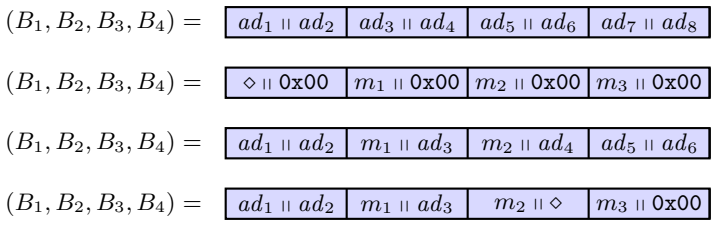

Fig. 5. Example encodings for the case $c = 1$ and $d = 2$.

$d - c + 1$ bytes remaining to be processed of associated data and 1 byte of message data. In principle, message and associated data would fit into a single block, but this would not be the case any longer if the message is padded to size c. On the other hand, for efficiency reasons we do not want to misalign all our remaining associated data. If we do not pad at all, when we process the next d bytes of associated data, we can only fit $d-1$ bytes in the block and have to put 1 byte into the next block. Hence, we pad m up to a multiple of the memory alignment value mav. Therefore, we prepend a padded message with its length $|m|$; this will uniquely determine where m stops. We then fill up with null bytes until reaching a multiple of mav. This restricts us to $c \leq 256$ bytes such that $|m|$ always can be encoded into a single byte. As far as we are aware, any current practical compression function satisfies this requirement.

In Fig. 5, for the artificially small case with $c = 1$ and $d = 2$ we provide four examples of what the blocks would look like for different inputs. The top row shows the encoding of 8 bytes of associated data and an empty message. The second row shows the encoding of empty associated data and 3 bytes of message data. The third row shows the encoding of 6 bytes of associated data and 2 bytes of message data. The final row shows the encoding of 3 bytes of associated data and 3 bytes of message data.

We have two ambiguities remaining. (1) How to tell whether ad was padded or not? Consider the first row in Fig. 5. What distinguishes the case $ad = ad_1 \mid\mid \ldots \mid\mid ad_7$ from $ad = ad_1 \mid\mid \ldots \mid\mid ad_7 \mid\mid ad_8$ with $ad_8 = \diamond$? A similar question applies to the message. (2) How to tell whether a block contains message data or not? Compare e.g., the first row with the third row. This is where the tweaks come into play.

First of all, we tweak the first block if and only if the message is empty. This fully separates the authentication-only case from the case where we have message input.

Next, if the message is non-empty, we use the tweaks to indicate when we switch from processing message data to ad-only: we tweak when we have consumed all of m, but still have ad left. Note the first block never processes message data, so the earliest block this may tweak is the second block and hence this rule does not interfere with the first rule. Furthermore, observe this rule never tweaks the final block, as by definition of being the final block, we do not have any associated data left to process.

Next, we need to distinguish between the cases whether m is padded or not. In fact, as the empty message was already taken care of, we need to do this only if m is at least one byte in size. As in this case the final block does not coincide with the first block, we can exploit that its tweak is still unused; we correspondingly tweak the final block if and only if m is padded. Obviously, this does not interfere with the previous rules.

Finally, we need to decide whether ad was padded or not. We do not want to enforce a policy of 'always pad', as this could result in an extra block and hence an extra compression function invocation. Instead, we use our offset output. We set the offset ω to ω_1 if ad was padded; otherwise we set it to ω_0.

This completes our description of the encoding function. The decoding function is a technical exercise carefully unwinding the steps taken in the encoding function, which we perform in Fig. 6. We obtain that for all $m \in \mathcal{M}, ad \in \mathcal{AD}$ we have $\text{decode}(\text{encode}(ad, m)) = (ad, m)$. It immediately follows that our encoding function is injective. For readability we have implemented the core functionality of the encoding in a coroutine called nxt, rather than a subroutine. Instead of generating the entire sequence of (B, t) pairs and returning the result, it will 'Yield' one pair and suspend its execution. The next time it is called (e.g., the next step in a for loop), it will resume execution from where it called 'Yield', instead of at the beginning of the function, with all of its state intact. The encode procedure is a simple wrapper that runs the nxt procedure and collects its output, but our authenticated encryption engine described in Sec. 4.2 will call the nxt procedure directly.

```
Proc encode(ad, m)
00 S[·] ← ·; i ← 0
01 For (B, t) ∈ nxt(ad, m):
02   If B ≠ ϵ:
03     i ← i + 1
04     S[i] ← (B, t)
05   Else: ω ← t
06 Return (S, ω)

Proc decode(S, ω)
07 ad ← ϵ; m ← ϵ
08 n ← |S|; j ← |S|
09 If n = 0:
10   Return (ad, m)
11 For i ← 1 to n:
12   (B_i, t_i) ← S[i]
13 For i ← 1 to n − 1:
14   If t_i = 1: j ← i
15 ad ←" B_1
16 For i ← 2 to j − t_n:
17   B_i ∥ B'_i ←_c B_i
18   m ←" B_i
19   ad ←" B'_i
20 If n > 1 ∧ t_n = 1:
21   l ∥ B_j ←_1 B_j
22   m' ∥ B_j ←_l B_j
23   m ←" m'
24   z ← −l mod mav
25   _ ∥ ad' ←_{z−1} B_j
26   ad ←" ad'
27 For i ← j + 1 to n:
28   ad ←" B_i
29 If ω = ω_1:
30   Split ad ∥ ⋄ ∥ 0* ← ad
31 Return (ad, m)

Proc nxt(ad, m)
32 ad_main ← T
33 m_main ← T
34 ω ← ω_0; t̄ ← 0; n ← 0
35 While ad ≠ ϵ ∨ m ≠ ϵ:
36   n ← n + 1
37   (B_n, t_n) ← (ϵ, 0)
38   If n > 1 ∧ m ≠ ϵ:
39     If |m| < c:
40       t̄ ← 1
41       j ← −|m| mod mav
42       m ← |m| ∥ m ∥ 0^{j−1}
43     l ← min(c, |m|)
44     B_n ∥ m ←_l m
45   d' ← d − |B_n|
46   If |ad| < d':
47     If ad_main:
48       ω ← ω_1
49       ad ←" ⋄
50       ad_main ← F
51     j ← d' − |ad|
52     ad ←" 0^j
53   ad' ∥ ad ←_{d'} ad
54   B_n ←" ad'
55   If m_main ∧ m = ϵ:
56     If n = 1 ∨ ad ≠ ϵ:
57       t_n ← 1
58       m_main ← F
59   If ad = ϵ ∧ m = ϵ:
60     If n > 1: t_n ← t̄
61   Yield (B_n, t_n)
62 Yield (ϵ, ω)

Proc enc(k, ad, m)
63 ct ← ϵ; C ← IV; i ← 0
64 For (B, t) ∈ nxt(ad, m):
65   If B ≠ ϵ:
66     i ← i + 1
67     If i = 1 ∨ m ≠ ϵ:
68       B ← B ⊕ k
69     If i > 1 ∧ m ≠ ϵ:
70       j ← min(c, |m|)
71       m' ∥ m ←_j m
72       C' ←_j C
73       ct ←" m' ⊕ C'
74     C ← F_t(B, C)
75 bt ←_taglen C ⊕ t
76 Return (bt, ct)

Proc dec(k, bt, ad, ct)
77 m ← ϵ; C ← IV; i ← 0
78 For (B, t) ∈ nxt(ad, ct):
79   If B ≠ ϵ:
80     i ← i + 1
81     If i = 1 ∨ ct ≠ ϵ:
82       B ← B ⊕ k
83     If i > 1 ∧ ct ≠ ϵ:
84       If |ct| ≥ c:
85         ct' ∥ ct ←_c ct
86         m ←" ct' ⊕ C
87         B ←^⊕ C ∥ 0^{d−c}
88       Else:
89         C' ←_{|ct|} C
90         m ←" ct ⊕ C'
91         j ← d − |ct| − 1
92         B ←^⊕ 0 ∥ C' ∥ 0^j
93     C ← F_t(B, C)
94 bt' ←_taglen C ⊕ t
95 If bt' ≠ bt: Fail
96 Return m
```

Fig. 6. ETS construction: encoder, decoder, encryptor, and decryptor. (Procedure nxt is a coroutine for encode, enc, and dec, see text.) Using global constants mav, c, d, taglen, and IV.

4.2 Encryption Engine

We now turn our focus to the encryption engine. We assume that the associated data and message are present in encoded format, i.e., as a sequence of pairs (B, t), where $B \in \Sigma^d$ is a block and $t \in \{0, 1\}$ is a tweak, and additionally an offset $\omega \in \{\omega_0, \omega_1\}$. To be precise, we will use the nxt procedure that generates the next (B, t) pair on the fly.

We specify the encryption and decryption algorithms in Fig. 6 and assume they are provided with a key of length d. As illustrated in Fig. 4, the main idea is to XOR the key with all blocks that are involved with message processing. For the skeleton of the construction, we initialize the chaining value C to IV and loop through the sequence of pairs (B, t) output by the encoding function, each iteration updating the chaining value $C \leftarrow F_t(B, C)$. We now describe each iteration of the enc procedure in more detail, where numbers in brackets refer to line numbers in Fig. 6. If the block is empty [65], we are in the final iteration and do not do anything. Otherwise, we check if we are in the first iteration or if we have message data left [67]. In this case we XOR the key into the block [68]. This ensures we start with an unknown input block and that subsequent inputs are statistically independent of the message block. If we only have *ad* remaining we can use the block directly as input to the compression function. If we have message data left we will encrypt it starting from the second block [69]. To encrypt, we take a chunk

of the message, XOR it with the chaining value of equal size and append the result to the ciphertext [70–73]. We only start encrypting from the second iteration as the first chaining value is public. Finally, we call the compression function F_t to update our chaining value [74]. Once we have finished the loop, the last pair (B, t) equals (ϵ, ω) by definition. So we XOR the offset ω with the chaining value C and truncate the result to obtain the binding tag [75]. We return the binding tag along with the ciphertext.

The dec procedure is similar to the enc procedure but needs to be slightly adapted. Informally, the nxt procedure now outputs a block $B = (ct \mid\mid ad)$ [78] instead of $B = (m \mid\mid ad)$ [64]. Hence, we XOR with the chaining variable [87,92] such that the block becomes $B = (m \mid\mid ad)$ and the compression function call takes equal input compared to the enc procedure. The case distinction handles the slightly different positioning of ciphertext in the blocks. Finally, there obviously is a check if the computed binding tag is equal to the stored binding tag [95].

In order to prove security, we need further assumptions on our compression function than the standard assumption of preimage resistance and collision resistance. For example, we need F to be difference unpredictable. Roughly, this notion says it is hard to find a pair (x, y) such that $F(x) = F(y) \oplus z$ for a given difference z. Moreover, we truncate the binding tag, so actually it should be hard to find a tuple such that this equation holds for the first $|bt|$ bits. We note collision resistance of F does not imply collision resistance of a truncated version of F [3]. However, such assumptions could be justified when one considers the compression function as a random function. Hence, instead of several ad hoc assumptions, we prove our construction secure directly in the random oracle model.

As described in Sec. 2.5 we tweak the SHA2 compression function by modifying the chaining value depending on the tweak. Let F be the tweakable compression function in Fig. 6, we denote with F' the (standard) compression function that will take as input the block and the (modified) chaining value. Let $H\colon \Sigma^d \times \Sigma^c \to \Sigma^c$ be a random oracle. We will substitute H for F' in our construction. We remark the BLAKE2b compression function is a tweakable compression function and it can be substituted for a random oracle directly. Hence, we focus on the security proof with a standard compression function, as the case with a tweakable compression function is a simplification of the proof with a standard compression function. In the random oracle model, our ETS construction from Fig. 6 provides integrity (Thm 1) and indistinguishability (Thm 2), assuming sufficiently large tag and key lengths. Here, we only state the theorems. We provide the security proofs in Appendix A.

Theorem 1. Let π be the construction given in Fig. 6, H a random oracle replacing the compression function, $\mathcal{A}$ an adversary, $\mathbf{Adv}^{\mathrm{int}}_{\pi}(\mathcal{A})$ the advantage that $\mathcal{A}$ has against π in the integrity game of Fig. 1 and q the number of random oracle queries (either directly or indirectly via Dec). We have,

$$\mathbf{Adv}^{\mathrm{int}}_{\pi}(\mathcal{A}) \leq q \cdot 2^{-|bt|}.$$

Theorem 2. Let π be the construction given in Fig. 6, H a random oracle replacing the compression function, $\mathcal{A}$ an adversary, $\mathbf{Adv}^{\mathrm{ind}}_{\pi}(\mathcal{A})$ the advantage that $\mathcal{A}$ has against π in the indistinguishability games of Fig. 1 and q the number of random oracle queries (either directly or indirectly via Dec). We have,

$$\mathbf{Adv}^{\mathrm{ind}}_{\pi}(\mathcal{A}) \leq q^2 \cdot 2^{-c} + q \cdot 2^{-|k|} + \mathbf{Adv}^{\mathrm{int}}_{\pi}(\mathcal{A}).$$

5 Implementation of Encrypt-to-Self

We implemented three versions of the ETS primitive. Precisely, we implemented the padding scheme and encryption engine from Fig. 6 in C, based on the compression functions of SHA256, SHA512, and BLAKE2 [10, 14] and the tweaking approach described in Sec. 2.5. All three functions are particularly good performers in software, where specifically SHA512 and BLAKE2 excel on 64-bit platforms. Roughly, we measured that the BLAKE2 version is about 50% faster than the SHA512 version, which in turn is about 50% faster than the SHA256 version.[24] We note that our code is written in pure C and in principle

[24] We do not provide cycle counts as we measured on outdated hardware (an Intel Core i3-2350M CPU @ 2.30GHz) and our numbers would not allow for a meaningful efficiency estimation on current CPUs. Note that

would benefit from assembly optimizations. Fortunately, however, all three compression functions are ARX (add–rotate–xor) designs so that the penalty of not hand-optimizing is not too drastic. Further, many freely available assembly implementations of the SHA2 and BLAKE2 core functions exist, e.g., in the OpenSSL package, and we made sure that our API abstractions are compatible with these, allowing for drop in replacements.

Acknowledgments

We thank the reviewers of ESORICS'20 for their helpful comments and appreciate the feedback provided by Cristina Onete. The research of Pijnenburg was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1). The research of Poettering was supported by the European Union's Horizon 2020 project FutureTPM (779391).

References

1. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2(3), 117–126 (1987), https://doi.org/10.1007/BF01782772
2. Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: Simpler, smaller, fast as MD5. In: Jacobson Jr., M.J., Locasto, M.E., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 13. LNCS, vol. 7954, pp. 119–135. Springer, Heidelberg (Jun 2013)
3. Biham, E., Chen, R.: Near-collisions of SHA-0. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 290–305. Springer, Heidelberg (Aug 2004)
4. Biham, E., Dunkelman, O.: A framework for iterative hash functions - HAIFA. Cryptology ePrint Archive, Report 2007/278 (2007), http://eprint.iacr.org/2007/278
5. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: New generation of memory-hard functions for password hashing and other applications. In: EuroS&P. pp. 292–302. IEEE (2016)
6. Chang, D., Nandi, M., Yung, M.: Indifferentiability of the hash algorithm BLAKE. Cryptology ePrint Archive, Report 2011/623 (2011), http://eprint.iacr.org/2011/623
7. Dodis, Y., Grubbs, P., Ristenpart, T., Woodage, J.: Fast message franking: From invisible salamanders to encryptment. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 155–186. Springer, Heidelberg (Aug 2018)
8. Gueron, S.: Memory encryption for general-purpose processors. IEEE Secur. Priv. 14(6), 54–62 (2016)
9. Kaliski, B.: PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Sep 2000), https://rfc-editor.org/rfc/rfc2898.txt
10. NIST: FIPS 180-4: Secure Hash Standard (SHS). Tech. rep., NIST (2015), http://dx.doi.org/10.6028/NIST.FIPS.180-4
11. Pijnenburg, J., Poettering, B.: Key assignment schemes with authenticated encryption, revisited. IACR Trans. Symmetric Cryptol. 2020(2) (2020)
12. Rogaway, P.: Authenticated-encryption with associated-data. In: Atluri, V. (ed.) ACM CCS 2002. pp. 98–107. ACM Press (Nov 2002)
13. Rogaway, P.: Nonce-based symmetric encryption. In: Roy, B.K., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 348–359. Springer, Heidelberg (Feb 2004)
14. Saarinen, M.O., Aumasson, J.: The BLAKE2 cryptographic hash and message authentication code (MAC). RFC 7693 (2015), https://rfc-editor.org/rfc/rfc7693.txt

A Security Proofs

Let $H\colon \Sigma^d \times \Sigma^c \to \Sigma^c$ be a random oracle. Recall we consider an instantiation with a standard (non-tweakable) compression function F' transformed as described in Sec. 2.5 into a tweakable compression function F. We replace F', used internally by F, with random oracle H.

Theorem 1. *Let π be the construction given in Fig. 6, H a random oracle replacing the compression function, $\mathcal{A}$ an adversary, $\mathbf{Adv}^{\mathrm{int}}_{\pi}(\mathcal{A})$ the advantage that $\mathcal{A}$ has against π in the integrity game of Fig. 1 and q the number of random oracle queries (either directly or indirectly via* Dec*). We have,*

$$\mathbf{Adv}^{\mathrm{int}}_{\pi}(\mathcal{A}) \leq q \cdot 2^{-|bt|}.$$

https://blake2.net/ reports a performance of raw BLAKE2(b) on Skylake of roughly 1 GB/s. Our ETS adds the message encryption step on top of this (a series of memory accesses and XOR operations per message block), so it seems fair to expect an overall performance of more than 800 MB/s.

Proof. For all $ad \in \mathcal{AD}, m \in \mathcal{M}$ we will show that

$$\Pr[\text{INT}(ad, m, \mathcal{A})] \leq q \cdot 2^{-|bt|}.$$

Let $ad \in \mathcal{AD}$ be associated data and let $m \in \mathcal{M}$ be a message. The game $\text{INT}(ad, m, \mathcal{A})$ samples a uniformly random key $k \in \mathcal{K}$ and computes $(bt, c) = \text{enc}(k, ad, m)$. $\mathcal{A}$ wins the INT game if it provides a pair $(ad', c') \neq (ad, c)$ such that $\text{dec}(k, bt, ad', c')$ succeeds, which only happens if $bt' = bt$. Because the encoding function is injective it follows $\text{encode}(ad', c') \neq \text{encode}(ad, c)$. Recall the encoding function outputs a sequence S and an offset ω. Hence $S' \neq S$ or $\omega' \neq \omega$. Let us first assume $S' = S$. Let C_n denote the final chaining variable. Because the sequences are equal, we will arrive at $C'_n = C_n$. We must have $\omega' \neq \omega$, but clearly $C_n \oplus \omega_0$ is not equal to $C_n \oplus \omega_1$ (even after truncation), that is, $bt' \neq bt$. We have a contradiction and conclude $S' \neq S$.

For the case $S' \neq S$, let us now assume the subcase $\omega' \neq \omega$. The first $|bt|$ bits of C'_n must equal the first $|bt|$ bits of $C_n \oplus \omega \oplus \omega'$, i.e., $\mathcal{A}$ must find a partial preimage. Because H is a random oracle, $\mathcal{A}$ would succeed with probability at most $q \cdot 2^{-|bt|}$, where q is the number of queries. In the other subcase we have $\omega' = \omega$. Then the first $|bt|$ bits of C'_n must equal the first $|bt|$ bits of C_n, i.e., the first $|bt|$ bits of $H(B'_n, \hat{C}'_{n-1})$ must equal the first $|bt|$ bits of $H(B_n, \hat{C}_{n-1})$, where $\hat{C}'_{n-1}, \hat{C}_{n-1}$ are the chaining values C'_{n-1}, C_{n-1} after applying tweaks t'_n, t_n, respectively. If the inputs are not equal, $\mathcal{A}$ has found a partial second preimage. Since H is a random oracle, $\mathcal{A}$ would succeed with probability at most $q \cdot 2^{-|bt|}$, where q is the number of oracle queries. However, if the inputs are equal we know $\hat{C}'_{n-1} = \hat{C}_{n-1}$. Let us write $\hat{C}'_{n-1} = C'_{n-1} \oplus \tau'$ and $\hat{C}_{n-1} = C_{n-1} \oplus \tau$. We obtain $C'_{n-1} = C_{n-1} \oplus \tau \oplus \tau'$. Thus, either $\mathcal{A}$ has found a preimage or $C'_{n-1} = C_{n-1}$. We can repeat the argument to reason about C'_{n-2}, C_{n-2}, etc. If we eventually conclude $C'_0 = C_0$, we know one of the sequences is longer, otherwise they would be equal. Thus, we have $H(B_0, \hat{C}_{-1}) = C_0$. $\mathcal{A}$ has found a preimage of $C_0 = \text{IV}$. Because H is a random oracle, $\mathcal{A}$ would succeed with probability at most $q \cdot 2^{-c}$. □

Theorem 2. *Let π be the construction given in Fig. 6, H a random oracle replacing the compression function, $\mathcal{A}$ an adversary, $\mathbf{Adv}^{\text{ind}}_{\pi}(\mathcal{A})$ the advantage that $\mathcal{A}$ has against π in the indistinguishability games of Fig. 1 and q the number of random oracle queries (either directly or indirectly via* Dec*). We have,*

$$\mathbf{Adv}^{\text{ind}}_{\pi}(\mathcal{A}) \leq q^2 \cdot 2^{-c} + q \cdot 2^{-|k|} + \mathbf{Adv}^{\text{int}}_{\pi}(\mathcal{A}).$$

Proof. Other than the challenge pair (ad, c), we can assume the decryption oracle rejects all queries by $\mathcal{A}$. Otherwise $\mathcal{A}$ would immediately win the integrity game and the theorem holds. Encryption is done by XORing the message with the chaining variable. As long as the chaining variable never repeats, each input to H is a fresh query that has not been seen before. Then H will provide fresh, uniformly random output, as it is a random oracle. By a standard birthday argument we can bound the probability of a collision by $q^2 \cdot 2^{-c}$. Moreover, each chaining variable that is used to encrypt is output of a query to H that used the key k in the input. Additionally each block that has message data as input also takes the key k as input. Thus if $\mathcal{A}$ does not know k it cannot query H with it to obtain the chaining variable. The key is only used directly as input to the compression function, so if $\mathcal{A}$ learns (or guesses) the key it successfully computed a preimage of H. However, this happens with probability at most $q \cdot 2^{-|k|}$. □