Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC

José Bacelar Almeida[1,2], Manuel Barbosa[1,3], Gilles Barthe[4], and François Dupressoir[4]

[1] HASLab – INESC TEC
[2] University of Minho
[3] DCC-FC, University of Porto
[4] IMDEA Software Institute

Abstract. We provide further evidence that implementing software countermeasures against timing attacks is a non-trivial task and requires domain-specific software development processes: we report an implementation bug in the S2N library, recently released by AWS Labs. This bug (now fixed) allowed bypassing the balancing countermeasures against timing attacks deployed in the implementation of the MAC-then-Encode-then-CBC-Encrypt (MEE-CBC) component, creating a timing side-channel similar to that exploited by Lucky 13.
Although such an attack could only be launched when the MEE-CBC component is used in isolation – Albrecht and Paterson recently confirmed in independent work that S2N's second line of defence provides adequate mitigation – its existence shows that conventional software validation processes are not being effective in this domain. To solve this problem, we define a methodology for proving security of implementations in the presence of timing attackers: first, prove *black-box security* of an algorithmic description of a cryptographic construction; then, establish *functional correctness* of an implementation with respect to the algorithmic description; and finally, prove that the implementation is *leakage secure*.
We present a proof-of-concept application of our methodology to MEE-CBC, bringing together three different formal verification tools to produce an assembly implementation of this construction that is verifiably secure against adversaries with access to some timing leakage. Our methodology subsumes previous work connecting provable security and side-channel analysis at the implementation level, and supports the verification of a much larger case study. Our case study itself provides the first provable security validation of complex timing countermeasures deployed, for example, in OpenSSL.

1 Introduction

There is an uncomfortable gap between provable security and practical implementations. Provable security gives strong guarantees that a cryptographic construction is secure against efficient *black-box* adversaries. Yet, implementations of provably secure constructions may be vulnerable to practical attacks, due to implementation errors or side-channels. The tension between provable security and cryptographic engineering is illustrated by examples such as the MAC-then-Encode-then-CBC-Encrypt construction (MEE-CBC), which is well-understood from the perspective of provable

security [24,29], but whose implementation has been the source of several practical attacks in TLS implementations. These security breaks are, in the case of MEE-CBC, due to vulnerable implementations providing the adversary with padding oracles, either through error messages [32], or through non-functional behaviours such as timing [19,2]. These examples illustrate two shortcomings of provable security when it comes to dealing with implementations. First, the computational model relies on an idealized execution model, and the algorithmic descriptions used in proofs elide many potentially critical details; these details must be filled by implementors, who may not have the specialist knowledge required to make the right decision. Second, attackers targeting real-world platforms may exploit leakage via physical side-channels to break a system, which is absent in the black-box abstractions in which proofs are obtained.

These shortcomings are addressed independently by *real-world cryptography* and *secure coding methodologies*, both of which have their own limitations. Real-world cryptography [20] is a branch of provable security that incorporates lower-level system features in security notions and proofs (for example, precise error messages or message fragmentation). Real-world cryptography is a valuable tool for analyzing the security of real-world protocols such as TLS or SSH, but it is still giving its first steps in addressing side-channels [18] and, until now, it has remained disconnected from actual implementations. Secure coding methodologies effectively mitigate side-channel leakage; for instance, the constant-time methodology [15,23] is consensual among practitioners as a means to ensure a *good* level of protection against timing and cache-timing attacks. However, a rigorous justification of such techniques and their application is lacking and they are disconnected from provable security, leaving room for subtle undetected vulnerabilities even in carefully tailored implementations.

In this paper we show how the real-world crypto approach can be extended to formally capture the guarantees that implementors empirically pursue using secure coding techniques.

1.1 Our Contributions

Recent high-visibility attacks like Lucky 13 [2] have shown that timing leakage can be exploited in practice to break the security of pervasively used protocols such as TLS, and have led practitioners to pay renewed attention to software countermeasures against timing attacks. Two prominent examples of this are the recent reimplementation of MEE-CBC decryption in OpenSSL [26], which enforces a constant-time coding policy as mitigation for the Lucky 13 attack, and the *defense in depth* mitigation strategy adopted by Amazon Web Services - Labs (AWS-Labs) in a new implementation of TLS (called S2N), where various fuzzing- and balancing-based timing countermeasures are combined to reduce the amount of information leaked through timing. However, the secure-coding efforts of cryptography practitioners are validated using standard software engineering techniques such as testing and code reviews, which are *not* designed to enforce provable correctness or security guarantees.

As a first contribution and motivation for our work, we provide new evidence of this latent problem by recounting the story of Amazon's recently released S2N library, to which we add a new chapter.

NEW EVIDENCE IN S2N. In June 2015, AWS-Labs made public a new open-source implementation of the TLS protocol call S2N [31]. The library is designed to be "small, fast, with simplicity as a priority". By excluding rarely used options and extensions, the implementation is around only 6K lines of code. The authors also report extensive validation, including three external security evaluations and penetration tests. The library's source code and documentation are all publicly available from the S2N GitHub repository.[5]

Very recently, Albrecht and Paterson [1] presented a detailed analysis of the countermeasures against timing attacks in the original release of S2N, in light of the lessons learned in the aftermath of the Lucky 13 attack [2]. In their study, these authors found that the implementation of the MEE-CBC component was not properly balanced, and exposed a timing attack vector that was exploitable using Lucky 13-like techniques. Furthermore, they found that the second layer of countermeasures that randomizes error reporting delays was insufficient to remove the attack vector. Intuitively, the granularity of the randomized delays was large enough in comparison to the data-dependent timing variations generated by the MEE-CBC component that they could be 'filtered out' leaving an exploitable side-channel. As a response to these findings, the S2N implementation was patched,[6] and both layers of countermeasures were improved to remove the attack vector.

Unfortunately, this was not the end of the story. In this paper we report an implementation bug in this "fixed" version of the library, as well as a timing attack akin to Lucky 13 that bypassed once more the branch-balancing timing countermeasures deployed in the MEE-CBC implementation of this library. This implementation bug was subtly hidden in the implementation of the timing countermeasures themselves, which were added as mitigation for the attack reported by Albrecht and Paterson [1]. We show that the bug rendered the countermeasure code in the MEE-CBC component totally innefective by presenting a timing attack that breaked the MEE-CBC implementation when no additional timing countermeasures were present.

The implementation bug and timing attack were reported to AWS-Labs on September 4, 2015. The problem was promptly acknowledged and the current head revision of the official S2N repository[7] has been patched to remove the bug and potential attack vector from the MEE-CBC implementation. Subsequent discussions with AWS-Labs and the authors of [1] were conducted to assess the impact of our attack on the full S2N stack. The conclusion was that S2N's second line of defence (the finer grained error reporting delay randomization mechanism validated by Albrecht and Paterson) is sufficient to thwart potential exploits of the timing side-channel that was created by the bug. This means that systems relying on unpatched but *complete* versions of the library are safe. On the other hand, any system relying directly on the unpatched MEE-CBC implementation, without the global randomized delay layer, will be vulnerable to our attack and should upgrade to the latest version.

[5] `https://github.com/awslabs/s2n`

[6] See the details of the applied fixes in `https://github.com/awslabs/s2n/commit/4d3729`.

[7] `https://github.com/awslabs/s2n`

THE NEED FOR FORMAL VALIDATION. The sequence of events reported above[8] shows that timing countermeasures are extremely hard to get right and very hard to validate. Our view is that implementors currently designing and deploying countermeasures against side-channel attacks face similar problems to those that were faced by the designers of cryptographic primitives and protocols before the emergence of provable security. On the one hand, we lack a methodology to rigorously characterize and prove the soundness of existing designs such as the ones deployed, for example, in openSSL; on the other hand, we have no way of assessing the soundness of new designs, such as those adopted in S2N, except via empirical validation and trial-and-error. This leads us to the following natural question: *can we bring the mathematical guarantees of provable security to cryptographic implementations?* We take two steps towards answering this question.

A CASE STUDY: CONSTANT-TIME MEE-CBC. Our second and main contribution is the first formal and machine-checked proof of security for an x86 implementation of MEE-CBC in an attack model that includes control-flow and cache-timing channels. In particular, our case study validates the style of countermeasures against timing attacks currently deployed in the OpenSSL implementation of MEE-CBC. We achieve this result by combining three state-of-the-art formal verification tools: i. we rely on EasyCrypt [8,7] to formalize a specification of MEE-CBC and some of the known provable security results for this construction;[9] ii. we use Frama-C to establish a functional equivalence result between EasyCrypt specifications and C implementations; and iii. we apply the CompCert certified compiler [27] and the certified information-flow type-system from [5] to guarantee that the compiled implementation does not leak secret information through the channels considered, and that the compiled x86 code is correct with respect to the EasyCrypt specification proved secure initially.

A FRAMEWORK FOR IMPLEMENTATION SECURITY. To tie these verification results together, we introduce — as our third contribution — a framework of definitions and theorems that abstracts the details of the case study. This framework yields a general methodology for proving security properties of low-level implementations in the presence of adversaries that may observe leakage. This methodology relies on separating three different concerns: i. *black-box specification security*, which establishes the computational security of a functional specification (here one can adopt the real-world cryptography approach); ii. *implementation correctness*, which establishes that the considered implementation behaves, as a black-box, exactly like its functional specification; and iii. *leakage security*, which establishes that the leakage due to the execution of the implementation code *in some given leakage model* is independent from its secret in-

[8] The very interesting blog post in `http://blogs.aws.amazon.com/security/post/TxLZP6HNAYWBQ6/s2n-and-Lucky-13` analyses these events from the perspective of the AWS Labs development team.

[9] To formalize all known results for MEE-CBC would be beyond the scope of this paper, and we pragmatically assume that our EasyCrypt specification of the construction inherits all the security properties that have been proved by other authors in the literature, albeit not machine checked. In other words, in addition to the properties we formalized, we assume that our MEE-CBC specification satisfies the standard black-box notions of security for authenticated encryption as proved, for example, in [29]

puts. Our main theorem, which is proven using the previous methodology, establishes that our x86 implementation retains the black-box security properties of the MEE-CBC specification, i.e., it is a secure authenticated encryption scheme, in the presence of strong timing attacker.

We insist that we do *not* claim to formally or empirically justify the validity of any particular leakage model: for this we rely on the wisdom of practitioners. What we *do* provide is a means to take a well-accepted leakage model, and separately and formally verify, through leakage security, that a concrete deployment of a particular countermeasure in a given implementation does in fact guarantee the absence of leakage in the chosen leakage model.

1.2 Outline

In Section 2, we describe the MEE-CBC construction and informally discuss its security at specification- and implementation-level. We then present our attack on AWS-Labs' MEE-CBC implementation (Section 3), motivating the definitions for implementation-level security notions and the statement of our main theorem (Section 4). In Section 5, we introduce our methodology, before detailing its application to MEE-CBC in Section 6. We then present and discuss some benchmarking results in Section 7. Finally, we discuss potential extensions to our framework not illustrated by our case study (Section 8) and describe further related work in Section 9. We conclude the paper and discuss directions for future work in Section 10.

2 Case study: MEE-CBC

MAC-then-Encode-then-CBC-Encrypt (MEE-CBC) is an instance of the MAC-then-Encrypt generic construction that combines a block cipher used in CBC mode with some padding and a MAC scheme in order to obtain an authenticated encryption scheme. The details of the construction are presented in Figure 1. We consider the specific instantiation of the construction that is currently most widely used within TLS: i. A MAC tag of length tlen is computed over the TLS record header hdr, a sequence number seq and the payload pld. The length of the authenticated string is therefore the length of the payload plus a small and fixed number of bytes. Several MAC schemes can be used to authenticate this message, but we only consider HMAC-SHA256. ii. The CBC-encrypted message m comprises the payload pld concatenated with the MAC tag (the sequence number is not transmitted and the header is transmitted in the clear). iii. The padding added to m comprises plen bytes of value $\mathsf{plen} - 1$, where plen may be any value in the range $[1..256]$, such that $\mathsf{plen} + |\mathsf{m}|$ is a multiple of the cipher's block size. iv. We use AES-128 as block cipher, which fixes a 16-byte block size.

The details of the MAC computation are also important for a clear presentation of our first result, so we also include a detailed description of the HMAC algorithm in Figure 2. We consider a hash function such as SHA-256, which follows the Merkle-Damgärd paradigm: a compression function HComp is iterated to gradually combine the already computed hash value with a new 64-byte message block (hash values are tlen bytes long). We specify the MAC computation as it is often implemented, with a

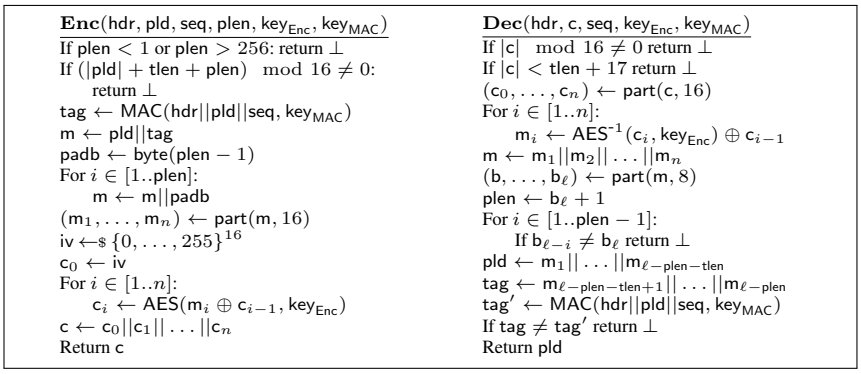

Fig. 1. The MEE-CBC construction. Lengths and block sizes are expressed in bytes. byte denotes the encoding of an integer value into 8 bits. $\mathsf{part}(\mathsf{m}, k)$ denotes the partitioning of a sequence of bytes m into blocks of k bytes each.

separate state initialization algorithm Initialize, a buffering Update routine that iteratively adds data to the initialized HMAC computation without explicit concatenation, and a Finalize routine that signals the end of data processing and triggers the final computations that produce the actual tag value. For simplicity, we assume all strings are byte-aligned and the MAC key is 0-padded to the hash-block length (64-bytes here). At the high level, the construction computes

$$H((\mathsf{key}_{\mathsf{MAC}} \oplus \mathsf{opad}) \,||\, H((\mathsf{key}_{\mathsf{MAC}} \oplus \mathsf{ipad}) \,||\, \mathsf{hdr} \,||\, \mathsf{seq} \,||\, \mathsf{pld}))\,.$$

Signaled in Figure 2 is an important detail at this point of the operation of Finalize: depending on the length of the data remaining in the buffer from the last Update call, the call to Finalize may trigger either 1 or 2 calls to the hash compression function to complete the inner hash computation, due to the HMAC-specific padding format. The HMAC computation is concluded by calculating the outer hash, which always takes one final compression function invocation.

INFORMAL SECURITY DISCUSSION. The theoretical security of MEE-CBC has received a lot of attention in the past, due to its high-profile usage in the SSL/TLS protocol. Although it is well-known that the MAC-then-Encrypt construction does *not* generically yield a secure authenticated encryption scheme [10], the particular instantiation used in TLS has been proven secure [24,28,29]. The most relevant result for this paper is that by Paterson et al. [29]. Crucially, their high-level proof explicitly clarifies the need for the implementation to not reveal, in any way, which of the padding or MAC check failed on decryption failures. This is exactly the kind of *padding oracles* exploited in practical attacks against MEE-CBC such as Lucky 13 [2].

After the disclosure of the Lucky 13 [2] attack, significant effort was invested into identifying all potential sources of timing leakage in the MEE-CBC decryption algorithm. The implementation subsequently incorporated into OpenSSL, for example, deploys constant-time countermeasures that guarantee the following behaviours [26]: i. removing the padding and checking its well-formedness occurs in constant-time; ii. the MAC of the unpadded message is always computed, even for bad padding; iii. the

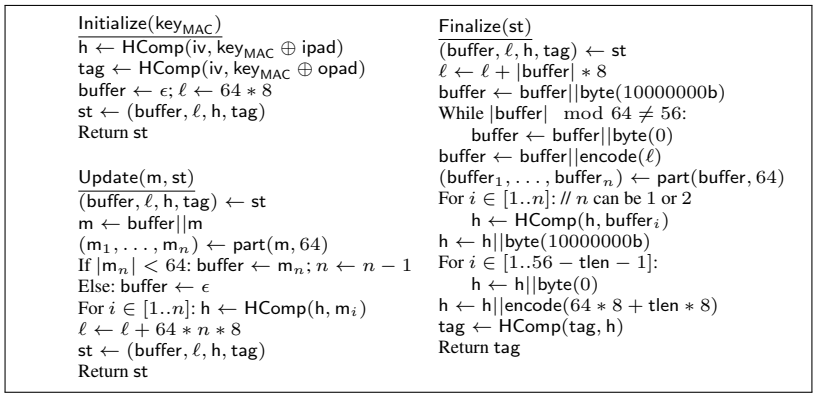

Fig. 2. The HMAC construction. Lengths and block sizes are expressed in bytes. encode denotes the encoding of an integer value into 8 bytes. opad and ipad denote constant 64-byte strings, whereas iv denotes a constant tlen-byte string, all are fixed by the HMAC specification.

MAC computation involves the same number of calls to the underlying compression function regardless of the number of hash input blocks in the decoded message, and regardless of the length of the final hash block (which may cause an additional block to be computed due to the internal Merkle-Damgärd length padding); and iv. the transmitted MAC is compared to the computed MAC in constant-time (the transmitted MAC's location in memory, which may be leaked through the timing of memory accesses, depends on the plaintext length). *Constant-time*, here and in the rest of this paper, is used to mean that the trace of program points and memory addresses accessed during the execution is independent from the initial value of secret inputs. In particular, we note that the OpenSSL MEE-CBC implementation is *not* constant time following this definition: the underlying AES implementation uses look-up table optimizations that make secret-dependent data memory accesses and may open the way to cache-timing attacks.

OUR IMPLEMENTATION. The main result of this paper is a security theorem for an x86 assembly implementation of MEE-CBC (MEE-CBC$_{x86}$). The implementation is compiled using CompCert from standard C code that replicates the countermeasures against timing attacks currently implemented in the OpenSSL library [26]. We do not use the OpenSSL code directly because the code style of the library (and in particular its lack of modularity) makes it a difficult target for verification. Furthermore, we wish to fully prove constant-time security, which we have seen is not achieved by OpenSSL. However, a large part of the code we verify is existing code, taken from the NaCl library [16] without change (for AES, SHA256 and CBC mode), or modified to include the necessary countermeasures (HMAC, padding and MEE composition). Our C code is composed of the following modules, explicitly named for later reference: i. AES128$_{NaCl}$ contains the NaCl implementation of AES128; ii. HMACSHA256$_{NaCl}$ contains a version of the NaCl implementation of HMAC-SHA256 extended with timing countermeasures mimicking those described in [26]; and iii. MEE-CBC$_{C}$ contains an implementation of MEE-CBC using AES128$_{NaCl}$ and HMACSHA256$_{NaCl}$. We do not include the code in the paper due to space constraints. However, we include in Figure 8

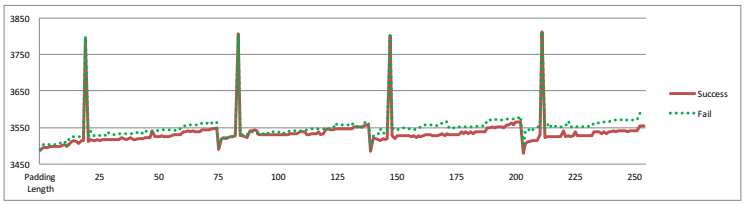

Fig. 3. Verification time (in clock cycles) for successful and unsuccessful verifications, for a record length of 288 and padding varying from 1 to 255 bytes.

(Appendix A) a representative snippet of the constant-time secure coding style which implements PKCS#7 decoding.

As we prove later in the paper, a strict adherence to this coding style is indeed sufficient to guarantee security against attackers that, in addition to input/output interaction with the MEE-CBC implementation, also obtain full traces of program counter and memory accesses performed by the implementation. However, not all TLS implementations have adopted a strict adherence to constant-time coding policies in the aftermath of the Lucky 13 attack. We now present the case of Amazon's S2N library, discussing their choice of countermeasures, and describing a bug in their implementation that leads to an attack.

3 Breaking the MEE-CBC implementation in S2N

IMPLEMENTATION AND COUNTERMEASURES. The S2N code for MEE-CBC decryption is given in Figure 9 (Appendix B). Although parts of the computation are written in the constant-time style, such as the padding validation on lines 45 to 49, there are many (intentional) deviations from a strict constant-time coding policy. For example, no attempt is made to de-correlate memory accesses from the sensitive padding_length value. As an alternative, the code includes countermeasures that intend to balance the execution time of secret-dependent conditional branches that might lead to significant variability in the execution time. In short, the goal of these countermeasures is to ensure that the total number of calls to the hash compression function is always the same, independently of the actual padding length. These are visible in two points in the code where the MAC is being computed: i. line 26 contains a call to a special Finalize routine and ii. line 31 contains a dummy call to the Update routine.

A BUG. The bug we found resides in the special Finalize routine, which we reproduce in Figure 10 (Appendix B). The comments on lines 19-23 reveal a subtle implementation error: the padding introduced by SHA-256 includes 8 bytes for length encoding plus one mandatory 10000000b byte that signals the padding start. This means that at least 9 bytes of space are needed in a hash block to encode the padding, and not 8 as checked by the implementation. This oversight means that the dummy compression function invocation on line 30 is invoked unnecessarily for padding values that trigger this corner case (there are exactly 4 such padding values, which can be easily inferred from the length of the encrypted record).

Figure 3 shows the execution time of S2N's MEE-CBC decryption routine as a function of padding length. The timing differences, correspond exacly to one invocation of the compression function (or $\sim$300 clock cycles, 10% variation on a 300-byte record). The leakage the bug produces is similar in size to that exploited by AlFardan and Paterson [2] to recover plaintexts. We have implemented a padding-oracle-style attack on the MEE-CBC decryption routine to recover single plaintext bytes from a ciphertext: one simply measures the decryption time to check if the recovered padding length causes the bug to activate and proceeds by trial and error.[10] The attack can be extended to full plaintext recovery using the same techniques reported in [2].

DISCUSSION. We already discussed the real-world impact of our attack and our disclosure interaction with AWS-Labs in the introduction of this paper. In short, the bug was promptly patched and it was found not to represent a risk to S2N users due to the deployment of additional timing countermeasures at the error reporting stage, elsewhere in the S2N software stack. However, we insist that for the purpose of this paper it is *not* the real-world impact of our attack that matters, but the software bug that gave rise to it in the first place. Indeed the existence of such a programming bug and the fact that it remained undetected through AWS Labs' code validation process (and in particular despite unit testing specifically designed to detect timing side-channels) reveal that there is a need for a formal framework in which to rigorously prove that an implementation is secure against timing attacks. This is what we set out to do in the rest of the paper.

4 Security definitions and main theorem

After a brief reminder of the syntax and security notions for secret key encryption relevant to our case study, we introduce and discuss the corresponding implementation-level security notions for the constant-time leakage model and state our main theorem. Cryptographic implementations are often hardwired at a particular security level, which means that asymptotic security notions are not adequate to capture the provided security guarantees. We therefore omit the security parameter in all our definitions. For simplicity we also keep the running time of algorithms implicit in our notations, although we take care to account for it in our security proofs and show that there is no hidden *slackness* in our reductions.

4.1 Secret Key Encryption

We recall that a secret-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is specified as three algorithms: i. a probabilistic key generation algorithm $\mathsf{Gen}()$ that returns a secret key SK; ii. a probabilistic encryption algorithm $\mathsf{Enc}(\mathsf{m}, \mathsf{SK}; \mathsf{r})$ that returns a ciphertext c on input a message m, the secret key SK, and some random coins r; and iii. a deterministic decryption algorithm $\mathsf{Dec}(\mathsf{c}, \mathsf{SK})$ that returns either a message m or a failure symbol $\perp$ on input a ciphertext c and secret key SK. We denote the set of valid messages with

[10] Plaintext recovery is easier than in Lucky 13, because leakage occurs whether or not the padding string is correct. This is visible in Figure 3 as synchronized spikes on both graphs.

MsgSp and adopt standard notions of correctness, confidentiality (IND$-CPA) and integrity (INT-PTXT and INT-CTXT) for authenticated symmetric encryption schemes. We include formal definitions of these notions in Appendix D.

Our goal in the rest of this section is to adapt these standard notions to formally capture implementation-level security. In particular, we wish to give the adversary the ability to observe the leakage produced by the computation of its oracle queries. We first give generic definitions for some core concepts.

4.2 Implementation: languages, leakage and generation

For the sake of generality, our definitions abstract the concrete implementation languages and leakage models adopted in our case study. We later instantiate these definitions with a black-box security model for C implementations and a timing leakage model for x86 assembly implementations.

LANGUAGE, LEAKAGE AND MACHINE. Given an implementation language $\mathcal{L}$, we consider a machine $\mathbb{M}$ that animates its semantics. Such a machine takes as input a program P written in $\mathcal{L}$, an input i for P, and some randomness r and outputs both the result o of evaluating P with i and r, and the leakage ℓ produced by the evaluation. We use the following notation for this operation $o \leftarrow \mathbb{M}(P, i; \mathsf{r})_{\rightsquigarrow \ell}$. We make the assumption that the machine is deterministic, so that all randomness required to execute programs is given by the input r. However, our security experiments are probabilistic, and we write $o \leftarrow\!\!\$\ \mathbb{M}(P, i)_{\rightsquigarrow \ell}$ to denote the probabilistic computation that first samples the random coins r that must be passed as randomness input of P, and then runs $\mathbb{M}(P, i; \mathsf{r})$. This approach agrees with the view that the problem of randomness generation is orthogonal to the one of secure implementation [16]. We discuss this further in Section 8.

We note that the definition of $\mathbb{M}$ makes three implicit assumptions. First, the semantics of a program must always be defined, since $\mathbb{M}$ always returns a result; termination issues can be resolved easily by aborting computations after a fixed number of steps. Second, our view of $\mathbb{M}$ does not allow an adversary to influence a program's execution other than through its queries. Finally, our model implies that the semantics of $\mathcal{L}$ can be equipped with meaningful notions of leakage. In the context of our use case, we adopt the common view of practical cryptography that timing leakage can be captured via the code-memory and data-memory accesses performed while executing a program. These can be sensibly formalized over assembly implementations, but not over higher-level implementations (e.g., over C implementations), not least because there is no guarantee that optimizing compilers do not introduce leakage. For this reason, in our case study, we consider the following two implementation models:

- a C-level model using a machine $\mathbb{M}_{\mathrm{C}}^{\emptyset}$ (or simply $\mathbb{M}_{\mathrm{C}}$) that animates the C language semantics with no leakage;
- an assembly-level model using a machine $\mathbb{M}_{\mathrm{x86}}^{\mathsf{CT}}$ that animates (a subset of) x86 assembly language, and produces leakage traces in the constant-time leakage model as detailed below.

In both languages, we adopt the semantic definitions as formalized in the CompCert certified compiler.

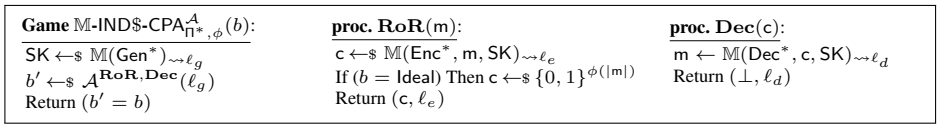

Fig. 4. $\mathbb{M}$-IND$-CPA experiment.

CONSTANT-TIME LEAKAGE TRACES. Formally, we capture the constant-time leakage model by assuming that each semantic step extends the (initially empty) leakage trace with a pair containing: i. the program point corresponding to the statement being executed; and ii. the (ordered) sequence of memory accesses performed during the execution step. We specify when this particular leakage model is used by annotating the corresponding notion with the symbol CT.

4.3 Authenticated encryption in the implementation model

Given a language $\mathcal{L}$ and a (potentially leaking) machine $\mathbb{M}$ animating its semantics, we now define $\mathbb{M}$-correctness, $\mathbb{M}$-IND$-CPA and $\mathbb{M}$-INT-PTXT security for $\mathcal{L}$-implementations of SKE schemes in the leakage model defined by $\mathbb{M}$. In what follows, we will let $\Pi^* = (\mathsf{Gen}^*, \mathsf{Enc}^*, \mathsf{Dec}^*)$ be an SKE implementation in language $\mathcal{L}$.

SKE IMPLEMENTATION CORRECTNESS. We say that Π^* is $\mathbb{M}$-correct if, for all $\mathsf{m} \in \mathsf{MsgSp}$, random coins $\mathsf{r}_{\mathsf{gen}}$, $\mathsf{r}_{\mathsf{enc}}$, and $\mathsf{SK} = \mathbb{M}(\mathsf{Gen}^*; \mathsf{r}_{\mathsf{gen}})$, we have that

$$\mathbb{M}(\mathsf{Dec}^*, \mathbb{M}(\mathsf{Enc}^*, \mathsf{m}, \mathsf{SK}; \mathsf{r}_{\mathsf{enc}}), \mathsf{SK}) = \mathsf{m}\,.$$

SKE IMPLEMENTATION SECURITY. The $\mathbb{M}$-IND$-CPA advantage of an adversary $\mathcal{A}$ against Π^* and public length function ϕ as the following (concrete) probability

$$\mathbf{Adv}^{\mathbb{M}\text{-ind\$-cpa}}_{\Pi^*,\phi,\mathcal{A}} := \Big|\mathbf{Pr}\left[\mathbb{M}\text{-IND\$-CPA}^{\mathcal{A}}_{\Pi^*,\phi}(\mathsf{Real}) \Rightarrow \mathsf{true}\right] - \mathbf{Pr}\left[\mathbb{M}\text{-IND\$-CPA}^{\mathcal{A}}_{\Pi^*,\phi}(\mathsf{Ideal}) \Rightarrow \mathsf{true}\right]\Big|,$$

where implementation-level game $\mathbb{M}$-IND$-CPA is shown in Figure 4. Here, public length function ϕ is used to capture the fact that SKEs may partially hide the length of a message. If ϕ is the identity function or is efficiently invertible, then the message length is trivially leaked by the ciphertext. In the case of our MEE-CBC specification, for example, the message length is revealed only up to AES block alignment.

We observe that in this refinement of the IND$-CPA security notion for implementations, the adversary may learn information about the secrets via the leakage produced by the decryption oracle Dec^*, even if its functional input-output behaviour reveals nothing. In particular, in a black-box adversary model where leakage traces are always empty, the Dec^* oracle can be perfeclty implemented by the procedure that ignores its argument and returns $(\perp, \epsilon)$, allowing us to recover the standard computational security experiment for IND$-CPA. On the other hand, in models where leakage traces are not always empty, the adversary is given the ability to use the decryption oracle with invalid ciphertexts and recover information through its leakage output.

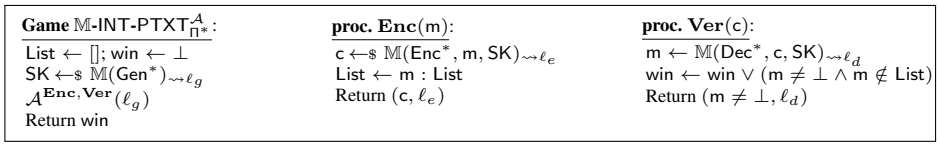

Fig. 5. $\mathbb{M}$-INT-PTXT experiment.

We extend the standard INT-PTXT security experiment in a similar way and define the $\mathbb{M}$-INT-PTXT advantage of an adversary $\mathcal{A}$ against Π^* as the following (concrete) probability:

$$\mathbf{Adv}^{\mathbb{M}\text{-int-ptxt}}_{\Pi^*,\mathcal{A}} := \mathbf{Pr}\left[\, \mathbb{M}\text{-INT-PTXT}^{\mathcal{A}}_{\Pi^*}() \Rightarrow \mathsf{true} \,\right],$$

where implementation-level game $\mathbb{M}$-INT-PTXT is shown in Figure 5.

We similarly "lift" INT-CTXT, PRP (pseudorandomness of a permutation) and UF-CMA (existential MAC unforgeability) security (as shown in Appendix D) experiments and advantages to implementations. This allows us to state our main theorem.

4.4 Main Theorem

The proof of Theorem 1 is fully machine-checked. However, foregoing machine-checking of the specification's security theorems allow us to strengthen the results we obtain on the final implementations. We discuss this further after we present our proof strategy.

Theorem 1 (CT security of MEE-CBC$_{\text{x86}}$). MEE-CBC$_{\text{x86}}$ *is* $\mathbb{M}^{\mathsf{CT}}_{\text{x86}}$*-correct and provides* $\mathbb{M}^{\mathsf{CT}}_{\text{x86}}$-IND$-CPA *and* $\mathbb{M}^{\mathsf{CT}}_{\text{x86}}$-INT-PTXT *security if the underlying components* AES128$_{\mathsf{NaCl}}$ *and* HMACSHA256$_{\mathsf{NaCl}}$ *are black-box secure as a PRP and a MAC, respectively. More precisely, let* $\phi(i) = \lceil (i+1)/16 \rceil + 3$*, then*

- *For any* $\mathbb{M}^{\mathsf{CT}}_{\text{x86}}$-IND$-CPA *adversary* $\mathcal{A}^{\mathsf{cpa}}$ *that makes at most* q *queries to its* $\mathbf{RoR}$ *oracle, each of length at most* n *octets, there exists an (explicitly constructed)* $\mathbb{M}^{\emptyset}_{\mathsf{C}}$-IND$-CPA *adversary* $\mathcal{B}^{\mathsf{prp}}$ *that makes at most* $\lceil (n+1)/16 \rceil + 2$ *queries to its forward oracle and such that*

$$\mathbf{Adv}^{\mathbb{M}^{\mathsf{CT}}_{\text{x86}}\text{-ind\$-cpa}}_{\mathsf{MEE\text{-}CBC}_{\text{x86}},\phi,\mathcal{A}^{\mathsf{cpa}}} \leq \mathbf{Adv}^{\mathbb{M}^{\emptyset}_{\mathsf{C}}\text{-prp}}_{\mathsf{AES128}_{\mathsf{NaCl}},\mathcal{B}^{\mathsf{prp}}} + 2 \cdot \frac{(q \cdot (\lceil \frac{n+1}{16} \rceil + 2))^2}{2^{128}}\,.$$

- *For any* $\mathbb{M}^{\mathsf{CT}}_{\text{x86}}$-INT-PTXT *adversary* $\mathcal{A}^{\mathsf{ptxt}}$ *that makes at most* q *queries to its* $\mathbf{Enc}$ *oracle and* q_V *queries to its* $\mathbf{Ver}$ *oracle, there exists an (explicitly constructed)* $\mathbb{M}^{\emptyset}_{\mathsf{C}}$-UF-CMA *adversary* $\mathcal{B}^{\mathsf{cma}}$ *that makes at most* q_E *queries to its* $\mathbf{Tag}$ *oracle and* q_V *queries to its* $\mathbf{Ver}$ *oracle and such that*

$$\mathbf{Adv}^{\mathbb{M}^{\mathsf{CT}}_{\text{x86}}\text{-int-ptxt}}_{\mathsf{MEE\text{-}CBC}_{\text{x86}},\mathcal{A}^{\mathsf{ptxt}}} \leq \mathbf{Adv}^{\mathbb{M}^{\emptyset}_{\mathsf{C}}\text{-uf-cma}}_{\mathsf{HMACSHA256}_{\mathsf{NaCl}},\mathcal{B}^{\mathsf{cma}}}\,.$$

In addition, the running time of our constructed adversaries is essentially that of running the original adversary plus the time it takes to emulate the leakage of the x86 implementations using dummy executions in machine $\mathbb{M}_{\text{x86}}$. Under reasonable assumptions on the efficiency of $\mathbb{M}_{\text{x86}}$, this will correspond to an overhead that is linear in

the combined inputs provided by an adversary to its oracles (the implementations are proven to run in constant time under the semantics of $\mathcal{L}$ when these inputs are fixed).

Note that the security assumptions we make are on C implementations of AES ($\mathsf{AES128}_{\mathsf{NaCl}}$) and HMAC-SHA256 ($\mathsf{HMACSHA256}_{\mathsf{NaCl}}$). More importantly, they are made in a black-box model of security where the adversary gets empty leakage traces.

The proof of Theorem 1 is detailed in Section 6 and relies on the general framework we now introduce. Rather than reasoning directly on the semantics of the executable x86 program (and placing our assumptions on objects that may not be amenable to inspection), we choose to prove complex security properties on a clear and simple functional specification, and show that each of the refinement steps on the way to an x86 assembly executable preserves this property, or even augments it in some way.

5 Formal framework and connection to PL techniques

Our formal proof of implementation security follows from a set of conditions on the software development process. We therefore introduce the notion of an implementation generation procedure.

IMPLEMENTATION GENERATION. An implementation generation procedure $\mathcal{C}^{\mathcal{L}_1 \to \mathcal{L}_2}$ is a mapping from specifications in language $\mathcal{L}_1$ to implementations in language $\mathcal{L}_2$. For example, in our use case, the top-level specification language is the expression language $\mathcal{L}_{\text{EC}}$ of EasyCrypt (a polymorphic and higher-order λ-calculus) and the overall implementation generation procedure $\mathcal{C}^{\mathcal{L}_{\text{EC}} \to \mathcal{L}_{\text{x86}}}$ is performed by a verified manual refinement of the specification into C followed by compilation to x86 assembly using CompCert (here, $\mathcal{L}_{\text{x86}}$ is the subset of x86 assembly supported by CompCert).

We now introduce two key notions for proving our main result: *correct implementation generation* and *leakage security*, which we relate to standard notions in the domain of programming language theory. This enables us to rely on existing formal verification methods and tools to derive intermediate results that are sufficient to prove our main theorem. In our definitions we consider two arbitrary languages $\mathcal{L}_1$ and $\mathcal{L}_2$, a (potentially leaking) machine $\mathbb{M}$ animating the semantics of the latter, and an implementation generation procedure $\mathcal{C}^{\mathcal{L}_1 \to \mathcal{L}_2}$. In this section, $\mathcal{L}_1$ and $\mathcal{L}_2$ are omitted when denoting the implementation generation procedure (simply writing $\mathcal{C}$ instead). In the rest of the paper, we also omit them when clear from context.

CORRECT IMPLEMENTATION GENERATION. Intuitively, the minimum requirement for an implementation generation procedure is that it preserves the input-output functionality of the specification. We capture this in the following definition.

Definition 1 (Correct implementation generation). *The implementation generation procedure $\mathcal{C}$ is correct if, for every adversary $\mathcal{A}$ and primitive specification Π, the game in Figure 6 always returns* true.

For the programming languages we are considering (deterministic, I/O-free languages) this notion of implementation generation correctness is equivalent to the standard language-based notion of simulation, and its specialization as semantic preservation when associated with general-purpose compilers. A notable case of this is CompCert [27] for which this property is formally proven in Coq. Similarly, as we discuss

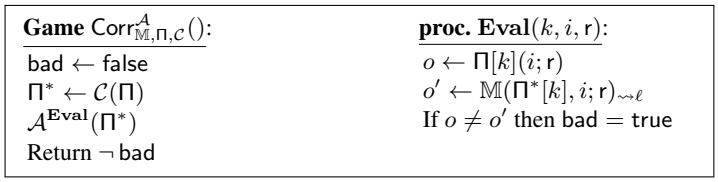

Fig. 6. Game defining correct implementation generation. For compactness, we use notation $\Pi[k]$ (resp. $\Pi^*[k]$) for $k \in \{1, 2, 3\}$ to denote the k-th algorithm in scheme Π (resp. implementation Π^*), corresponding to key generation (1), encryption (2) and decryption (3).

in Section 6, a manual refinement process can be turned into a correct implementation generation process by requiring a total functional correctness proof. This is sufficient to guarantee *black-box* implementation security. However, it is not sufficient in general to guarantee implementation security in the presence of leakage.

LEAKAGE SECURITY. In order to relate the security of implementations to that of black-box specifications, we establish that leakage does not depend on secret inputs. We capture this intuition via the notion of *leakage security*, which imposes that all the leakage produced by the machine $\mathbb{M}$ for an implementation is benign. Interestingly from the point of view of formal verification, leakage security is naturally related to the standard notion of non-interference [21]. In its simplest form, non-interference is formulated by partitioning the memory of a program into *high-security* (or secret) and *low-security* (or public) parts and stating that two executions that start in states that agree on their low-security partitions end in states that agree on their low-security partitions.

We define what the public part of the input means by specifying a function τ that parametrizes our definition of leakage security. For the case of symmetric encryption, for example, τ is defined to tag as public the inputs to the algorithms an attacker has control over through its various oracle interfaces (in IND$-CPA, INT-PTXT and INT-CTXT). More precisely, we define a specific projection function τ_{SKE} as follows:

$$\tau_{\mathsf{SKE}}(\mathsf{Gen}) = \epsilon \qquad \tau_{\mathsf{SKE}}(\mathsf{Enc}, \mathsf{key}, \mathsf{m}) = (|\mathsf{key}|, |\mathsf{m}|) \qquad \tau_{\mathsf{SKE}}(\mathsf{Dec}, \mathsf{key}, \mathsf{c}) = (|\mathsf{key}|, \mathsf{c})$$

Our definition of leakage security then consists in constraining the information-flow into the leakage due to each algorithm, via the following non-interference notion.[11]

Definition 2 **($(\mathbb{M}, \tau)$-non-interference).** *Let P be a program in $\mathcal{L}_2$ and τ be a projection function on P's inputs. Then, P is $(\mathbb{M}, \tau)$-non-interferent if, for any two executions $o \leftarrow \mathbb{M}(P, i; \mathsf{r})_{\rightsquigarrow \ell}$ and $o' \leftarrow \mathbb{M}(P, i'; \mathsf{r}')_{\rightsquigarrow \ell'}$, we have $\tau(P, i) = \tau(P, i') \;\Rightarrow\; \ell = \ell'$.*

Intuitively, $(\mathbb{M}, \tau)$-non-interference labels the leakage ℓ as a public output (which must be proved independent of secret information), whereas τ is used to specify which inputs of P are considered public. By extension, those inputs that are *not* revealed by τ are considered secret, and are not constrained in any way during either executions. Note that the leakage produced by a $(\mathbb{M}, \tau)$-non-interferent program for some input i can be predicted given only the public information revealed by $\tau(P, i)$: one can simply choose the remaining part of the input arbitrarily, constructing some input i' such that

[11] For simplicity, the length of random input is assumed to be given by the implementation itself.

$\tau(P, i) = \tau(P, i')$. In this case, $(\mathbb{M}, \tau)$-non-interference guarantees that the leakage traces produced by $\mathbb{M}$ when executing P on i and i' are equal.

We can now specialize this notion of leakage security to symmetric encryption.

Definition 3 (Leakage-secure implementation generation for SKE). *An implementation generation procedure $\mathcal{C}$ produces $\mathbb{M}$-leakage-secure implementations for SKE if, for all SKE specifications Π written in $\mathcal{L}_1$, we have that the generated $\mathcal{L}_2$ implementation $(\mathsf{Gen}^*, \mathsf{Enc}^*, \mathsf{Dec}^*) = \mathcal{C}(\Pi)$ is $(\mathbb{M}, \tau_{\mathsf{SKE}})$-non-interferent.*

PUTTING THE PIECES TOGETHER. The following lemma, shows that applying a correct and leakage secure implementation generation procedure to a black-box secure SKE specification is sufficient to guarantee implementation security.

Theorem 2. *Let $\mathcal{C}$ be correct and produce $\mathbb{M}$-leakage-secure implementations. Then, for all SKE scheme Π that is correct,* IND$-CPA-, INT-PTXT- *and* INT-CTXT-*secure, the implementation $\Pi^* = \mathcal{C}(\Pi)$ is $\mathbb{M}$-correct, $\mathbb{M}$-*IND$-CPA-, $\mathbb{M}$-INT-PTXT- *and* $\mathbb{M}$-INT-CTXT-*secure with the same advantages.*

Proof. Correctness of Π^* follows directly from that of $\mathcal{C}$ and Π. The security proofs are direct reductions. We only detail the proof of $\mathbb{M}$-IND$-CPA, but note that a similar proof can be constructed for $\mathbb{M}$-INT-PTXT and $\mathbb{M}$-INT-CTXT. Given an implementation adversary $\mathcal{A}$, we construct an adversary $\mathcal{B}$ against Π as follows. Adversary $\mathcal{B}$ runs Gen^* on an arbitrary randomness of appropriate size to obtain the leakage ℓ_{Gen} associated with key generation and runs adversary $\mathcal{A}$ on ℓ_{Gen}. Oracle queries made by $\mathcal{A}$ are simulated by using $\mathcal{B}$'s specification oracles to obtain outputs, and the same leakage simulation strategy to present a perfect view of the implementation leakage to $\mathcal{A}$. When $\mathcal{A}$ outputs its guess, $\mathcal{B}$ forwards it as its own guess. We now argue that $\mathcal{B}$'s simulation is perfect. The first part of the argument relies on the correctness of the implementation generation procedure, which guarantees that the values obtained by $\mathcal{B}$ from its oracles in the CPA-game are identically distributed to those that $\mathcal{A}$ would have received in the implementation game. The second part of the argument relies on the fact that leakage-secure implementation generation guarantees that $\mathcal{B}$ knows enough about the (unknown) inputs to the black-box algorithms (the information specified by τ_{SKE}) to predict the exact leakage that such inputs would produce in the implementation model. Observe for example that, in the case of decryption leakage, the adversary $\mathcal{B}$ only needs the input ciphertext c to be able to exactly reproduce the leakage ℓ_{Dec}. Finally, note that the running time of the constructed adversary $\mathcal{B}$ is that of adversary $\mathcal{A}$ where each oracle query $\mathcal{A}$ introduces an overhead of one execution of the implementation in machine $\mathbb{M}$ (which can reasonably be assumed to be close to that of the specification). □

6 Implementation security of MEE-CBC

We now return to our case study, and explain how to use the methodology from Section 5, instantiated with existing verification and compilation tools, to derive assembly-level correctness and security properties for $\mathsf{MEE\text{-}CBC}_{\mathrm{x86}}$.

PROOF STRATEGY. We first go briefly over each of the steps in our proof strategy, and then detail each of them in turn in the remainder of this section. In the first step, we specify and verify the correctness and black-box computational security of the MEE-CBC construction using EasyCrypt. We include the specification in Appendix C, Figure 11. In a second step, we use Frama-C to prove the functional correctness of program $\mathsf{MEE\text{-}CBC}_{\mathsf{C}}$ with respect to the EasyCrypt specification. Finally, we focus on the x86 assembly code generated by CompCert ($\mathsf{MEE\text{-}CBC}_{\mathrm{x86}}$), and prove: i. its functional correctness with respect to the C code (and thus the top-level EasyCrypt specification); and ii. its leakage security. An instantiation of Theorem 2 allow us to conclude the proof of Theorem 1.

BLACK-BOX SPECIFICATION SECURITY. We use EasyCrypt to prove that the MEE-CBC construction provides IND$-CPA security (when used with freshly and uniformly sampled IVs for each query) and INT-PTXT security.

Lemma 1 (Machine-chekced MEE-CBC security). *The following two results hold:*

- *For all legitimate* IND$-CPA *adversary* $\mathcal{A}^{\mathsf{cpa}}$ *that makes at most* q *queries, each of length at most* n *octets, to its* $\mathbf{RoR}$ *oracle, there exists an explicitly constructed PRP adversary* $\mathcal{B}^{\mathsf{prp}}$ *that makes* $\lceil (n+1) \,/\, \lambda \rceil + 2$ *queries to its forward oracle and such that:*

$$|\mathbf{Adv}^{\mathsf{ind\$\text{-}cpa}}_{\Pi,\phi,\mathcal{A}}| \leq 2 \cdot (|\mathbf{Adv}^{\mathsf{prp}}_{\mathsf{Perm}}(\mathcal{B}^{\mathsf{prp}})| + 2 \cdot \frac{(q \cdot \lceil \frac{n+1}{\lambda} \rceil + 2)^2}{2^{8 \cdot \lambda}}),$$

  where $\phi(i) = \lceil (i+1) \,/\, \lambda \rceil + 3$ *reveals only the number of blocks in the plaintext (and adds to it the fixed number of blocks due to IV and MAC tag).*
- *For all* PTXT *adversary* $\mathcal{A}$ *that makes* q_V *queries to its* $\mathbf{Dec}$ *oracle, there exists an explicitly constructed* SUF-CMA *adversary* $\mathcal{B}^{\mathsf{cma}}$ *that makes exactly* q_V *queries to its* $\mathbf{Ver}$ *oracle and such that:*

$$|\mathbf{Adv}^{\mathsf{int\text{-}ptxt}}_{\Pi,\mathcal{A}}| \leq |\mathbf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathsf{Mac}}(\mathcal{B}^{\mathsf{cma}})|.$$

Our EasyCrypt specification (Figure 11) relies on abstract algorithms for the primitives. More precisely, it is parameterized by an abstract, stateless and deterministic block cipher Perm with block size λ octets, and by an abstract, stateless and deterministic MAC scheme Mac producing tags of length $2 \cdot \lambda$.[12] The proofs, formalized in EasyCrypt, are fairly standard and account for all details of padding and message formatting in order to obtain the weak length-hiding property shown in this lemma. Running times for $\mathcal{B}^{\mathsf{prp}}$ and $\mathcal{B}^{\mathsf{cma}}$ are as usual.

We note that, although we have not formalized in EasyCrypt the proof of INT-CTXT security (this would imply a significant increase in interactive theorem proving effort) the known security results for MEE-CBC also apply to this specification and, in particular, it follows from [29] that it also achieves this stronger level of security when the underlying MAC and cipher satisfy slightly stronger security requirements.

IMPLEMENTATION GENERATION. Using Frama-C, a verification platform for C programs,[13] we prove functional equivalence between the EasyCrypt specification and our

[12] This is only for convenience in these definitions.

[13] `http://frama-c.com/`

C implementation. Specifically, we use the deductive verification (WP) plugin to check that our C code fully and faithfully implements a functionality described in the ANSI/ISO C Specification Language (ACSL). To make sure that the ACSL specification precisely corresponds to the EasyCrypt specification on which black-box security is formally proved, we rely on Frama-C's ability to link ACSL logical constructs at the C annotation level to specific operators in underlying Why3 theories, which we formally relate to those used in the EasyCrypt proof. This closes the gap between the tools allowing us to refer to a common specification. Note that, since the abstract block cipher Perm and MAC scheme Mac are concretely instantiated in the C implementation, we instantiate $\lambda = 128$ (the AES block length) in this common specification and lift the assumptions on Perm and Mac to the C implementation of their chosen instantiation. Figure 8 illustrates the format of contracts and general shape of obligations we verify in Frama-C. The challenges faced during this step of the proof are discussed in Appendix A. We then use the CompCert certified compiler [27] to produce our final x86 assembly implementation.

To prove leakage security, we use the certifying information-flow type system for x86 built on top of CompCert [5], marking as public those inputs that correspond to values revealed by τ_{SKE}. Obtaining this proof does not put any additional burden on the user—except for marking program inputs as secret or public. However, the original C code must satisfy a number of restrictions in order to be analyzed using the dataflow analysis from [5]. Our C implementations were constructed to meet these restrictions, and lifting them to permit a wider applicability of our techniques is an important challenge for further work.[14]

PROOF OF THEOREM 1. Let us denote by $\mathcal{C}^{\mathcal{L}_{\mathsf{EC}} \to \mathrm{x86}}$ the implementation generation procedure that consists of hand-crafting a C implementation (annotated with τ_{SKE} consistent security types), equivalence-checking it with an EasyCrypt specification using Frama-C, and then compiling it to assembly using CompCert (accepting only assembly implementations that type-check under the embedded secure information-flow type system), as we have done for our use case. We formalize the guarantees provided by this procedure in the following lemma.

Lemma 2 (Implementation generation). *$\mathcal{C}^{\mathcal{L}_{\mathsf{EC}} \to \mathrm{x86}}$ is a $\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}$-correct implementation generation procedure that produces $\mathbb{M}^{\mathsf{CT}}_{\mathrm{x86}}$-leakage secure SKE implementations.*

Proof. Correctness follows from the combination of the Frama-C functional correctness proof and the semantic preservation guarantees provided by CompCert. CompCert's semantics preservation theorem implies that the input/output behaviour of the assembly program exactly matches that of the C program. Functional equivalence checking using Frama-C yields that the C implementation has an input/output behaviour that is consistent with that of the EasyCrypt specification (under the C semantics adopted

[14] In a recent development in this direction, Almeida et al. [4] describe a method, based on limited product programs, for verifying constant-time properties of LLVM code. Their method and the implementation they describe can deal with many examples that the type system from [5] cannot handle, including some of the OpenSSL code for MEE-CBC, whilst preserving a high degree of automation. In addition, their construction easily extends to situations where declassification is needed.

#	Implementation	Compiler	Clock Cycles	Time
1	S2N	GCC x86-64 -O2	14K	5μs
2	OpenSSL	GCC x86-64 -O2	23K	9μs
3	MEE-CBC$_{\mathsf{C}}$ (AES-NI)	CompCert x86-32	51K	21μs
4	MEE-CBC$_{\mathsf{C}}$	GCC x86-64 -O2	59M	25ms
5	MEE-CBC$_{\mathsf{C}}$	GCC x86-64 -O1	62M	26ms
6	MEE-CBC$_{\mathsf{x86}}$	CompCert x86-32	101M	42ms
7	MEE-CBC$_{\mathsf{C}}$	GCC x86-64 -O0	237M	99ms

Fig. 7. Performance comparison of various MEE-CBC implementations. (Median over 500 runs.)

by Frama-C). Finally, under the reasonable assumption that the CompCert semantics of C are a sound refinement of those used in Frama-C, we obtain functional correctness of the assembly implementation with respect to the EasyCrypt specification. For leakage security, we rely on the fact that the information-flow type system of [5] enforces τ_{SKE}-non-interference and therefore only accepts $(\mathbb{M}^{\mathsf{CT}}_{\mathsf{x86}}, \tau_{\mathsf{SKE}})$-leakage secure implementations. □

Theorem 1 follows immediately from the application of Theorem 2 instantiated with Lemmas 1 and 2. Furthermore, foregoing machine-checking the black-box specification security proof and simply accepting known results on MEE-TLS-CBC [29], we can also show that MEE-CBC$_{\mathsf{x86}}$ is $\mathbb{M}^{\mathsf{CT}}_{\mathsf{x86}}$INT-CTXT-secure under slightly stronger black-box assumptions on AES128$_{\mathsf{NaCl}}$ and HMACSHA256$_{\mathsf{NaCl}}$.

7 Performance Comparison

We now characterize the different assurance/performance trade-offs of the timing mitigation strategies discussed in this paper. Figure 7 shows the time taken by 5 different implementations of MEE-CBC (one of them compiled in three different ways) when decrypting a 1.5KB TLS1.2 record using the AES128-SHA256 ciphersuite.[15] More specifically, we consider code from S2N (#1) and OpenSSL (#2), and three different compilations of our formally verified MEE-CBC implementation (#3-7), focusing on raw invocations of MEE-CBC.

It is clear that the S2N code (#1) benefits from its less strict timing countermeasures, gaining roughly 1.8x performance over OpenSSL's (semi-)constant-time implementation approach (#2). The figures for our verified implementation of MEE-CBC show both the cost of formal verification and the cost of full constant-time guarantees. Indeed, the least efficient results are obtained when imposing full code and data memory access independence from secret data (#4-6).

The assembly implementation produced using the constant-time version of CompCert (#6), is roughly 8400x slower than S2N, but still over twice as fast as unoptimized GCC. However, the fact that the same C code compile with `GCC -O2` (#4) is only 1.7x faster than the fully verified CompCert-generated code shows that the bottleneck does not reside in verification, but in the constant-time countermeasures. Profiling reveals that NaCl's constant-time AES accounts for 97% of the execution time. These results

[15] The numbers were obtained in a virtualized Intel x86-64 Linux machine with 4 GB RAM.

confirm the observations in [14] as to the difficulties of reconciling resistance against cache attacks and efficiency in AES implementations. To further illustrate that point, we also include measurements corresponding to a modification of our MEE-CBC implementation that uses hardware-backed AES (#3). This cannot, in fairness, be compared to the other implementations, but it does demonstrate that, with current verification technology, the performance cost of a fully verified constant-time MEE-CBC implementation is not prohibitive.

8 Discussion

ON RANDOMNESS. Restricting our study to deterministic programs with an argument containing random coins does not exclude the analysis of real-world systems. There, randomness is typically scarce and pseudorandom generators are used to expand short raw high-entropy bitstrings into larger random-looking strings that are fed to deterministic algorithms, and it is common to assume that the small original seed comes from an ideal randomness source, as is done in this paper. Our approach could therefore be used to analyze the entire pseudorandom generation implementation, including potential leakage-related vulnerabilities therein.

ON LENGTH-HIDING SECURITY. We note that existing implementations of MEE-TLS-CBC (and indeed our own implementation of MEE-CBC) are not length-hiding as defined in [29] in the presence of leakage. Indeed, the constant-time countermeasures are only applied in the decryption oracle and precise information about plaintext lengths may be leaked during the execution of the encryption oracle. Carrying length-hiding properties down to the level of those implementations may therefore require them to be modified (and the Frama-C equivalence proof adapted accordingly). However, Lemma 1 does capture the length-hiding property given by our choice of minimal padding.

LEAKAGE SIMULATION AND WEAKER NON-INTERFERENCE NOTIONS. Our use of leakage security in proving that leakage is not useful to an adversary naturally generalizes to a notion of *leakage simulation*, whereby an implementation is secure as long as its leakage can be efficiently and perfectly simulated from its public I/O behaviour, including its public *outputs*. For example, an implementation of Encrypt-then-MAC that aborts as soon as MAC verification fails, but is otherwise fully constant-time should naturally be considered secure, since the information gained through the leakage traces is less than that gained by observing the output of the $\mathbf{Ver}$ oracle. The more general notion of leakage simulation informally described here would capture this and can be related to weaker notions of non-interference, where equality on low outputs is only required on those traces that agree on the value of some *declassified* outputs. Theorem 2 can be easily modified to replace leakage security with the (potentially weaker) leakage simulation hypothesis.

9 Further related work

There is a large body of work that applies formal tools for reasoning about cryptographic implementations. However, we are not aware of any other work that combines

relatively abstract and spearate notions of functional correctness, leakage security, and black-box security for achieving provable security in presence of implementation leakage.

The work that stands closest to ours is the provable security analysis of an assembly implementation of PKCS in the program counter model [3]. Their analysis uses 3 ingredients: i. security of a C implementation in the program counter model, using EasyCrypt; ii. functional correctness of assembly, using CompCert; and iii. an assembly-level static analysis that justifies that the compiler did not introduce additional leakage.

Leakage of assembly-level implementations is also considered in [5]. They propose a formally verified, fine-grained information-flow analysis for checking that assembly programs generated by CompCert execute in constant-time (which we use here). Moreover, they prove that constant-time programs are leakage-secure in a stronger adversary model than the one we consider, in which the adversary is co-located on a virtualization platform with the victim program, and controls the scheduler and the cache. This proof brings significant strength to the guarantees delivered by constant-time cryptography, independently of the method used to enforce it. However, they do not establish any connection between leakage security and provable security.

Another relevant work is [12], which shows functional correctness of an HMAC implementation and provable security (based on the earliest proof of HMAC); their proof of functional correctness is split in two parts: they use the VST program logic to show functional correctness of a C implementation with respect to a functional specification, and they derive implementation correctness using CompCert. However, they do not consider leakage and do not prove any result for carrying the security proof to the assembly implementation.

One popular approach for reasoning about the security of cryptographic implementations is to use standard program verification—typically at source code level. This approach has been used to carry out a security analysis of miTLS [17], an implementation of TLS written in the F# language. The CVJ framework [25] (Cryptographic Verification of Java-like programs) is another instance of approach based on program verification; the approach reduces security of an implementation to provable security of the underlying primitives and information flow properties of implementation. A similar approach [9] uses the KeY prover for proving information-flow properties. While source code implementations are more precise than algorithmic descriptions used in security proofs, approaches based on source verification cannot precisely model side-channels. In addition, these methods are based on general-purpose program verification tools that do not support reasoning about probabilistic computations; therefore, the validity of such approaches relies on a soundness proof, which is carried out with pen and paper.

10 Conclusions and directions for future work

Our proposed methodology allows deriving guarantees on assembly implementations from more focused and tractable verification tasks. Each of these more specialized tasks additionally carries its own challenges.

Proving security in lower-level leakage models for assembly involves considering architectural details such as memory management, scheduling and data-dependent leakage sources. Automatically relating source and *existing* assembly implementations requires developing innovative methods for checking (possibly conditional or approximate) equivalence between low-level probabilistic programs. Finally, obtaining formal proofs of computational security and functional correctness in general both remain important bottlenecks, requiring high levels of both expertise and effort. However, combining formal and generic composition principles (such as those used in our case study) with techniques that automate these two tasks for restricted application domains [6,22,13] should enable the formal verification of extensive cryptographic libraries, in the presence of leakage. We believe that this goal is now within reach.

On the cryptographic side, the study of computational security notions that allow the adversary to tamper with the oracle implementation [11] may lead to relaxed functional correctness requirements that may be easier to check. Extensions of our framework to settings where the adversary has the ability to tamper with the execution of the oracle are possible, and would allow it to capture recent formal treatments of countermeasures against fault injection attacks [30].

References

1. Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on amazon's s2n implementation of tls. Cryptology ePrint Archive, Report 2015/1129, 2015. `http://eprint.iacr.org/`.
2. Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy, SP 2013*, pages 526–540. IEEE Computer Society, 2013.
3. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 1217–1230. ACM Press, November 2013.
4. José Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. Manuscript, 2015. `https://fdupress.net/files/ctverif.pdf`.
5. Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 1267–1279. ACM Press, November 2014.
6. Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, Yassine Lakhnech, Benedikt Schmidt, and Santiago Zanella Béguelin. Fully automated analysis of padding-based encryption in the computational model. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 1247–1260. ACM Press, November 2013.
7. Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2014.
8. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, Heidelberg, August 2011.

9. Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Information flow in object-oriented software. In Gopal Gupta and Ricardo Peña, editors, *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2013.
10. Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 531–545. Springer, Heidelberg, December 2000.
11. Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 1–19. Springer, Heidelberg, August 2014.
12. Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium*, August 2015.
13. Dan Bernstein and Peter Schwabe. Cryptographic software, side channels, and verification. COST CryptoAction WG3 Meeting, April 2015.
14. Daniel J. Bernstein. Aes timing variability at a glance. `http://cr.yp.to/mac/variability1.html`, Accessed October 25th, 2015.
15. Daniel J. Bernstein. Cache-timing attacks on AES, 2005. Available from author's webpage.
16. Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 159–176. Springer, Heidelberg, October 2012.
17. K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *Symposium on Security and Privacy, S&P 2013*. IEEE, 2013.
18. Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. On symmetric encryption with distinguishable decryption failures. In *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, pages 367–390, 2013.
19. Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 583–599. Springer, Heidelberg, August 2003.
20. J.-P. Degabriele, K.G. Paterson, and G.J. Watson. Provable security in the real world. *IEEE Security & Privacy*, 9(3), 2011.
21. Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.
22. Viet Tung Hoang, Jonathan Katz, and Alex J. Malozemoff. Automated analysis and synthesis of authenticated encryption schemes. Cryptology ePrint Archive, Report 2015/624, 2015. `http://eprint.iacr.org/2015/624`.
23. Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.
24. Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 310–331. Springer, Heidelberg, August 2001.
25. R. Küsters, T. Truderung, and J. Graf. A framework for the cryptographic verification of java-like programs. In *Computer Security Foundations Symposium, CSF 2012*. IEEE, 2012.

26. Adam Langley. Lucky thirteen attack on TLS CBC. Imperial Violet, https://www.imperialviolet.org/2013/02/04/luckythirteen.html, February 2013. Accessed October 25th, 2015.
27. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Symposium on Principles of Programming Languages, POPL 2006*. ACM, 2006.
28. Ueli Maurer and Björn Tackmann. On the soundness of authenticate-then-encrypt: formalizing the malleability of symmetric encryption. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10*, pages 505–515. ACM Press, October 2010.
29. Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 372–389. Springer, Heidelberg, December 2011.
30. Pablo Rauzy and Sylvain Guilley. A formal proof of countermeasures against fault injection attacks on CRT-RSA. *J. Cryptographic Engineering*, 4(3):173–185, 2014.
31. Stephen Schmidt. Introducing s2n, a new open source tls implementation. https://blogs.aws.amazon.com/security/post/TxCKZM94ST1S6Y/Introducing-s2n-a-New-Open-Source-TLS-Implementation, June 2015. Accessed October 25th, 2015.
32. Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 534–546. Springer, Heidelberg, April / May 2002.

A C implementation with ACSL annotations

The unpad function shown in Figure 8 is a good example of classic constant-time implementation techniques: one ensures that all loops have a number of iterations that depends only on public data, and that no secret-dependent branching is used. When implementing non straight-line code, all conditional branches that depend on secrets are replaced with code that executes all branches and selects (in constant-time, using bitmasks) the correct result. In this case, the loop is always iterating for the maximum number of padding bytes, and constructs a mask that accumulates the result of the padding check only for those positions that are part of the padding.

```
/*@
 requires len%16 == 0;
 requires \valid(out_len) && \valid_read(in+(0..len-1));
 assigns *out_len;
 behavior failure:
  assumes unpad(mem_blocks(in,len/16)) == none;
  ensures \result == 0;
 behavior success:
  assumes is_some(unpad(mem_blocks(in,len/16)));
  ensures \result == 1;
  ensures unpad(mem_blocks(in,len/16))
    == some(mem_bytes(in,*out_len/16),mem_tag(in+*out_len));
 complete behaviors;   disjoint behaviors;
*/
int crypto_pad_remove(unsigned long *out_len,
                      const unsigned char *in,
                      unsigned long len)
{
 unsigned padding_length, good, i;
 if (len < 3*OUTPUTBYTES) return 0;
 padding_length = in[len - 1];
 good = constant_time_ge(OUTPUTBYTES, padding_length);
 good &= constant_time_lt(0, padding_length);
 for (i = 0; i < OUTPUTBYTES; i++) {
  unsigned char mask = constant_time_lt(i,padding_length);
  good &= ~(mask & (padding_length ^ in[len - 1 - i]));
 }
 good = constant_time_eq(0xff, good & 0xff);
 padding_length = good & (2*OUTPUTBYTES+padding_length);
 *out_len = len - padding_length;
 return constant_time_select(good, 1, 0);
}
```

Fig. 8. Unpad operation in constant time (with ACSL contract).

B Snippets from the S2N implementation

The code snippets in Figures 9 and 10 show the code for functions `s2n_verify_cbc` (from file `tls/s2n_cbc.c`) and `s2n_hmac_digest` (from file `crypto/s2n_hmac.c`) that contain the implementation bug we have reported.

C EasyCrypt specification of MEE-CBC

Figure 11 shows the formal specification of the construction, parameterized by: i. a family of permutations $\mathsf{P} \in \mathcal{K}_e \to \mathcal{O}^\lambda \to \mathcal{O}^\lambda$, where $\mathcal{O} = \{0,1\}^8$ is the type of octets,

```
1  int s2n_verify_cbc (struct s2n_connection *conn, struct s2n_hmac_state *hmac, struct s2n_blob *decrypted)
2  {
3      struct s2n_hmac_state copy;
4      int mac_digest_size = s2n_hmac_digest_size (hmac→ alg);
5
6      /* The record has to be at least big enough to contain the MAC, plus the padding length byte */
7      gt_check(decrypted→ size, mac_digest_size);
8
9      int payload_and_padding_size = decrypted→ size − mac_digest_size;
10
11     /* Determine what the padding length is */
12     uint8_t padding_length = decrypted→ data[decrypted→ size − 1];
13
14     int payload_length = payload_and_padding_size − padding_length − 1;
15     if (payload_length < 0) {
16         payload_length = 0;
17     }
18
19     /* Update the MAC */
20     GUARD(s2n_hmac_update(hmac, decrypted→ data, payload_length));
21     GUARD(s2n_hmac_copy(&copy, hmac));
22
23     /* Check the MAC */
24     uint8_t check_digest [S2N_MAX_DIGEST_LEN];
25     lte_check (mac_digest_size , sizeof (check_digest ));
26     GUARD(s2n_hmac_digest_two_compression_rounds(hmac, check_digest, mac_digest_size ));
27
28     int mismatches = s2n_constant_time_equals (decrypted→ data + payload_length, check_digest , mac_digest_size ) ^ 1;
29
30     /* Compute a MAC on the rest of the data so that we perform the same number of hash operations */
31     GUARD(s2n_hmac_update(&copy, decrypted→ data + payload_length + mac_digest_size,
32                 decrypted→ size − payload_length − mac_digest_size − 1));
33
34     /* SSLv3 doesn't specify what the padding should actualy be */
35     if (conn→ actual_protocol_version == S2N_SSLv3) {
36         return 0 − mismatches;
37     }
38
39     /* Check the padding */
40     int check = 255;
41     if (check > payload_and_padding_size) {
42         check = payload_and_padding_size ;
43     }
44
45     int cutoff = check − padding_length;
46     for (int i = 0, j = decrypted→ size − check; i < check && j < decrypted→ size; i++, j++) {
47         uint8_t mask = ~(0xff << ((i ≥ cutoff) * 8));
48         mismatches |= (decrypted→ data[j] ^ padding_length ) & mask;
49     }
50
51     if (mismatches) {
52         S2N_ERROR(S2N_ERR_CBC_VERIFY);
53     }
54
55     return 0;
56 }
```

Fig. 9. The MEE-CBC verification code in S2N.

```
1  int s2n_hmac_digest(struct s2n_hmac_state *state, void *out, uint32_t size)
2  {
3      if (state->alg == S2N_HMAC_SSLv3_SHA1 || state->alg == S2N_HMAC_SSLv3_MD5) {
4          return s2n_sslv3_mac_digest(state, out, size);
5      }
6
7      GUARD(s2n_hash_digest(&state->inner, state->digest_pad, state->digest_size));
8      GUARD(s2n_hash_reset(&state->outer));
9      GUARD(s2n_hash_update(&state->outer, state->xor_pad, state->block_size));
10     GUARD(s2n_hash_update(&state->outer, state->digest_pad, state->digest_size));
11
12     return s2n_hash_digest(&state->outer, out, size);
13 }
14
15 int s2n_hmac_digest_two_compression_rounds(struct s2n_hmac_state *state, void *out, uint32_t size)
16 {
17     GUARD(s2n_hmac_digest(state, out, size));
18
19     /* If there were 8 or more bytes of space left in the current hash block
20      * then the serialized length will have fit in that block. If there were
21      * fewer than 8 then adding the length will have caused an extra compression
22      * block round. This digest function always does two compression rounds,
23      * even if there is no need for the second.
24      */
25     if (state->currently_in_hash_block > (state->hash_block_size - 8))
26     {
27         return 0;
28     }
29
30     return s2n_hash_update(&state->inner, state->xor_pad, state->hash_block_size);
31 }
```

Fig. 10. The HMAC finalization funtion in S2N.

$\mathcal{K}_e \subseteq \mathcal{O}^*$ is the type of encryption keys, and λ is the block length, which we identify with the security parameter; and ii. a family of functions $\mathsf{M} \in \mathcal{K}_m \to \mathcal{O}^* \to \mathcal{O}^{2\cdot\lambda}$, where $\mathcal{K}_m \subseteq \mathcal{O}^*$ is the type of MAC keys.[16]

We use to_τ to denote conversion to type τ (for example from lists of blocks to lists of octets, or from octets to unsigned integers), and use standard list operations to manipulate lists:

- [] denotes the empty list;
- :: denotes the construction of list by prepending a single element to an existing list;
- create x l constructs the list that contains l copies of elements x;
- ++ concatenates two lists;
- take l xs denotes the truncated list containing only the first l elements of list xs;
- drop l xs denotes the truncated list containing only the last size xs − l elements of list xs; and
- last xs is the last element of list xs.

We write $\oplus$ to denote the bitwise exclusive-or of two blocks (in $\mathcal{O}^\lambda$).

We let $\mathcal{P} = (\mathsf{Gen}_e, \mathsf{P}, \mathsf{P}^{-1})$ be a PRP on $\mathcal{O}^\lambda$ indexed by $\mathcal{K}_e$ (such that P and P^{-1} are stateless and deterministic), and $\mathsf{Mac} = (\mathsf{Gen}_m, \mathsf{M})$ be a (stateless and deterministic) MAC scheme on $\mathcal{O}^*$, with tags in $\mathcal{O}^{2\cdot\lambda}$ and indexed by $\mathcal{K}_m$. Let $\Pi = (\mathsf{Gen}_e \times \mathsf{Gen}_m, \mathsf{MEE}^{\mathsf{P},\mathsf{M}}_{\$\text{-}\mathsf{enc}}, \mathsf{MEE}^{\mathsf{P}^{-1},\mathsf{M}}_{\$\text{-}\mathsf{dec}})$ be the secret key encryption scheme that samples a fresh and uniform random IV for each encryption query and prepends it to the ciphertext, and uses the first block of ciphertext as IV on decryption queries.

D Security models for specification security

Definition 4 (Secrecy). *We define the advantage of an adversary $\mathcal{A}$ against the* IND$-CPA *security of SKE scheme Π as*

$$\mathbf{Adv}^{\mathsf{ind\$\text{-}cpa}}_{\Pi,\mathcal{A}} := 2 \cdot \boldsymbol{Pr}\left[\mathsf{IND\$\text{-}CPA}^{\mathcal{A}}_{\Pi}() \Rightarrow \mathsf{true}\right] - 1\,,$$

where game IND$-CPA *is shown in Figure 12.*

Note that we give the adversary access to a dummy decryption oracle that performs the decryption but always returns $\perp$. This formulation and the more standard one where the adversary does not have access to the decryption oracle are equivalent in the (black-box) computational model, but importantly not in the presence of leakage, as we have seen in Section 4.

Definition 5 (Plaintext integrity). *We define the advantage of an adversary $\mathcal{A}$ against the* INT-PTXT *security of SKE scheme Π as*

$$\mathbf{Adv}^{\mathsf{int\text{-}ptxt}}_{\Pi,\mathcal{A}} := \boldsymbol{Pr}\left[\mathsf{INT\text{-}PTXT}^{\mathcal{A}}_{\Pi}() \Rightarrow \mathsf{true}\right],$$

where game INT-PTXT *is shown in Figure 13.*

```
(** Encryption **)
op CBC^P_enc (k ∈ K_e) (iv ∈ O^λ) (p ∈ O^λ*) =
  with p = []        ⇒ []
  with p = pi :: p ⇒ let ci = P k (st ⊕ pi) in
                      ci :: (CBC^P_enc k ci p).

op pad (m ∈ O*) (t ∈ O^{2·λ}) =
  let l = λ − size m % λ in
  let p = create (to_O l) l in
  to_{Oλ*} (m ++ to_{O*} t ++ p).

let MEE^{P,M}_enc ((ek,mk) ∈ K_e × K_m) (iv ∈ O^λ) (m ∈ O*) =
  let t = M mk m in
  let p = pad m t in
  CBC^P_enc ek iv p.

(** Decryption **)
op CBC^{P^-1}_dec (k ∈ K_e) (iv ∈ O^λ) (c ∈ O^λ*) =
  with c = []        ⇒ []
  with c = ci :: c ⇒ let pi = Pi k ci in
                      (pi ⊕ st) :: (CBC^{P^-1}_dec k ci c).

op unpad (bs ∈ O^λ*) =
  let os = to_{Oλ*} bs in
  let l = to_ℕ (last os) in
  let m = take (size os − l − 2 · λ) os in
  let t = to_{O2·λ} (take (2 · λ) (drop (size os − l − 2 · λ) os)) in
  let p = drop (size os − l) os in
  if (3 ≤ size bs ∧ 1 ≤ l ≤ λ ∧ p = create (to_O l) l)
  then (m,t)
  else ⊥.

op MEE^{P^-1,M}_dec ((ek,mk) ∈ K_e × K_m) (iv ∈ O^λ) (c ∈ O^λ*) =
  let p = CBC^{P^-1}_dec ek iv c in
  let (m,t)_⊥ = unpad p in
  if ((m,t)_⊥ ≠ ⊥)
  then if M mk m = t
       then m
       else ⊥
  else ⊥.
```

Fig. 11. A specification of MEE-CBC.

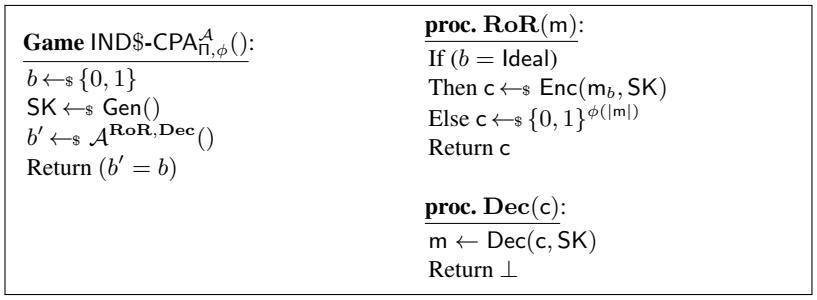

Fig. 12. Game defining specification-level indistinguishability under chosen-plaintext attacks.

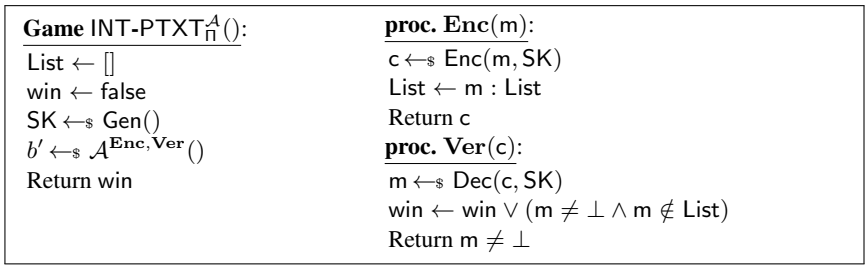

Fig. 13. Game defining specification-level plaintext integrity.

Definition 6 (Ciphertext integrity). *We define the advantage of an adversary $\mathcal{A}$ against the* INT-CTXT *security of SKE scheme Π as*

$$\mathbf{Adv}^{\mathsf{int\text{-}ctxt}}_{\Pi,\mathcal{A}} := \boldsymbol{Pr}\left[\,\mathsf{INT\text{-}CTXT}^{\mathcal{A}}_{\Pi}() \Rightarrow \mathsf{true}\,\right],$$

where game INT-CTXT *is shown in Figure 14.*

MESSAGE AUTHENTICATION CODE: SYNTAX AND SECURITY.. We recall that a (deterministic) message authentication code $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Ver})$ is specified as three algorithms:

- A probabilistic key generation algorithm $\mathsf{Gen}()$ that returns a secret key SK;
- a deterministic tag algorithm $\mathsf{Tag}(\mathsf{m}, \mathsf{SK})$ that returns a tag t on input a message m and the secret key SK; and
- a deterministic verification algorithm $\mathsf{Ver}(\mathsf{m}, \mathsf{t}, \mathsf{SK})$ that returns a boolean on input message m, tag t and secret key SK.

Such that, for every key SK and message m, $\mathsf{Ver}(\mathsf{m}, \mathsf{Tag}(\mathsf{m}, \mathsf{SK})), \mathsf{SK}) = \mathsf{true}$.

Definition 7 (Existential Unforgeability under Chosen Message Attack). *We define the advantage of an adversary $\mathcal{A}$ against the* UF-CMA *security of MAC scheme* MAC

[16] We only require that the MAC's tag size be a multiple of the block size for simplicity of this description. This matches our particular case study, but the formal security proof itself applies more generally.

Game $\mathsf{INT\text{-}CTXT}^{\mathcal{A}}_{\Pi}()$:	**proc.** $\mathbf{Enc}(\mathsf{m})$:
$\mathsf{List} \leftarrow []$	$\mathsf{c} \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{m}, \mathsf{SK})$
$\mathsf{win} \leftarrow \mathsf{false}$	$\mathsf{List} \leftarrow \mathsf{c} : \mathsf{List}$
$\mathsf{SK} \leftarrow\!\!\$\ \mathsf{Gen}()$	Return c
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathbf{Enc},\mathbf{Ver}}()$	**proc.** $\mathbf{Ver}(\mathsf{c})$:
Return win	$\mathsf{m} \leftarrow\!\!\$\ \mathsf{Dec}(\mathsf{c}, \mathsf{SK})$
	$\mathsf{win} \leftarrow \mathsf{win} \vee (\mathsf{m} \neq \perp \wedge \mathsf{c} \notin \mathsf{List})$
	Return $\mathsf{m} \neq \perp$

Fig. 14. Game defining specification-level ciphertext integrity.

Game $\mathsf{UF\text{-}CMA}^{\mathcal{A}}_{\Pi}()$:	**proc.** $\mathbf{Tag}(\mathsf{m})$:
$\mathsf{List} \leftarrow []$	$\mathsf{t} \leftarrow\!\!\$\ \mathsf{Tag}(\mathsf{m}, \mathsf{SK})$
$\mathsf{win} \leftarrow \mathsf{false}$	$\mathsf{List} \leftarrow \mathsf{m} : \mathsf{List}$
$\mathsf{SK} \leftarrow\!\!\$\ \mathsf{Gen}()$	Return t
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathbf{Enc},\mathbf{Ver}}()$	**proc.** $\mathbf{Ver}(\mathsf{m}, \mathsf{t})$:
Return win	$b \leftarrow \mathsf{Ver}(\mathsf{m}, \mathsf{t}, \mathsf{SK})$
	$\mathsf{win} \leftarrow \mathsf{win} \vee (b \wedge \mathsf{m} \notin \mathsf{List})$
	Return b

Fig. 15. Game defining specification-level existential unforgeability under chosen message attack.

as

$$\mathbf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathsf{MAC},\mathcal{A}} := \boldsymbol{Pr}\left[\,\mathsf{UF\text{-}CMA}^{\mathcal{A}}_{\mathsf{MAC}}() \Rightarrow \mathsf{true}\,\right],$$

where implementation-level game UF-CMA *is shown in Figure 15.*

Game $\mathsf{PRP}^{\mathcal{A}}_{\Pi}(b)$:	**proc.** $\mathbf{RoR}(\mathsf{m})$:
$\mathsf{SK} \leftarrow\!\!\$\ \mathsf{Gen}()$	$\mathsf{c} \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{m}, \mathsf{SK})$
$\mathrm{map} \leftarrow \emptyset$	If $(b = \mathsf{Ideal})$ Then
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathbf{RoR}}()$	If $(\mathsf{m} \notin \mathrm{Dom}(\mathrm{map}))$
Return $(b' = b)$	Then
	$\mathsf{c} \leftarrow\!\!\$\ \{0,1\}^{\lvert\mathsf{c}\rvert}$
	$\mathrm{map} \leftarrow \mathrm{map} \cup (\mathsf{m} \mapsto \mathsf{c})$
	Else
	$\mathsf{c} \leftarrow \mathrm{map}(\mathsf{m})$
	Return c

Fig. 16. Game defining specification-level pseudo-random permutation security.

Definition 8 (Pseudo-Random Permutation.). *We define the advantage of an adversary $\mathcal{A}$ against the* PRP *security of cipher Π as*

$$\mathbf{Adv}^{\mathsf{prp}}_{\Pi,\mathcal{A}} := \left|\boldsymbol{Pr}\left[\,\mathsf{PRP}^{\mathcal{A}}_{\Pi}(\mathsf{Real}) \Rightarrow \mathsf{true}\,\right] - \boldsymbol{Pr}\left[\,\mathsf{PRP}^{\mathcal{A}}_{\Pi}(\mathsf{Ideal}) \Rightarrow \mathsf{true}\,\right]\right|,$$

where specification-level game PRP *is shown in Figure 16.*