Automated Analysis and Synthesis of Authenticated Encryption Schemes

Viet Tung Hoang  
University of Maryland  
Georgetown University  
tvhoang@umd.edu

Jonathan Katz  
University of Maryland  
jkatz@cs.umd.edu

Alex J. Malozemoff  
University of Maryland  
amaloz@cs.umd.edu

Abstract

Authenticated encryption (AE) schemes are symmetric-key encryption schemes ensuring strong notions of confidentiality and integrity. Although various AE schemes are known, there remains significant interest in developing schemes that are more efficient, meet even stronger security notions (e.g., misuse-resistance), or satisfy certain non-cryptographic properties (e.g., being patent-free).

We present an automated approach for analyzing and synthesizing blockcipher-based AE schemes, significantly extending prior work by Malozemoff et al. (CSF 2014) who synthesize encryption schemes satisfying confidentiality only. Our main insight is to restrict attention to a certain class of schemes that is expressive enough to capture several known constructions yet also admits automated reasoning about security. We use our approach to generate thousands of AE schemes with provable security guarantees, both known (e.g., variants of OCB and CCM) and new. Implementing two of these new schemes, we find their performance competitive with existing, state-of-the-art AE schemes.

1 Introduction

Historically, symmetric-key encryption schemes were designed only to ensure confidentiality. With the realization that practitioners were often (implicitly) assuming that such schemes also provided some form of integrity, however, researchers began explicit consideration and analysis of encryption schemes additionally satisfying that property [15, 7]. Since then, a tremendous amount of research has focused on the design of authenticated encryption (AE) schemes ensuring both confidentiality and integrity.

While a generic construction of an AE scheme based on any CPA-secure encryption scheme and message authentication code is possible [7], more efficient AE schemes can be devised. One example is OCB [24, 22, 16], which is online (i.e., requires only a single pass over the data), provably secure, and very fast. Unfortunately, due to patent restrictions, the scheme has not gained widespread use. Other well-known AE schemes include CCM [10] and GCM [19]; however, these schemes are slower than OCB [16] and have other disadvantages as well.

Designing AE schemes remains a nontrivial task, and even well-studied AE schemes can sometimes admit surprising optimizations. For example, OCB was first introduced in 2001 [24] and its authors have been active in maintaining and optimizing it, releasing OCB2 in 2004 [22] and OCB3 in 2011 [16]. In spite of all this, Minematsu [20] was able to recently show that a simple change allows OCB to be constructed using only the forward direction of the underlying blockcipher. Overall, the problem of designing AE schemes is still of significant interest to the cryptographic community, as evidenced, e.g., by the CAESAR competition [9].

Motivated by the above concerns, we propose in this work an automated approach for analyzing and synthesizing AE schemes. Our approach builds on and extends the work of Malozemoff et al. [18], who explored a similar goal but for the case of encryption schemes achieving confidentiality only. (We also

1For example, GCM is fairly complex and has a problematic security proof [13], whereas CCM is not online and cannot pre-process associated data.)
explicitly consider messages of arbitrary length, whereas Malozemoff et al. only deal with messages whose
length is a multiple of the block length.) At a high level, as in their work, we view an encryption scheme as
being defined by a directed acyclic graph in which each node corresponds to an instruction (e.g., XORing two
values) and is associated with an intermediate \( n \)-bit value (where \( n \) is the block length). The graph defines
how individual message blocks are processed; messages of arbitrary length are encrypted by iterating the
computation defined by this graph over all the blocks of the message. (We actually consider processing \( \textit{two} \)
message blocks at a time, as this allows us to capture more AE schemes within our framework.) We develop
a type system for the nodes of such graphs, and define constraints on how nodes can be typed based on their
parents’ types. We then show that any “well-typed” graph defines a secure AE scheme. This allows us to
automatically \textit{analyze} a given scheme by checking whether the graph defining the scheme can be properly
typed, and to \textit{synthesize} schemes by enumerating over valid graphs and determining whether each one is
secure.

Although our high-level approach is similar to that of Malozemoff et al., the technical details differ
greatly due to the added challenge of handling integrity (which was left as an explicit open question in their
work). Indeed, we were unable to \textit{directly} extend their work to deal with integrity; instead, we modify their
approach and consider a restricted class of encryption schemes for which an automated analysis of integrity
is tractable. Specifically, we focus on schemes constructed from \textit{tweakable}\footnote{Roughly, a tweakable blockcipher accepts a “tweak” in addition to a key and an input block; for a fixed key, different tweaks should produce “independent-looking” permutations. See the following section for a formal definition.} blockciphers \cite{17} in a particular
way. Several existing AE schemes satisfy our requirements, indicating that our framework is not overly
restrictive. Despite our attention to this restricted class of schemes, our graphs turn out to be a lot more
complex than those considered by Malozemoff et al., and their analysis requires new techniques. As one
illustration of this, we note that in the prior work it is relatively easy to derive the decryption algorithm
from the encryption algorithm, as there is only one path (in the graph representing the encryption algorithm)
from a node representing a plaintext block to a node representing a ciphertext block. In our case, this no
longer holds—e.g., the graphs of schemes like OTR \cite{20} have multiple paths between such pairs of nodes—
and so we instead work backward, starting from a description of the decryption algorithm and then applying
a nontrivial algorithm to derive the graph of the corresponding encryption algorithm.

Using our approach, we are able to synthesize thousands of secure AE schemes, hundreds of which are
“optimal” in the sense that they use only \textit{one} tweakable blockcipher call per block, on par with OCB. These
schemes are provably secure, and our analysis tool not only verifies their security but also gives \textit{concrete}
security bounds. (In contrast, the prior work of Malozemoff et al. \cite{18} only implied asymptotic security.) We
also employ a simple algorithm to identify \textit{fully parallelizable} constructions among the “optimal” schemes;
we discover seventeen new such schemes, five of which use the same number of instructions as OCB. We
implement two of those schemes and find that their efficiency is comparable to that of OCB. Thus, these
schemes may be of interest to practitioners looking for efficient, simple, and patent-free AE schemes. Finally,
in Appendix C, we devise a method for automatically searching for \textit{attacks} on schemes that our approach
cannot prove secure. Interestingly, we find that most schemes that cannot be proven secure by our tools are
indeed susceptible to concrete attacks.

\textbf{Related work.} Recently there has been a growing interest in applying automated techniques to the analysis
and design of cryptographic primitives. In the public-key setting, Barthe et al. \cite{4} introduced an approach
applicable to RSA-based encryption schemes. More recently, Tiwari et al. \cite{25} developed a unified technique
for synthesizing both RSA-based encryption schemes and modes of operation, among other cryptographic
primitives. Other work has looked at automated analysis of assumptions in generic groups \cite{5} with applica-
tions to automated synthesis of signature schemes having certain properties \cite{5, 6}. Finally, Akinyele et
al. \cite{1, 2} have developed tools for analyzing signature and encryption schemes to determine when (and how)
known secure transformations can be applied.
2 Preliminaries

Notation. Let $\mathbb{Z}$ be the set of integers, and let $\mathbb{N}$ denote the set of positive integers. Let $\{0,1\}^*$ be the set of all binary strings, including the empty string. For a string $M$, let $|M|$ be the length of $M$. For $M \in \{0,1\}^*$ and $1 \leq i \leq j \leq |M|$, let $M[i:j]$ denote the $i$-th bit of $M$, and $M[i:j]$ the substring of $M$ from the $i$th to the $j$th bit, inclusive. For two strings $X$ and $Y$, we write $XY$ or $X \parallel Y$ to denote the concatenation of $X$ and $Y$.

We write $x \leftarrow S$ to denote uniform sampling of $x$ from finite set $S$. For finite sets $S_1, S_2$, and random variables $X, Y \in S_1, Z \in S_2$, define $\|X - Y\|_2$, the statistical distance between $X$ and $Y$ given $Z$, as

$$\frac{1}{2} \sum_{x \in S_1, z \in S_2} Pr[Z = z] \cdot \left| Pr[X = v \mid Z = z] - Pr[Y = v \mid Z = z] \right|.$$

Games. We use the code-based, game-playing framework of Bellare and Rogaway [8], and we assume the reader is familiar with this framework.

Tweakable blockciphers [17]. Let $n \in \mathbb{N}$. A tweakable blockcipher on $n$-bit strings with tweak space $\mathcal{T}$ and key space $\mathcal{K}$ is a map $E : \mathcal{K} \times \mathcal{T} \times \{0,1\}^n \rightarrow \{0,1\}^n$ such that $E_K(T, \cdot)$ is a permutation on $\{0,1\}^n$ for any $K \in \mathcal{K}$ and $T \in \mathcal{T}$. Let $E^{-1}$ denote the inverse of $E$, meaning $E^{-1}_K(T, E_K(T, x)) = x$ for $K \in \mathcal{K}$, $T \in \mathcal{T}$, and $x \in \{0,1\}^n$. For brevity we sometimes write $E^{-1}_K(x)$ for $E_K(T, x)$. Define the strong tweakable-PRP advantage of an adversary $A$ against $E$ as

$$\text{Adv}^{\text{prp}}_E(A) = \left| Pr[K \leftarrow \mathcal{K} : A^{E_K(\cdot), E^{-1}_K(\cdot)} \Rightarrow 1] - Pr[\pi \leftarrow \text{Perm}(\mathcal{T}, n) : A^{\pi(\cdot), \pi^{-1}(\cdot)} \Rightarrow 1] \right|,$$

where $\text{Perm}(\mathcal{T}, n)$ is the set of all $\mathcal{T}$-indexed families of permutations on $\{0,1\}^n$. (I.e., $\text{Perm}(\mathcal{T}, n)$ is the set of all functions $\pi : \mathcal{T} \times \{0,1\}^n \rightarrow \{0,1\}^n$ with the property that for each $T \in \mathcal{T}$, the reduced function $\pi(T, \cdot)$ is a permutation on $\{0,1\}^n$.) If the adversary is prohibited from making queries to the second oracle, we drop the word “strong” and write $\text{Adv}^{\text{prp}}_E(A)$ instead.

Authenticated encryption. Rather than view encryption schemes as being randomized or stateful, we follow Rogaway [23] in viewing them as deterministic transformations that take as input a message along with some associated data (which need not be kept secret) as well as a user-supplied nonce. Security is then required to hold as long as the same nonce is never used twice.

Formally, an authenticated encryption (AE) scheme [7, 15, 21, 23] is a tuple $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ with key space $\mathcal{K}$, nonce space $\mathcal{N}$, associated data space $\mathcal{A}$, message space $\mathcal{M}$, and tag length $\tau \in \mathbb{N}$. Both algorithms $\mathcal{E}$ and $\mathcal{D}$ are deterministic. The encryption algorithm $\mathcal{E}$ maps an input tuple $(K, N, A, M) \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M}$ to a ciphertext $C \in \{0,1\}^*$. Decryption $\mathcal{D}$ reverses encryption, mapping an input tuple $(K, N, A, C) \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \{0,1\}^*$ to either a message $M \in \mathcal{M}$ or a distinguished error symbol $\bot$. The correctness requirement demands that $\mathcal{D}_K(N, A, C(M)) = M$ for every $(K, N, A, M) \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M}$.

We define the privacy advantage of an adversary $A$ against an AE scheme $\Pi$ as

$$\text{Adv}^{\text{priv}}_\Pi(A) = \Pr[K \leftarrow \mathcal{K} : A^{E_K(\cdot)} \Rightarrow 1] - \Pr[A^{\$\langle\cdot, \cdot\rangle} \Rightarrow 1],$$

where $\$\langle\cdot, \cdot\rangle$ is an oracle that, on any input $(N, A, M)$, outputs a fresh, uniform $(|M| + \tau)$-bit answer. We require here that the adversary never uses the same nonce twice as input to its oracle. Informally, a scheme satisfies privacy if the privacy advantage of any efficient adversary is small. Nonces used by the honest party during encryption need only be unique\(^3\), not uniform.

For authenticity, the adversary is again given access to an encryption oracle $\mathcal{E}_K(\cdot, \cdot, \cdot)$, and as before must not use the same nonce twice. We say that $A$ outputs a forgery if it outputs $(N, A, C)$ such that $\mathcal{D}_K(N, A, C) \neq \bot$ and $C$ was not the result of a prior oracle query $\mathcal{E}_K(N, A, M)$ for some message $M$. We define the authenticity advantage of $A$ as $\text{Adv}^{\text{auth}}_\Pi(A) = \Pr[K \leftarrow \mathcal{K} : A^{E_K(\cdot, \cdot)} \text{ outputs a forgery}].$ Informally, a scheme satisfies authenticity if the authenticity advantage of any efficient adversary is small.

\(^3\)The requirement that nonces be unique is necessary, since repeating $(N, A, M)$ will repeat the corresponding ciphertext. For real schemes such as OCB, reusing a nonce is devastating, damaging the privacy and authenticity of not just past queries, but also future ones. It is the responsibility of the implementation to ensure that nonces are unique.
3 Automated Security Analysis

We now describe our approach to the automated analysis of AE schemes constructed from tweakable block-ciphers following a particular template (cf. Section 3.1). Although this template does not capture all known AE schemes, it is expressive enough to include simplified variants of, e.g., OCB [22], XCBC [12], COPA [3], OTR [20], and CCM [10].

As discussed in the Introduction, we view an encryption scheme as being defined by a directed acyclic graph in which each node is associated with an instruction and carries an n-bit intermediate value. In Section 3.2 we describe a type system for the nodes of such graphs, and show how to use these types for reasoning about properties of the intermediate values that those nodes carry. Then, in Section 3.3, we show how this reasoning enables us to automatically verify whether an AE scheme, given by its graph representation, satisfies privacy and authenticity.

3.1 A Template for AE Schemes

Fix associated data space A, and let $\mathcal{N} = \{0, 1\}^n$. Let $T = \mathcal{N} \times A \times Z$ and let $E : K \times T \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be a tweakable block-cipher. We consider AE schemes $\Pi[E] = (\mathcal{F}, \mathcal{D})$ that use $E$ as an oracle. The schemes we consider have message space $\mathcal{M} = (\{0, 1\}^{2n})^*$ and are built from algorithms $(\text{Enc}, \text{Dec}, \text{Tag})$ having the following form:

- $\text{Enc}^E_{\mathcal{K}}$ takes as input tweak $T = (N, A, v) \in T$, an initial state $X \in \{0, 1\}^{2n}$, and a (double-length) message block $M \in \{0, 1\}^{2n}$. It outputs a (double-length) ciphertext block $C \in \{0, 1\}^{2n}$ and final state $Y \in \{0, 1\}^{2n}$. This algorithm makes a fixed number of queries to $E_K$, denoted by $\text{Cost}(\Pi)$, and we require that the tweak in the $i$th such query is $(N, A, v + i - 1)$.

- $\text{Dec}^E_{K, K^{-1}}$ “inverts” algorithm $\text{Enc}$ in the following sense: if $\text{Enc}^E_K(T, X, M) = (Y, C)$ then it holds that $\text{Dec}^E_{K, K^{-1}}(T, X, C) = (Y, M)$.

- $\text{Tag}^E_X$ takes as input tweak $T \in T$ and initial state $X \in \{0, 1\}^{2n}$, and produces a tag $V \in \{0, 1\}^n$. It makes a single query to $E_K$ using tweak $T$.

The encryption/decryption algorithms $(\mathcal{F}, \mathcal{D})$ of $\Pi$ are then defined as in Figure 3.1, where we require $\tau \leq n$. Roughly, to encrypt a message $M = M_1M_2 \cdots M_m$ using nonce $N$ and associated data $A$, set the initial state $X = 0^{2n}$ and set $T = (N, A, 1)$. Then, iteratively process two message blocks at a time using $\text{Enc}$, each time updating the initial state and outputting the next two ciphertext blocks. After processing the entire message, $\text{Tag}$ is used to compute a tag based on the final state output by $\text{Enc}$ and a designated tweak that depends on the message length; the (truncated) tag is appended to the ciphertext.

Graph representation. As in the work of Malozemoff et al. [18], we represent algorithms $\text{Enc}$, $\text{Dec}$, and $\text{Tag}$ as directed acyclic graphs, where each node is associated with an instruction and carries an n-bit value. The n-bit value on each node is determined by applying the instruction at that node to the values at the parent nodes. In the next section we introduce a system for “typing” the nodes of such graphs; our main theorem states that AE schemes built from $\text{Enc}$, $\text{Dec}$, and $\text{Tag}$ algorithms whose graphs can be correctly typed are secure.

The main instructions we support are $\text{XOR}$, which computes the XOR of two n-bit strings, and $\text{TBC}$, which invokes the tweakable blockcipher or its inverse. We also have an instruction $\text{DUP}$ that duplicates a

---

4For efficiency, the real-world variants are often built directly from a blockcipher instead of a tweakable one, and employ a scheme-specific way to handle fragmentary data. The real-world CCM is not online due to its treatment of fragmentary data, whereas our variant is online. OCB is built from a tweakable blockcipher, but the tweaks are $(N, A, i)$. To handle associated data, OCB employs an XOR-universal hash (based on a tweakable blockcipher), and XORs the hash image to the tag.

5One can extend the XEX construction [22] of a tweakable blockcipher from any blockcipher to tweak space $\mathcal{N} \times \{0, 1\}^* \times Z$ as follows: On tweak $(N, A, i)$, one applies a (keyed) universal hash to $(N, A)$ to derive a synthetic nonce $N'$, and apply the XEX construction on $(N', i)$. By buffering $L$, hashing need only be done once per message.

6Messages of arbitrary length can be handled by naive padding in the usual way. In Appendix B we describe a more efficient approach for handling messages of arbitrary length.
\(E_K(N, A, M)\)
\[
X := 0^n; \quad v := 1; \quad M_1 \cdots M_{2m} := M \quad \text{// } |M_i| = n
\]
for \(i = 1\) to \(m\)
\[
T := (N, A, v)
\]
\[
(Y, C_{2i-1}C_{2i}) := Enc^{E_K}(T, X, M_{2i-1}M_{2i}) \quad \text{// } |C_i| = n
\]
\[
v := v + \text{Cost}(II); \quad X := Y
\]
\[
T := (N, A, 1 - v); \quad V := \text{Tag}^{E_K}(T, X)
\]
return \(C_1 \cdots C_{2m} \parallel V[1, \tau]\

\(D_K(N, A, C)\)
if \(|C| \neq \tau \pmod{2n}\) then return \(\perp\)
\(C_1 \cdots C_{2m} \parallel \text{tag} := C \quad \text{// } |C_i| = n \text{ and } |\text{tag}| = \tau\)
\[
X := 0^n; \quad v := 1
\]
for \(i = 1\) to \(m\)
\[
T := (N, A, v)
\]
\[
(Y, M_{2i-1}M_{2i}) := \text{Dec}^{E_K,E^{-1}_K}(T, X, C_{2i-1}C_{2i})
\]
\[
v := v + \text{Cost}(II); \quad X := Y
\]
\[
T := (N, A, 1 - v); \quad V := \text{Tag}^{E_K}(T, X)
\]
if \(\text{tag} \neq V[1, \tau]\) then return \(\perp\) else return \(M_1 \cdots M_{2m}\)

**Figure 3.1:** Code of an AE scheme \(\Pi = (K, E, D)\) following our template. The scheme is based on a tweakable blockcipher \(E\) and a triple of deterministic algorithms \((Enc, Dec, Tag)\).

Node values. Nodes corresponding to input blocks are labeled \text{IN}, those corresponding to output blocks are labeled \text{OUT}, those corresponding to the initial state are labeled \text{INI}, and those corresponding to the final state are labeled \text{FIN}. These labels, along with their in-/out-degree, are summarized for convenience next:

<table>
<thead>
<tr>
<th>Name</th>
<th>In-deg</th>
<th>Out-deg</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{IN}</td>
<td>0</td>
<td>1</td>
<td>Input block</td>
</tr>
<tr>
<td>\text{INI}</td>
<td>0</td>
<td>1</td>
<td>Initial state</td>
</tr>
<tr>
<td>\text{FIN}</td>
<td>1</td>
<td>0</td>
<td>Final state</td>
</tr>
<tr>
<td>\text{OUT}</td>
<td>1</td>
<td>0</td>
<td>Output block</td>
</tr>
<tr>
<td>\text{DUP}</td>
<td>1</td>
<td>2</td>
<td>Duplicate</td>
</tr>
<tr>
<td>\text{XOR}</td>
<td>2</td>
<td>1</td>
<td>XOR operation</td>
</tr>
<tr>
<td>\text{TBC}</td>
<td>1</td>
<td>1</td>
<td>Tweakable blockcipher</td>
</tr>
</tbody>
</table>

**Figure 3.2** illustrates the OCB scheme \([22]\) and its \(Enc, Dec,\) and \(Tag\) graphs, whereas **Figure 3.3** shows the corresponding algorithms. (In OCB, only the first \(n\) bits of the state are used, so we treat the state as an element of \(\{0, 1\}^n.\) Note that Figure 3.2 is informal and omits information needed to fully specify OCB; see next for formal details of how graphs are specified.

Formally, we denote a graph \(G\) by a tuple \((d, r, F, P, L)\), where \(d \in \{2, 4\}\) is the total number of \text{IN} and \text{INI} nodes (\(Enc\) and \(Dec\) graphs have \(d = 4\); \(Tag\) graphs have \(d = 2\)), and \(r \in \mathbb{N}\) is the total number of nodes. Each node in the graph is numbered from 1 to \(r\), and we require that if node \(i\) is a parent of node \(j\) then \(i < j\). (This ensures that \(G\) is acyclic.) Let \(Nodes\) denote the power set of \(\{1, \ldots, r\}\), and let \(Inst = \{\text{IN}, \ldots, \text{TBC}\}\) be the set of instructions. Then \(F : \{1, \ldots, r\} \rightarrow Inst\) gives the instruction of each node and \(P : \{1, \ldots, r\} \rightarrow Nodes\) gives the set of parents for each node. We require that \(F(1) = F(2) = \text{INI}\) and \(F(r) = \text{OUT}\). For \(Enc\) and \(Dec\) graphs, we additionally require that \(F(3) = F(4) = \text{IN}, F(r - 2) = F(r - 3) = \text{FIN}\), and \(F(r - 1) = \text{OUT}\).

For \(Enc\) and \(Dec\) graphs, let \(S \subseteq \{1, \ldots, r\}\) be the set of all nodes corresponding to a \text{TBC} instruction. Function \(L : S \rightarrow \mathbb{Z}\) specifies, for each such node \(i\), whether the tweakable blockcipher is computed in the forward direction (if \(L(i) \geq 0\)) or the reverse direction (if \(L(i) < 0\)) at that node.\(^7\) (Note that \(L(i) \geq 0\) for

\(^7\)While it might be conceptually simpler to use two different instructions for \(E_K\) and \(E^{-1}_K\), instead of just a single \text{TBC} instruction with positive/negative labels, our approach is an optimization that prunes the search space when synthesizing schemes (cf. Section 4).
Enc graphs.) Moreover, \(|L(i)|\) determines the tweak at node \(i\); i.e., on input tweak \((N, A, v)\), the tweak at node \(i\) is \((N, A, v + |L(i)|)\).

Let \(G^− = (d, r, F, P)\) denote the unlabeled graph corresponding to a graph \(G\). In Section 3.2, we introduce a type system and show that one can reason about the security of an AE scheme by evaluating the scheme’s unlabeled graphs.

Fix some graph \(G\), tweakable blockcipher \(E\), and key \(K\). Given a tweak \(T\) and \(n\)-bit values for all INI/IN nodes in \(G\), we can naturally define an \(n\)-bit value \(Z_i\) associated with each node \(i\) in the graph. We describe this formally as procedure \(\text{Eval}\) in Figure 3.4, which shows how to compute \(Z_i\) given values \(Z_1, \ldots, Z_d \in \{0, 1\}^n\) and tweak \(T\).

### 3.2 A Type System for AE Schemes

Let \(\text{Types} = \{\text{\$}, \bot, 0, 1\}\) be a set of “types” we can assign to nodes. Intuitively, ‘\(\$\)’ indicates a node whose output value is (pseudo)random (when the key \(K\) for \(E\) is random and secret), whereas ‘\(\bot\)’ indicates a node whose output value is arbitrary (i.e., potentially controlled by an attacker). Looking ahead, types ‘0’ and ‘1’ will be used to compare values on the same node in two different decryption queries using the same nonce and associated data; ‘0’ means the corresponding values are the same, and ‘1’ means they are different.

In Figure 3.5, we define a deterministic procedure \(\text{Map}\) that takes as input an unlabeled graph \(G^−\), pre-assigned types \(\text{type}_1, \ldots, \text{type}_d \in \text{Types}\) for all the INI/IN nodes, and a boolean flag \(\text{rand}\), and returns a map \(R\) that associates each node \(i\) with a pair \((\text{type}_i, \text{ctr}_i)\) \(\in \text{Types} \times \mathbb{N}\). The procedure \(\text{Map}\) traverses the graph in topological order and assigns types to each node of the graph based on the instruction associated with that node and the types of its parents. These types are used for probabilistic reasoning about the underlying \(n\)-bit values on that node; e.g., we show that if a node has type \(\$\) then the \(n\)-bit value of that node is (pseudo)random. The \(\text{ctr}\) values are used as “timestamps” for values output by TBC nodes in order to
briefly explain this reasoning. The fact that type
utilize the $\text{XOR}$ prior query, and thus the $\text{XOR}$ the two queries, and thus that node gets type $0$. On the other hand, if one parent is typed $0$ and the other
those nodes are equal in the two queries, then clearly the value computed at the $\text{XOR}$ definitive and thus
is independent of the value of $y$

Procedure to compute the value $Z_i$ of each node $i$ in a graph $G$, given input $Z_1, \ldots, Z_d$ and tweak $T$.

\begin{figure}[h]
\centering
\begin{verbatim}
proc Eval\textsubscript{$E_K, E_K^{-1}$} (G, T, Z_1, \ldots, Z_d) \\
(d, r, F, P, L) := G \\
for i = d + 1 to r do \\
  if $F(i)$ is in \{DUP, OUT, FIN\} then \\
    \{p\} := $P(i)$; $Z_i := Z_p$ \\
  else \// $F(i)$ is TBC \\
    $\ell := L(i)$; \{(N, A, v) := T \}
    $T^* := (N, A, v + [\ell])$; \{p\} := $P(i)$ \\
  if $\ell > 0$ then $Z_i := E_K(T^*, Z_p)$ else $Z_i := E_K^{-1}(T^*, Z_p)$ \\
return $(Z_1, \ldots, Z_r)$
\end{verbatim}
\caption{ Procedure to compute the value $Z_i$ of each node $i$ in a graph $G$, given input $Z_1, \ldots, Z_d$ and tweak $T$.}
\end{figure}

determine independence among values of type $\dollar$. Finally, the \texttt{rand} flag denotes whether the nonce/associated data are fresh.

For \texttt{FIN}, \texttt{OUT}, and \texttt{DUP} nodes, \texttt{Map} simply propagates the type of the parent node. For \texttt{TBC} nodes, if the nonce or input is fresh then the output is (pseudo)random and independent of any prior random values, and so the node gets type $\dollar$; otherwise, we propagate the type of the parent node.

For \texttt{XOR} nodes, we have several cases. If the two input nodes $x$ and $y$ are typed $\text{type}_x$ and $\text{type}_y$, respectively, with $(\text{type}_x, \text{type}_y) \in \{(0, 0), (0, 1), (1, 0)\}$, then we type the \texttt{XOR} node as $\text{type}_x \oplus \text{type}_y$. We briefly explain this reasoning. The fact that $\text{type}_x, \text{type}_y \in \{0, 1\}$ means there is a prior query using the same nonce and associated data. If the two parents have type 0, indicating that the values computed at those nodes are equal in the two queries, then clearly the value computed at the \texttt{XOR} node is also equal in the two queries, and thus that node gets type 0. On the other hand, if one parent is typed 0 and the other is typed 1, then the value computed at the \texttt{XOR} node will be different from the corresponding value in the prior query, and thus the \texttt{XOR} node is assigned type 1. If $(\text{type}_x, \text{type}_y) = (1, 1)$ then we cannot say anything definitive and thus \texttt{Map} assigns type $\bot$ to the \texttt{XOR} node. Finally, suppose input node $x$ has type $\dollar$. Here we utilize the $\text{ctr}$ values. If the $\text{ctr}$ value at $x$ is different from the $\text{ctr}$ value of $y$, then the (random) value of $x$ is independent of the value of $y$, and hence we assign the \texttt{XOR} node type $\dollar$.  

\begin{figure}[h]
\centering
\begin{verbatim}
proc Map(G^−, \text{type}_1, \ldots, \text{type}_d, \text{rand}) \\
(d, r, F, P) := G^−; \text{maxCtr} := 0 \\
for i = 1 to d do \\
  if $\text{type}_i = \dollar$ then $R(i) := ($\dollar$, 1)$; \text{maxCtr} := 1 \\
  else $R(i) := (\text{type}_i, 0)$ \\
for i = d + 1 to r do \\
  if $F(i)$ is in \{FIN, OUT, DUP\} then \\
    \{p\} := $P(i)$; $R(i) := R(p)$ \\
  else \// $F(i)$ is TBC \\
    \{p\} := $P(i)$; $(x, \text{ctr}) := R(p)$ \\
    if $x \in \{1, \dollar\}$ or (\text{rand} = true) then \\
      \text{maxCtr} := \text{maxCtr} + 1; $R(i) := ($\dollar$, \text{maxCtr})$ \\
    else $R(i) := (x, \text{ctr})$ \\
  else \// $F(i)$ is \texttt{XOR} \\
    \{p_1, p_2\} := $P(i)$; $(x, \text{ctr}) := R(p_1)$; $(y, \text{ctr}') := R(p_2)$ \\
    \// Assume that $\text{ctr} \geq \text{ctr}'$ \\
    if $(x, y) \in \{(0, 0), (0, 1), (1, 0)\}$ then $R(i) := (x \oplus y, \text{ctr})$ \\
    elseif $x = \dollar$ and $\text{ctr} > \text{ctr}'$ then $R(i) := ($\dollar$, \text{ctr})$ \\
    else $R(i) := (\bot, \text{ctr})$ \\
return $R$
\end{verbatim}
\caption{ A procedure for generating a mapping $R : \{1, \ldots, r\} \rightarrow \text{Types} \times \mathbb{N}$ for a given unlabeled graph.}
\end{figure}

\begin{figure}[h]
\centering
\begin{verbatim}
for i = 1 to d do \\
  if $\text{type}_i = \dollar$ then $R(i) := ($\dollar$, 1)$; \text{maxCtr} := 1 \\
  else $R(i) := (\text{type}_i, 0)$ \\
end for
\end{verbatim}
\caption{ A procedure for generating a mapping $R : \{1, \ldots, r\} \rightarrow \text{Types} \times \mathbb{N}$ for a given unlabeled graph.}
\end{figure}
In the next two lemmas we show how determining the types for an unlabeled graph can be used to reason about the values that one obtains when evaluating the labeled graph. We first show that all values typed $\bot$ by Map (when inputs are typed $\bot$ and hence may be under arbitrary control of the adversary) are indeed random when computed using Eval and a truly random tweakable permutation.

**Lemma 3.1.** Let $G = (d,r,F,P,L)$ and let $n \geq 1$ be an integer. Set $R := Map(G^-,\bot,\ldots,\bot,\text{true})$. Fix arbitrary $Z_1,\ldots,Z_d \in \{0,1\}^n$ and $T \in \mathcal{T}$, and consider the following probabilistic experiment:

1. Choose $f \leftarrow \text{Perm}(T,n)$.
2. Run $(Z_1,\ldots,Z_d) := \text{Eval}_{f,f^{-1}}(G,T,Z_1,\ldots,Z_d)$. Then for any $j$ with $R(j) = (\$, \text{ctr}_j)$, the random variable $Z_j$ is uniform and independent of $\{Z_i \mid \text{ctr}_i < \text{ctr}_j\}$.

**Proof.** First note that for any node $i$ and its parent $p$, we have $\text{ctr}_i \geq \text{ctr}_p$. Thus, there is a topological ordering $s_1,\ldots,s_r$ of the nodes such that the sequence $\text{ctr}_{s_1},\ldots,\text{ctr}_{s_r}$ is non-decreasing, and $s_i = i$ for $i \leq d$. Write $i < j$ if node $i$ precedes node $j$ in this topological order. We prove by induction (with respect to $\prec$) that for all $j$ we have (i) $\text{type}_j \in \{\bot,\$\}$ and (ii) if $\text{type}_j = \$ then $Z_j$ is uniform and independent of $\{Z_i \mid \text{ctr}_i < \text{ctr}_j\}$. Note that these claims are trivially true when $j \leq d$, because then $\text{type}_j = \bot$.

Suppose both claims hold for all $i < j$. If $F(j) \in \{\text{FIN,OUT,DUP}\}$ then let $p = \text{ctr}_j$ be the parent of $j$. Since $(\text{type}_j,\text{ctr}_j) = (\text{type}_p,\text{ctr}_p)$, the claims follow for $j$. If $F(j) = \text{TBC}$ then $\text{type}_j = \$, proving claim (i). Since $f \leftarrow \text{Perm}(T,n)$, and Eval never repeats a tweak in querying $f$, we see that random variable $Z_j$ is uniform and independent of $\{Z_i \mid \text{ctr}_i < \text{ctr}_j\}$, justifying claim (ii). Finally, say $F(j) = \text{XOR}$. Let $i$ and $t$ be the parents of $j$, and assume $t < i$. Then $\text{type}_j,\text{type}_i \in \{\bot,\$,false\}$, and thus so is $\text{type}_j$, proving claim (i). For claim (ii), note that $\text{type}_j = \$ only if $\text{type}_i =$ $ and $\text{ctr}_i > \text{ctr}_j$. Since $\text{ctr}_j = \text{ctr}_i$, the claim follows. \( \Box \)

The next lemma proves a similar property as above for pairs of queries. Consider the query $(Y_1,\ldots,Y_r) := \text{Eval}_{f,f^{-1}}(G,T,Y_1,\ldots,Y_d)$ followed by query $(Z_1,\ldots,Z_r) := \text{Eval}_{f,f^{-1}}(G,T,Z_1,\ldots,Z_d)$, where each $Z_i$ is either chosen equal to $Y_i$ (and thus $\text{type}_i = 0$), distinct from $Y_i$ (and thus $\text{type}_i = 1$), or uniformly (and thus $\text{type}_i = \$). We show that for all nodes $j$ of type $\bot$ assigned by Map($G^-,\text{type}_1,\ldots,\text{type}_d,\text{false}$), the statistical difference between $Z_j$ and uniform is small, even conditioned on all the $\{Y_i\}$.

**Lemma 3.2.** Let $G = (d,r,F,P,L)$ and let $n \geq 1$ be an integer. Fix arbitrary $Y_1,\ldots,Y_r \in \{0,1\}^n$ and $T \in \mathcal{T}$ such that the set $S = \{f \in \text{Perm}(T,n) \mid (Y_1,\ldots,Y_r) = \text{Eval}_{f,f^{-1}}(G,T,Y_1,\ldots,Y_d)\}$ is non-empty. For each $i \leq d$, choose $Z_i$ and $\text{type}_i$ in one of the following ways: (i) $Z_i = Y_i$ and $\text{type}_i = 0$, (ii) $Z_i \neq Y_i$ and $\text{type}_i = 1$, or (iii) $Z_i \leftarrow \{0,1\}^n$ and $\text{type}_i = \$. Let $R = Map(G^-,\text{type}_1,\ldots,\text{type}_d,\text{false})$. Consider the following probabilistic experiment:

1. Choose $f \leftarrow S$.
2. Run $(Z_1,\ldots,Z_d) := \text{Eval}_{f,f^{-1}}(G,T,Z_1,\ldots,Z_d)$. Then for any $j$ with $R(j) = (\$, \text{ctr}_j)$, the statistical difference between the random variable $Z_i$ and uniform, conditioned on $\{Z_i \mid \text{ctr}_i < \text{ctr}_j\}$ and all the $\{Y_i\}$, is at most $2 \cdot \text{ctr}_j/2^n$.

**Proof.** As in the previous lemma, there is a topological ordering $s_1,\ldots,s_r$ of the nodes such that $\text{ctr}_{s_1},\ldots,\text{ctr}_{s_r}$ is non-decreasing and $s_i = i$ for $i \leq d$. Write $i < j$ if $i$ precedes $j$ in this topological order. We prove by induction (with respect to $\prec$) that for all $j$: (i) if $\text{type}_j = 0$ then $Z_j = Y_j$, (ii) if $\text{type}_j = 1$ then $Z_j \neq Y_j$, and (iii) if $\text{type}_j = \$ then the statement of the lemma holds. These claims all hold when $j \leq d$.

Suppose that all three of the claims hold for all $i < j$. If $F(j) \in \{\text{FIN,OUT,DUP}\}$ then let $p = \text{ctr}_j$ be the parent of $j$. Since $(\text{type}_j,\text{ctr}_j) = (\text{type}_p,\text{ctr}_p)$, the claims follow easily in this case. If $F(j) = \text{XOR}$, let $t,i < j$ be the parents of $j$, and assume $t < i$. Note that $\text{type}_i \in \{0,1\}$ only if $\text{type}_p,\text{type}_i \in \{0,1\}$ and at most one of these values is $1$, in which case the claims all hold. On the other hand, $\text{type}_j = \$ only if $\text{type}_i = \$ and $\text{ctr}_i > \text{ctr}_j$, in which case the claims also follow. Finally, if $F(j) = \text{TBC}$ then let $i$ be the parent of $j$. Let $\ell = L(j)$, and let $T = (N,A,v)$. Then $Y_j = f(T^*,Y_p)$ and $Z_j = f(T^*,Z_p)$, where $T^* = (N,A,v + |\ell|)$. Consider the following cases:

8
Case 1. \( \text{type}_i \in \{0, \perp \} \). Then \((\text{type}_j, \text{ctr}_j) = (\text{type}_i, \text{ctr}_i)\) and the claims follow.

Case 2. \( \text{type}_i = 1 \). Then \( \text{type}_j = \$ \) and \( \text{ctr}_j \geq 1 \). First, since \( \text{ctr}_t \geq \text{ctr}_j \) when \( t \) is a descendant of \( j \), we see that no node \( t \) with \( \text{ctr}_t < \text{ctr}_j \) is a descendant of \( j \). Next, since \( Z_i \neq Y_i \) and we use a different tweak for each TBC node, \( Z_j \leftarrow \{0, 1\}^n \backslash \{Y_j\} \) is independent of \( \{Y_t \mid t \leq r\} \) and \( \{Z_t \mid \text{ctr}_t < \text{ctr}_j\} \). Hence the statement of the lemma follows.

Case 3. \( \text{type}_i = \$. Then \( \text{type}_j = \$ \). By the induction hypothesis, \( Z_i \) is \( (2\text{ctr}_i/2^n) \)-close to uniform (even conditioned on \( \{Y_t \mid t \leq r\} \) and \( \{Z_t \mid \text{ctr}_t < \text{ctr}_i\} \)). If \( Z_i \neq Y_i \), which occurs except with probability at most \( (2\text{ctr}_i+1)/2^n \), then \( Z_j \) is \( 2^{-n} \)-close to uniform (even conditioned on all the \( \{Y_t\} \) values and \( \{Z_t \mid \text{ctr}_t < \text{ctr}_j\} \)). Hence, overall, \( Z_j \) is \( (2\text{ctr}_j+2)/2^n \)-close to uniform (conditioned on \( \{Y_t \mid t \leq r\} \) and \( \{Z_t \mid \text{ctr}_t < \text{ctr}_j\} \)) and the statement of the lemma follows since we have \( \text{ctr}_j \geq \text{ctr}_i + 1 \).

3.3 Verifying Privacy and Authenticity

We use Lemmas 3.1 and 3.2 to automatically check if a candidate AE scheme is secure in the sense of both privacy and authenticity. Specifically, Figure 3.6 shows procedures \textsf{Priv} and \textsf{Auth} to check for privacy and

authenticity, respectively, of an AE scheme $Π$.

Intuitively, for privacy we verify that the tag and all the ciphertext blocks output by the scheme are random and independent (namely, have type $\perp$ and distinct counter values) even when the inputs—that is, the message blocks—are controlled by the adversary (namely, have type $\perp$). We remark that the values of $\text{ctr}$ assigned to nodes by the map $R$ output by $\text{Map}$ depend on the topological order in which $\text{Map}$ traverses the input graph; see Figure 3.7 for an example. Thus, there are schemes which, depending on the order in which the graph is traversed, are accepted or (incorrectly) rejected by $\text{Priv}$ (due to the $\text{ctr}$ values for the $\text{OUT}$ nodes being equal). This shows that the test is sound but not complete.$^8$

The authenticity check for a scheme $(\text{Enc}, \text{Dec}, \text{Tag})$ is more complicated. We now argue informally that if a scheme passes the checks of algorithm $\text{Auth}$ (cf. Figure 3.6), then the scheme satisfies authenticity. To see this, consider a candidate forgery $(N, A, C)$ output by an adversary. First suppose there was no prior query $(N, A, \ast)$ to the encryption oracle. $\text{Auth}$ verifies that the $\text{Tag}$ algorithm outputs a random tag when the tweak for the $\text{TBC}$ node in $\text{Tag}$ was not used previously; thus, the candidate forgery will be invalid except with probability $2^{-7}$. (Recall that $τ$ is the tag length.) Next, consider the case that there was a prior encryption query $(N, A, M)$, and let $C'$ be the corresponding ciphertext. Then $C \neq C'$; otherwise $(N, A, C)$ is not a valid forgery. If $C$ and $C'$ only differ in their tags, the candidate forgery must be invalid because the tag is uniquely determined by $N$, $A$, and the rest of the ciphertext. Otherwise, consider the first pair of blocks in which $C$ and $C'$ differ. $\text{Auth}$ verifies that (i) the first half of the final state produced by $\text{Dec}$ when run on those blocks is random, (ii) $\text{Dec}$ has the property that if the first half of its initial state is random, then the first half of the final state it outputs is random, and (iii) $\text{Tag}$ has the property that if the first half of its initial state is random, then the tag it outputs is random.$^9$ Taken together, these imply that the tag will be random, and hence the candidate forgery will be invalid except with probability $2^{-7}$.

To demonstrate the strength of our approach, consider a modified version of the OTR scheme [20]. The original OTR scheme (cf. Figure 3.8) is secure, which our automated tests confirm. If, however, the scheme is changed so that $Σ$ is computed as the checksum of the odd blocks $M_1 \oplus M_3 \oplus \cdots$, rather than the even blocks, then it becomes insecure. And, indeed, the modified scheme does not pass our tests. Namely, on input $(0, 0, 1, 0)$ to $\text{Map}$ we find that the required $\text{FIN}$ node is typed 1 instead of $\perp$.

Proofs of correctness. We now prove that schemes that pass our tests are secure. We first show that if $\text{Priv}$ returns $\text{true}$ when given the (unlabeled) graphs corresponding to the $\text{Enc}$ and $\text{Tag}$ components of some AE scheme, then that scheme satisfies privacy when instantiated with a secure tweakable blockcipher.

---

$^8$In Appendix C we describe a technique for generating attacks given a scheme which fails the tests of Figure 3.6. Looking ahead, we find only a handful of schemes which we can neither prove secure nor find concrete attacks for; see Section 4.

$^9$Although here we are considering just the first half of the final/initial state, if one switches to the second half then one will get the same set of synthesized schemes: if one changes the topological ordering in the graphs so that the first $\text{FIN/INI}$ node becomes the second one, and vice versa, then the scheme remains the same.
Theorem 3.3. Let $\Pi[E] = (K, E, D)$ be an AE scheme for which $\text{Priv}(G_1^-, G_2^-) = \text{true}$, where $G_1^-$ and $G_2^-$ are the unlabeled graphs for algorithms $\text{Enc}$ and $\text{Tag}$ of $\Pi$, respectively. Then for any adversary $A$, there is an adversary $B$ with $\text{Adv}_{\Pi[E]}^{\text{priv}}(A) \leq \text{Adv}_{E}^{\text{prp}}(B)$. Adversary $B$ has the same running time as $A$ and makes at most $(\text{Cost}(\Pi) + 1) \cdot \sigma/2$ queries, where $\sigma$ is the number of message blocks in the queries of $A$.

Proof. Adversary $B$ runs $A$. For each of $A$’s queries $(N, A, M)$, adversary $B$ runs the encryption scheme $\Pi[E]$ on $(N, A, M)$ with each call to $E_K$ replaced by a query to $B$’s oracle, and returns the ciphertext to $A$. Finally, $B$ outputs the same guess as $A$. Let $\Pi[\pi]$ be the ideal variant of $\Pi[E]$, where calls to $E_K$ are replaced by corresponding queries to $\pi$, with $\pi \leftarrow \text{Perm}(T, n)$. It suffices to show that $\text{Adv}_{\Pi[\pi]}^{\text{priv}}(A) = 0$.

Consider experiments $H_1 - H_4$ in Figure 3.9. The adversary has oracle access to the encryption scheme of $\Pi[\pi]$ in experiment $H_1$, and oracle access to $(\cdot, \cdot, \cdot)$ in experiment $H_4$. Experiment $H_2$ is identical to $H_1$, except that we re-sample $\pi \leftarrow \text{Perm}(T, n)$ each time we use $\text{Enc}$ or $\text{Tag}$. Since a tweak to $\pi$ is never repeated, $\Pr[H_1^A \rightarrow \text{true}] = \Pr[H_2^A \rightarrow \text{true}]$. In experiment $H_3$, instead of calling $\text{Tag}\pi(T, X)$ to get the tag, we sample the tag at random. Considering lines 02–03 of $\text{Priv}$ (and the fact that $\text{Priv}(G_1^-, G_2^-) = \text{true}$) in conjunction with Lemma 3.1 shows that the string $V := \text{Tag}\pi(T, X)$ is uniform and so experiments $H_2$ and $H_3$ are identical. Finally, experiment $H_4$ is identical to $H_3$, except that instead of calling $\text{Enc}\pi(T, X, M_{2i-1}M_{2i})$ to get the blocks $C_{2i-1}C_{2i}$ of the ciphertext, we sample them at random. Considering lines 04–05 of $\text{Priv}$ (and the fact that $\text{Priv}(G_1^-, G_2^-) = \text{true}$) in conjunction with Lemma 3.1 shows that the output blocks of $\text{Enc}\pi(T, X, M_{2i-1}M_{2i})$ are uniform and independent (and this is true even conditioned on all prior ciphertext blocks). Hence $H_3$ and $H_4$ are identical, and $\text{Adv}_{\Pi[\pi]}^{\text{priv}}(A) = \Pr[H_1^A \rightarrow \text{true}] - \Pr[H_4^A \rightarrow \text{true}] = 0$.

Next, in Theorem 3.4, we show that if $\text{Auth}$ in Figure 3.6 returns true when given graphs corresponding to the $\text{Dec}$ and $\text{Tag}$ components of some AE scheme, then that scheme satisfies authenticity when instantiated with a secure tweakable blockcipher. (Examination of the proof shows that if algorithm $\text{Dec}$ does not use $E_K^{-1}$, as in the case of OTR, then the term $\text{Adv}_{E}^{\text{prp}}(B)$ in Theorem 3.4 can be weakened to $\text{Adv}_{E}^{\text{prp}}(B)$.)

Theorem 3.4. Let $\Pi[E] = (K, E, D)$ be an AE scheme such that $\text{Auth}(G_1^-, G_2^-) = \text{true}$, where $G_1^-, G_2^-$ are the unlabeled graphs for algorithms $\text{Dec}$ and $\text{Tag}$ of $\Pi$, respectively. Then for any adversary $A$, there is an adversary $B$ with $\text{Adv}_{\Pi[E]}^{\text{auth}}(A) \leq 2^{-\tau} + \ell \cdot (\text{Cost}(\Pi) + 2)/2^n + \text{Adv}_{\Pi[E]}^{\text{prp}}(B)$, where $\ell$ is the number of blocks in the forgery output by $A$. Adversary $B$ has the same running time as $A$ and makes at most $(\text{Cost}(\Pi) + 1) \cdot \sigma/2$ queries, where $\sigma$ is the total number of message blocks in the queries of $A$.

Proof. Adversary $B$ runs $A$. For each of $A$’s encryption queries, $B$ runs the encryption scheme of $\Pi[E]$ but with each call to $E_K$ replaced by a query to $B$’s first oracle, and returns the ciphertext to $A$. When $A$ outputs a forgery $(N, A, C)$, adversary $B$ runs the decryption scheme of $\Pi[E]$ on $(N, A, C)$, but with each call to $E_K^{-1}$ replaced by a query to $B$’s oracles. Adversary $B$ returns 1 if $A$ output a valid forgery, and returns 0.
Experiments

Let \( A \) be the ideal variant of \( \Pi \). In conjunction with Lemma 3.1, it follows that the string \( V := \Tag\pi(T, X) \) is uniform. Thus \( \Pr[\mathcal{A} \text{ forges in } H_3] = \Pr[\mathcal{A} \text{ forges in } H_2] \). The probability that \( \mathcal{A} \) can forge in \( H_3 \) is at most \( 2^{-\tau} \). Hence \( \Adv_{\Pi[\pi]}(\mathcal{A}) \leq 2^{-\tau} \) in this case.

Now, suppose that there is an encryption query \((N, A, M')\) such that \( |M'| = |C| - \tau \). (Note that there can be at most one such query, since the attacker is not allowed to re-use a nonce value in two encryption queries.) Let \( C' \) be the corresponding ciphertext output by this encryption query, and let \( C = C_1 \cdots C_{2m} \parallel \tag \) and \( C' = C'_1 \cdots C'_{2m} \parallel \tag' \). If \( C_j = C_j' \) for every \( j \leq 2m \) then \( \Tag \) and \( \Tag' \) must be different and, thus, since \( \Tag \) is deterministic, the forgery is invalid. Otherwise, take the least index \( r \leq m \) such that \( C_{2r-1}C_{2r} \neq C'_{2r-1}C'_{2r} \).

Consider experiments \( P_1, \ldots, P_{m-r+4} \) in Figure 3.11. In \( P_1 \), the adversary has two oracles: \( \Encrypt \) and \( \Decrypt \). The first implements the encryption scheme of \( \Pi[\pi] \), and the second implements the decryption scheme of \( \Pi[\pi] \) but returns false if the decrypted value is \( \bot \) and returns true otherwise.

Let \( S \) be the subset of \( \Permut_{T, n} \) such that for any \( f \in S \) and query \((T, X)\) that \( \Encrypt[\pi](N, A, M') \) makes to \( \pi \), we have \( f(T, X) = \pi(T, X) \). Experiment \( P_2 \) is identical to \( P_1 \), except that in procedure \( \Decrypt \), each time we call \( \Decrypt \) or \( \Tag \) we resample \( \pi \leftarrow S \). Since in the forgery query we do not repeat the tweak of any encryption query other than \((N, A, M')\), and \( \pi \) and \( \pi^{-1} \) are called with distinct tweaks, we have \( \Pr[A \text{ forges in } P_1] = \Pr[A \text{ forges in } P_2] \). In experiment \( P_3 \) we sample \( Y \) uniformly instead of computing \( Y := \Decrypt[\pi^{-1}](T, X, C_{2r-1}C_{2r}) \). Considering lines 14–16 of \( \Auth \) (and the fact that \( \Auth(G_1, G_2) = \text{true} \)) in conjunction with Lemma 3.2, we have \( \Pr[A \text{ forges in } P_2] - \Pr[A \text{ forges in } P_3] \leq \frac{2\Cost(\Pi) + 2}{2^n} \).
Finally, procedure Encrypt includes the corresponding boxed statement, but experiment $H_1$ does not. Each experiment also has a procedure Decrypt, implementing the encryption algorithm of $\Pi[\pi]$, that is not shown for simplicity.

For $j = 1, \ldots, m - r$, experiment $P_{3+j}$ is identical to $P_{2+j}$, except that we sample $Y$ uniformly instead of computing $Y := \text{Dec}^{\pi,-1}_\pi(T, X, C_{2r+2j-1}C_{2r+2j})$. Considering lines 17–19 of Auth (and the fact that $\text{Auth}(G_1, G_2) = \text{true}$) in conjunction with Lemma 3.2, we conclude that $\Pr[A \text{ forges in } P_{2+j}] - \Pr[A \text{ forges in } P_{3+j}] \leq \frac{2\text{Cost}(\Pi) + 2}{2^n}$.

Experiment $P_{m-r+4}$ is identical to $P_{m-r+3}$ except that we sample $V$ uniformly when checking the validity of the forgery instead of computing $V := \text{Tag}^*(T, X)$. Let $X'$ be the state used by Tag in $\text{Encrypt}[\pi](N, A, M')$. If $X[1, n] \neq X'[1, n]$, which happens with probability at least $1 - 2^{-n}$, then applying Lemma 3.2 to lines 20–22 of procedure Auth, we have $\Pr[A \text{ forges in } P_{m-r+3}] - \Pr[A \text{ forges in } P_{m-r+4}] \leq 2^{-r}$. Finally, $\Pr[A \text{ forges in } P_{m-r+4}] \leq 2^{-r}$. Summing up,

$$\text{Adv}_{\Pi[\pi]}^\text{auth}(A) \leq 2^{-r} + \frac{2(m - r + 1)\text{Cost}(\Pi) + 3}{2^n} \leq 2^{-r} + \frac{\ell(\text{Cost}(\Pi) + 2)}{2^n}.$$  

To summarize, Theorems 3.3 and 3.4 show that if the graphs induced by a given scheme $\Pi$ satisfy $\text{Priv}$ and $\text{Auth}$ as defined in Figure 3.6, then $\Pi$ is a secure AE scheme.

### 4 Implementation and Results

We have implemented the $\text{Priv}$ and $\text{Auth}$ algorithms described in Section 3, and used them to synthesize AE schemes. The code is written in OCaml and available at [https://github.com/analoj/ae-generator](https://github.com/analoj/ae-generator).

---

10All results in this section were computed using the code from commit 515959bcf9fa805c8c102aa82772cfc3c35906f7.
We implement an algorithm of $\Pi[T,X,C]$, that is not shown for simplicity. Here $S$ is the set of $f \in \text{Perm}(T,n)$ such that for any query $(T,X)$ that $\text{Encrypt}[\pi](N,A,M')$ makes to $\pi$, it holds that $f(T,X) = \pi(T,X)$.

Our system has two modules: an analysis module that, given graphs corresponding to an AE scheme, verifies whether the scheme is secure, and a synthesis module that synthesizes AE schemes by enumerating candidate AE schemes and using the analysis module to see if they are secure. We describe these components below, where throughout this section, the term graph denotes an unlabeled graph.

**Analyzer.** The analysis module takes as input a representation (in a stack-based language) of the Dec and Tag graphs; the stack-based language makes it easy to both convert the inputs into their respective graphs as well as to synthesize schemes. We first derive a graph for the $\text{Enc}$ algorithm given the graph for the $\text{Dec}$ algorithm, as described below. Given graphs for the $\text{Enc}$, $\text{Dec}$, and $\text{Tag}$ algorithms, we can then run the privacy and authenticity checks described in Figure 3.6 to check security of the scheme. Our analyzer is able to verify simplified variants of OCB [22], XCBC [12], COPA [3], OTR [20], and CCM [10], among others.

**Deriving the $\text{Enc}$ graph.** We implement an algorithm $\text{Reverse}$ that, given a Dec graph, computes a corresponding Enc graph if one exists. The basic idea is to swap the IN and OUT nodes of the input graph (recall that IN and OUT nodes in the Dec graph denote ciphertext blocks and plaintext blocks, respectively, whereas IN and OUT nodes in the Enc graph are flipped), and then selectively reverse the edges to ensure that each node has correct ingoing/outgoing degrees. Deriving the Enc graph is thus simple if there is at most one path from an IN node to an OUT node and these paths do not cross, as in the case of OCB. However, in other schemes, such as OTR, each IN node may have multiple paths to each OUT node. We handle this as described next.

On input $G_1$, let $G_1$ be the undirected graph of $G_1$. In $G_1$, rename IN nodes as OUT nodes, and OUT nodes

![Figure 3.11: Experiments $P_1,\ldots,P_{4+m-r}$ in the proof of Theorem 3.4. Experiment $P_2$ includes the corresponding boxed statement, but $P_1$ does not. Each experiment also has a procedure $\text{Encrypt}[\pi]$, implementing the encryption algorithm of $\Pi[\pi]$, that is not shown for simplicity. Here $S$ is the set of $f \in \text{Perm}(T,n)$ such that for any query $(T,X)$ that $\text{Encrypt}[\pi](N,A,M')$ makes to $\pi$, it holds that $f(T,X) = \pi(T,X)$.](image-url)
as \( \text{IN} \) nodes; let \( G_2 \) be the resulting graph. Reverse then assigns direction to the edges of \( G_2 \) such that each node has correct ingoing/outgoing degrees; the resulting graph \( G_2^- \) is output. (If no assignment is possible, then the output is \( \perp \).

To implement this idea efficiently, we color each node either “red” or “blue”, where red nodes denote nodes that have already been processed, and blue nodes denote unprocessed nodes. Starting from \( G_2 \), we initially color \( \text{IN} \) and \( \text{INI} \) nodes red and all other nodes blue. We repeatedly iterate over the blue nodes until we reach a fixed point, where in each iteration we assign direction to some edges and re-color some nodes red. If a fixed point is reached before all nodes have been colored red, we return \( \perp \); otherwise, we return \( G_2 \), which represents the reversed graph. If the graph \( G_2 \) has \( r \) nodes then we have at most \( r \) iterations with each iteration taking \( O(r) \) time.

In each iteration, we process each blue node \( x \) as follows. Let \( \text{ord}(x) = 2 \) if \( x \) is an \( \text{XOR} \) node, and let \( \text{ord}(x) = 1 \) otherwise. If there are exactly \( \text{ord}(x) \) red neighbors of \( x \) then (1) for each such neighbor \( y \), assign the direction \( y \rightarrow x \), and (2) color \( x \) red. Note that in each step we ensure that the current node \( x \) has the correct ingoing degree if we color it red. We never assign an ingoing edge to \( x \) in any other step. Hence when there are no blue nodes, each node in the directed graph has the correct ingoing/outgoing degrees. See Figure 4.1 for an illustration of Reverse on OTR.

We prove in Appendix A that Reverse is sound; namely, that if running Reverse on a Dec graph produces an Enc graph, then Dec is a correct decryption algorithm for Enc.

As a side note, the Reverse algorithm allows us to easily check if a scheme is inverse-free (i.e., the scheme only uses the forward direction of the TBC), which is important when constructing hardware realizations of AE schemes due to the potential savings in chip space, among other benefits [14, 20]. After running Reverse, we can check if the parent nodes for all the TBC nodes in the Enc and Dec graph are the same; if so, the scheme is inverse-free.

**Synthesizer.** We synthesize schemes as follows. Fixing a Tag graph, we enumerate all possible Dec graphs of a given size, pruning out “uninteresting” schemes such as ones with two (or more) TBC nodes chained together, and feed each pair of (Dec, Tag) graphs to our analysis module. To generate the Dec graph, we start from a graph containing just the \( \text{IN} \) and \( \text{INI} \) nodes, and add nodes and their corresponding edges until the given size bound is reached. If the resulting graph is “well-formed” (i.e., there are no “dangling” edges and no loops), we derive the corresponding Enc graph as discussed above and run the analysis module on the result. Unfortunately, this approach is prohibitively expensive as described, especially as the size bound increases. Thus, we use several optimizations to speed up the process.

Firstly, instead of synthesizing graphs with \( \text{FIN} \) and \( \text{OUT} \) nodes, we replace these with “terminal” nodes.
We only use one INI.

Note that we can view the encryption of a message \( M \) thus pruning out all non-"rate-1" schemes; this removes 95% of the found schemes.

Upon deriving a well-formed graph, we replace the “terminal” nodes with all possible permutations of FIN and OUT nodes and check security of each. Thus we no longer need to explore the search space for each FIN and OUT node; instead, we explore the search space once using a “terminal” node, and later replace the “terminal” node with all possible combinations of FIN and OUT nodes. Likewise, we can apply this same idea to INI and IN nodes by introducing a “start” node.

Secondly, we observe that AE schemes like OCB, COPA, and OTR do not utilize one of the INI nodes in the sense that they simply output the input value directly. Thus, we can remove two nodes from the synthesis by only synthesizing schemes containing one INI and FIN node. The drawback of this optimization is that it misses schemes such as XCBC and CCM which do in fact use both INI nodes; however, it greatly speeds up synthesis. All the results that follow use this optimization. (It would, of course, be possible to synthesize schemes without using this optimization.)

Results. Using the optimizations described above, we ran our synthesizer to find AE schemes with Dec and Enc graphs of sizes between twelve and sixteen (we found no AE schemes with size less than twelve). Note that our synthesizer does not remove duplicate schemes. In addition, there are many "equivalent" schemes in the sense that one is the same as another except with the outputs and/or inputs flipped. We thus developed a heuristic to remove duplicate and "equivalent" schemes as follows. Let \( F(\cdot,\cdot,\cdot) \) be the encrypt operation of a given scheme, where the first argument is the INI input (recall we consider the simplified variant where we only use one INI node) and the other arguments are the IN inputs. Choosing arbitrary but fixed inputs \( X, M_1, \) and \( M_2 \), we compute \( Y^||C_1||C_2 := F(X, M_1, M_2) \) and \( Y'^||C_1'||C_2' := F(X, M_2, M_1) \). We maintain a table of existing ciphertexts; if any of \( Y'C_1C_2, Y'C_2C_1, Y'C_1'C_2' \text{ or } Y'C_2'C_1' \) exists in the table, we discard the scheme as a “duplicate”; otherwise, we add each of these to the table and continue.

Figure 4.2 shows the results. The experiments were run on a commodity laptop; because of the long running time for synthesizing schemes of size sixteen, we stopped the synthesis after three hours for this size. Due to the large number of discovered schemes, we developed two algorithms to prune the result space. The first simply filters out all schemes \( \Pi \) such that \( \text{Cost}(\Pi) > c \) for some integer \( c \). In Figure 4.2 we set \( c = 2 \), thus pruning out all non-“rate-1” schemes; this removes 95% of the found schemes.

Our second algorithm checks whether a scheme is parallelizable, an important criterion for AE schemes. Note that we can view the encryption of a message \( M = M_1 \cdots M_{2m} \) as a single graph constructed from \( m \) Enc graphs \( G_1, \ldots, G_m \), where the FIN nodes of \( G_i \) coincide with the INI nodes of \( G_{i+1} \). We can then assign a "depth" to each node in this graph as follows:

- The INI nodes in \( G_1 \) and the IN nodes in \( \{G_i\} \) get a depth of 0.
- For each node \( x \), let \( t \) be the maximum depth of \( x \)'s parent(s). If \( x \) is a TBC node then \( \text{depth}(x) = t + 1 \); otherwise \( \text{depth}(x) = t \).

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>Unique</th>
<th>“Optimal”</th>
<th>WP</th>
<th>SP</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>13 (0)</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>47 sec</td>
</tr>
<tr>
<td>13</td>
<td>142 (0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4.3 min</td>
</tr>
<tr>
<td>14</td>
<td>582 (2)</td>
<td>171</td>
<td>48 (4)</td>
<td>5</td>
<td>24.2 min</td>
</tr>
<tr>
<td>15</td>
<td>2826 (54)</td>
<td>40</td>
<td>18</td>
<td>6</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>16</td>
<td>3090 (—)</td>
<td>66</td>
<td>25 (4)</td>
<td>1</td>
<td>3 hours*</td>
</tr>
</tbody>
</table>

Total | 6653 | 290 | 98 (8) | 17 |

Figure 4.2: Synthesis results. The first column shows the number of instructions in the Dec graph of the given scheme; the second column the number of secure (and unique) schemes, with the number in parentheses denoting the number of schemes in which the security check fails but we cannot automatically find a concrete attack; the third column the number of (secure) schemes that are “optimal”, i.e., having two DEC nodes per Dec graph; the fourth column the number of “optimal” weakly parallelizable schemes, with the number in parentheses denoting the weakly parallelizable schemes which only use the forward direction of the TBC; the fifth column the number of “optimal” strongly parallelizable schemes; and the final column the total synthesis time, where an asterisk indicates that we halted execution after the given time.
Looking at the results of Figure 4.2, we found thirteen secure AE schemes of size twelve, five of which are strongly parallelizable. Of these schemes, as far as we know, only OCB exists in the literature. In Figure 4.3 we show two of these newly synthesized schemes, along with one which is not parallelizable. (For all the schemes in the figure, encryption is strongly parallelizable; for the third scheme, however, decryption cannot be parallelized.) We implemented all three schemes and compared their performance with that of an optimized implementation of OCB by Krovetz\textsuperscript{11} using AES-NI; see Figure 4.4. (Note that the results in Figure 4.4 are preliminary timing numbers; the purpose of these experiments is to show that our schemes are competitive with, not necessarily better than, OCB.) We find that the encryption procedure for all four schemes is comparable. However, the decryption procedure of the third synthesized scheme is noticeably slower than the others. This is because decryption for this scheme is not parallelizable; namely, to decrypt ciphertext block $C_i$ we need plaintext block $M_{i-1}$.

In addition, among the weakly parallelizable schemes, we found eight schemes which are inverse-free (we found no such schemes for strongly parallelizable schemes). The schemes of size fourteen that we found use one fewer XOR instruction than OTR, the fastest known inverse-free AE scheme we are aware of.

We also ran our attack generation algorithm (cf. Appendix C) over schemes of size 12–15 and found that the number of schemes where no attack could be found closely matched the number of schemes our analysis found secure, thus pointing to the fact that while our analysis is not sound, it appears to capture most secure schemes. In particular, we found a total of 56 schemes which we could neither prove secure nor find a concrete attack for sizes between 12 and 15.

We remark that our tool currently takes a given bound \( S \) and enumerates all schemes in which decryption can be implemented using at most \( S \) instructions. In future work one could consider assigning a cost to different instructions (e.g., letting \texttt{DUP} have cost 0, and letting \texttt{TBC} have cost some fixed multiple of \texttt{XOR}) and enumerating all schemes having at most some given cost.

5 Conclusion

In this work, we present a methodology for automatically proving the security of a large class of authenticated encryption (AE) schemes. Using our approach, we are able to synthesize thousands of schemes, most of which have never been studied in the literature. Among these, we discovered five new schemes which are as “compact” (in terms of the number of instructions per message block), as “efficient” (in terms of the number of blockcipher calls per message block), and as parallelizable as OCB, with competitive performance.

There are several interesting avenues for future work. Further optimizing the synthesis procedure would allow us to generate more schemes. Some of these schemes may have additional properties of interest, such as misuse-resistance [11]; developing techniques for automatically checking schemes for these additional properties would be very useful. Taking a different approach, it would be interesting to see if similar techniques can be applied to more general classes of AE schemes.

Acknowledgments

We thank Samuel Neves for pointing out that on Haswell CPUs, TurboBoost must be disabled to measure the timing accurately. We also thank the anonymous reviewers of CCS 2015 for their suggestion of automatic generation of attacks on schemes that fail our tests.

Work of Jonathan Katz was done for Exelis under contract number N00173-11-C-2045 to NRL. Work of Alex J. Malozemoff was conducted with Government support awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a.

References


A Correctness of Reverse

Recall that Dec is supposed to reverse Enc: namely, for any \( f \in \text{Perm}(\mathcal{T}, n) \), every \( T \in \mathcal{T} \), every initial state \( X \), and any input \( M \), if \( (Y, C) = Enc^f(T, X, M) \) then \( (Y, M) = Dec^{f^{-1}}(T, X, C) \). We call this the
reversal condition. We now prove that Reverse produces a “correct” Enc graph given a Dec graph as input, by showing that the input graph \( G_1^- \) is a reversal of the output graph \( G_2^- \). That is, for any labeling of \( G_2^- \) such that the labels on TBC nodes are distinct positive integers, one can assign a corresponding labeling to \( G_1^- \) such that the resulting algorithms Enc and Dec—that are derived from the corresponding labeled graphs \( G_2 \) and \( G_1 \), respectively—satisfy the reversal condition.

Lemma A.1. If Reverse\((G_1^-)\) outputs \( G_2^- \neq \bot \), then \( G_1^- \) is a reversal of \( G_2^- \).

Proof. We first describe how to label \( G_1^- \) given the labeling for \( G_2^- \). For convenience, we use the same symbol to refer to corresponding nodes in the two graphs. Recall that if a node \( x \) is a TBC node in \( G_2^- \) then \( x \) is also a TBC node in \( G_1^- \). Let \( \ell \) be the label of \( x \) in \( G_2^- \) and let \( y \) be the parent of \( x \) in \( G_2^- \). If \( y \) is also the parent of \( x \) in \( G_1^- \), then assign label \( \ell \) to \( x \) in \( G_1^- \); otherwise assign \( x \) label \( -\ell \).

We prove our result for general graphs, meaning that the number of INI, FIN, IN, and OUT nodes can be arbitrary. Our proof is by induction on the total number \( s \) of TBC, XOR, and DUP nodes in \( G_1^- \). The base case \( s = 0 \) is trivial.

Let \( G_2^- \neq \bot \) denote the output of Reverse\((G_1^-)\). Fix some \( f \in \text{Perm}(T, n) \), and a labeling on \( G_2^- \) such that the labels on TBC nodes are distinct, positive integers. Fix a tweak \( T \in T \), an initial state \( X \) and input \( M \) for the resulting labeled graph \( G_2 \). Evaluate \( G_2 \) on input \((T, X, M)\) using \( f \), and let \( \text{val}(x) \) be the value on node \( x \). Let \( (Y, C) \) be the result of this evaluation. Among the DUP, TBC, and XOR nodes of \( G_1^- \), let \( x_0 \) denote the first node that Reverse\((G_1^-)\) re-colors red. We consider the case that \( x_0 \) is an XOR node; the other cases are similar.

Let \( x_1 \) and \( x_2 \) be the red neighbors of \( x_0 \), and let \( x_3 \) be the other (blue) neighbor of \( x_0 \). Note that \( x_1 \) can’t be an IN node in \( G_1^- \) (and thus an OUT node in \( G_2^- \)), since such a node will be re-colored red only after its unique neighbor turns red, but here \( x_1 \) is already red while \( x_3 \) is still blue. Likewise, \( x_2 \) can’t be an IN node in \( G_1^- \). Then \( x_1 \) and \( x_2 \) must be INI or OUT nodes in \( G_1^- \), because \( x_0 \) is the first among the DUP, XOR, and TBC nodes of \( G_1^- \) that is re-colored red.

Let \( H_1^- \) be the graph obtained from \( G_1^- \) by merging \( x_0, x_1, \) and \( x_2 \) into a single node \( y \). If both \( x_1 \) and \( x_2 \) are INI nodes in \( G_1^- \), then let \( y \) be an INI node; otherwise, let \( y \) be an OUT node. Each node in \( H_1^- \) has correct ingoing/outgoing degrees and there are \( s - 1 \) TBC, XOR, and DUP nodes in \( H_1^- \). Note that \( G_2^- \) can be obtained by (i) running Reverse\((H_1^-)\) to get a graph \( H_2^- \), (ii) splitting the node \( y \) in \( H_2^- \) into the nodes \( x_0, x_1, x_2 \) as in \( G_1^- \), and (iii) changing OUT nodes to IN nodes among the reinstated nodes, and reversing the direction of their attached edges.

By the induction hypothesis, \( H_1^- \) is a reversal of \( H_2^- \). Use the same labeling of \( G_1^- \) and \( G_2^- \) for \( H_1^- \) and \( H_2^- \), respectively, and let \( H_1 \) and \( H_2 \) be the corresponding labeled graphs. In \( H_2 \), assign the value \( \text{val}(z) \) to each INI/IN node \( z \), with \( \text{val}(y) := \text{val}(x_1) \oplus \text{val}(x_2) \), which is also \( \text{val}(x_3) \). If we evaluate \( H_2 \) on tweak \( T \) and the assigned input using \( f \), then the value at each node \( x \) is also \( \text{val}(x) \). Since \( H_1^- \) is a reversal of \( H_2^- \), if we assign value \( \text{val}(x) \) on each INI/IN node \( x \) of \( H_1 \), and evaluate \( H_1 \) on tweak \( T \) and the assigned input using \( f \) and \( f^{-1} \), then the value of each FIN/OUT node \( x \) is also \( \text{val}(x) \). Next, evaluate \( G_1 \) on \((T, X, C)\) using \( f \) and \( f^{-1} \). We consider the following cases.

Case 1: Both \( x_1 \) and \( x_2 \) are INI nodes in \( G_1 \). Then we are repeating the evaluation of \( H_1 \), with the three nodes \( x_0, x_1, \) and \( x_2 \) simulating node \( y \), and assigning value \( \text{val}(x_1) \oplus \text{val}(x_2) = \text{val}(y) \) to \( x_3 \). Thus, in the evaluation of \( G_1 \), the value at each FIN/OUT node \( x \) is also \( \text{val}(x) \), leading to the result \((Y, M)\).

Case 2: One of \( x_1 \) or \( x_2 \) is an OUT node in \( G_1 \). Without loss of generality, suppose that \( x_1 \) is an OUT node. Hence in \( G_1 \), the XOR node \( x_0 \) is the parent of \( x_1 \), and so \( x_2 \) must be an INI node. (Recall that \( x_2 \) must be either INI or OUT, but if it is an OUT node then the XOR node \( x_0 \) has outgoing degree at least two, a contradiction.) Then we are repeating the evaluation of \( H_1 \) to get \( \text{val}(y) = \text{val}(x_1) \oplus \text{val}(x_2) \) on node \( y \), inserting node \( x_2 \) with value \( \text{val}(x_2) \), and computing \( \text{val}(x_2) \oplus \text{val}(y) = \text{val}(x_1) = \text{val}(x_0) \) at nodes \( x_1 \) and \( x_0 \). Thus, for the evaluation on \( G_1 \), the value at each FIN/OUT node \( x \) is also \( \text{val}(x) \), leading to result \((Y, M)\). □

B Arbitrary Message Lengths
for a tweakable blockcipher as before, while \(\Delta\) is an incoming state block.

Message length

While arbitrary length messages can be handled by reversibly padding out to a multiple of 2 bits. The trapezoids represent truncation and padding with 10.

Our template in Section 3.1 only handles messages in \((\{0,1\}^2)^n\), yet in practice schemes should support messages of arbitrary length. While arbitrary length messages can be handled by reversibly padding out to a multiple of 2n bits, we show here how to extend our template to handle messages in \{0,1\}^* more efficiently.

The key for an AE scheme will now be a pair \((K, \Delta)\). The first component \(K\) will be used as the key for a tweakable blockcipher as before, while \(\Delta\) is an n-bit string used to handle the final (fragmentary) block. In Figure B.1, we show algorithms \(\sharp\text{Enc}\) and \(\sharp\text{Dec}\) that operate on strings of length less than 2n bits (see Figure B.2 for a graphical depiction); they are based on how OTR [20] handles fragmentary blocks. Figure B.3 shows how to use \(\sharp\text{Enc}\) and \(\sharp\text{Dec}\) (along with \text{Enc}, \text{Dec}, and \text{Tag}) to construct encryption and decryption algorithms for arbitrary length messages. Note that the code in Figure 3.1 is a special case of that in Figure B.3 when the message length is a multiple of 2n bits. Below, we show that the extended template indeed leads to secure schemes.
Theorem B.1. Let \( \Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D}) \) be an AE scheme on \( \{0,1\}^* \) for which \( \text{Priv}(G_1^-, G_2^-) = \text{true} \), where \( G_1^- \) and \( G_2^- \) are the unlabeled graphs for \( \mathcal{E} \) and \( \mathcal{D} \), respectively. Then for any adversary \( \mathcal{A} \), there is an adversary \( \mathcal{B} \) with \( \text{Adv}_{\Pi[E]}^{\text{priv}}(\mathcal{A}) \leq \text{Adv}_{\mathcal{E}}^{\text{prp}}(\mathcal{B}) \). Adversary \( \mathcal{B} \) has the same running time as \( \mathcal{A} \) and makes at most \( (\text{Cost}(\Pi) + 1) \cdot \sigma/2 \) queries, where \( \sigma \) is the number of (full) message blocks in the queries of \( \mathcal{A} \).

Proof. Adversary \( \mathcal{B} \) runs \( \mathcal{A} \) as follows. For each of \( \mathcal{A} \)'s queries \((N, A, M)\), adversary \( \mathcal{B} \) runs the encryption scheme \( \Pi[E] \) on \((N, A, M)\) with each call to \( E_{\mathcal{K}} \) replaced by a query to \( \mathcal{B} \)'s oracle, and returns the ciphertext to \( \mathcal{A} \). Finally, \( \mathcal{B} \) outputs the same guess as \( \mathcal{A} \). To compute the the number of queries of \( \mathcal{B} \), note that \( \text{Cost}(\Pi) \geq 2 \) and \( E_{\mathcal{K}} \) decs on message \( M \), makes \( |M/n| \leq |M/n| \cdot \text{Cost}(\Pi)/2 \) calls to the tweakable blockcipher or its inverse.

Let \( \Pi[\pi] \) be the ideal variant of \( \Pi[E] \), where \( E_{\mathcal{K}} \) calls are replaced by corresponding queries to \( \pi \), with \( \pi \leftarrow \text{Perm}(T,n) \). It suffices to show that \( \text{Adv}_{\Pi[\pi]}^{\text{priv}}(\mathcal{A}) = 0 \). Consider experiments \( H_1 \sim H_5 \) in Figure B.4. The adversary has oracle access to the encryption scheme of \( \Pi[\pi] \) in experiment \( H_1 \), and oracle access to \( \S(\cdot, \cdot) \) in experiment \( H_5 \). Experiment \( H_2 \) is identical to \( H_1 \), except that we re-sample \( \pi \leftarrow \text{Perm}(T,n) \) each time we use \( \mathcal{E} \) or \( \mathcal{D} \). Since a tweak to \( \pi \) is never repeated, \( \text{Pr}[H_4^A \Rightarrow \text{true}] = \text{Pr}[H_2^A \Rightarrow \text{true}] \). In experiment \( H_3 \), instead of calling \( \mathcal{D}^\pi(T,X) \) to get the tag, we sample the tag at random. We then unroll procedure \( \mathcal{E} \) and instead of calling the dead code that processes the state \( X \). By applying Lemma 3.1 to lines 02–03 of procedure \( \text{Priv} \), the string \( V := \mathcal{D}^\pi(T,X) \) is uniform and so experiments \( H_2 \) and \( H_3 \) are identical. Next, in experiment \( H_4 \), we directly sample \( C_{2m+1} \leftarrow \{0,1\}^{M_{2m+1}} \) instead of processing via \( \pi \). Since we never call \( \pi \) on the same tweak twice, each output of \( \pi(\cdot, \cdot) \) is an independent, uniform string, and thus experiments \( H_3 \) and \( H_4 \) are identical. Finally, experiment \( H_5 \) is identical to \( H_4 \), except that instead of calling \( \mathcal{E}^\pi(T,X,M_{2l-1}M_{2l}) \) to get the blocks \( C_{2l-1}C_{2l} \) of the ciphertext, we sample them uniformly. By applying Lemma 3.1 to lines 04–05 of procedure \( \text{Priv} \), the output blocks of \( \mathcal{E}^\pi(T,X,M_{2l-1}M_{2l}) \) are uniform and independent (even conditioned on all prior ciphertext blocks). Hence \( H_4 \) and \( H_5 \) are identical, and \( \text{Adv}_{\Pi[\pi]}^{\text{priv}}(\mathcal{A}) = \text{Pr}[H_4^A \Rightarrow \text{true}] - \text{Pr}[H_4^A \Rightarrow \text{true}] = 0 \).
\begin{figure*}

\begin{verbatim}
proc ENCRYPT[\Delta, \pi](N, A, M) // Experiments $H_2$
M_1 \cdots M_{2m}, M_{2m+1} := M; X := 0^{2n}; v := 1
// |M_i| = n for i \leq 2n, and |M_{2m+1}| < 2n
for i = 1 to m do
  T := (N, A, v); \pi \leftarrow \text{Perm}(T, n)
  (Y, C_{2i-1}C_{2i}) := Enc^\pi(T, X, M_{2i-1}M_{2i})
  v := v + Cost(\Pi); X := Y
  (X, C_{2m+1}) := \#Enc^\pi((N, A, v), X, M_{2m+1})
  v := v + |C_{2m+1}/n|; \pi \leftarrow \text{Perm}(T, n)
  T := (N, A, 1 - v); V := Tag^\pi(T, X)
return C_1 \cdots C_{2m}C_{2m+1} \parallel V[1, \tau]
\end{verbatim}

\begin{verbatim}
proc ENCRYPT[\Delta, \pi](N, A, M) // Experiment $H_3$
M_1 \cdots M_{2m}, M_{2m+1} := M; X := 0^{2n}; v := 1
// |M_i| = n for i \leq 2n, and |M_{2m+1}| < 2n
for i = 1 to m do
  T := (N, A, v); \pi \leftarrow \text{Perm}(T, n)
  (Y, C_{2i-1}C_{2i}) := Enc^\pi(T, X, M_{2i-1}M_{2i})
  v := v + Cost(\Pi); X := Y
  T := (N, A, v); L := (N, A, v + 1)
if M_{2m+1} = \varepsilon then C_{2m+1} := \varepsilon
else |M_{2m+1}| > n then
  M_1^gM_2^g := M_{2m+1}; // |M_1| = n
  V := \pi^T(M_1^g); C_2^g := V[1, |M_1^g|] \parallel M_2^g
  C_2^l := \pi^l(C_2^g 10^n) \parallel M_1^g; C_{2m+1} := C_2^lC_2^g
else C_{2m+1} := \pi^T(0^n)[1, |M_{2m+1}|] \parallel M_{2m+1}
V \leftarrow \{0, 1\}^n
return C_1 \cdots C_{2m}C_{2m+1} \parallel V[1, \tau]
\end{verbatim}

\begin{verbatim}
proc ENCRYPT[\Delta, \pi](N, A, M) // Experiments $H_4$, $H_5$
M_1 \cdots M_{2m}, M_{2m+1} := M; X := 0^{2n}; v := 1
// |M_i| = n for i \leq 2n, and |M_{2m+1}| < 2n
for i = 1 to m do
  T := (N, A, v); \pi \leftarrow \text{Perm}(T, n)
  (Y, C_{2i-1}C_{2i}) := Enc^\pi(T, X, M_{2i-1}M_{2i})
  C_{2i-1}C_{2i} \leftarrow \{0, 1\}^{2n}
  v := v + Cost(\Pi); X := Y
  T := (N, A, v); L := (N, A, v + 1)
  C_{2m+1} \leftarrow \{0, 1\}^{\beta_{2m+1}}; V \leftarrow \{0, 1\}^n
return C_1 \cdots C_{2m}C_{2m+1} \parallel V[1, \tau]
\end{verbatim}

\textbf{Figure B.4:} Experiments $H_1$–$H_5$ in the proof of Theorem B.1. Experiments $H_2$ and $H_5$ include the corresponding boxed statements, but $H_1$ and $H_4$ do not.

\textbf{Theorem B.2.} Let $\Pi[\mathcal{E}] = (K, E, D)$ be an AE scheme on $\{0, 1\}^*$ such that $\text{Auth}(G_1^-, G_2^-) = \text{true}$, where $G_1^-$, $G_2^-$ are the unlabeled graphs for algorithms Dec and Tag of $\Pi$, respectively. Then for any adversary $A$, there is an adversary $B$ with $\text{Adv}_{\Pi[\mathcal{E}]}^\text{auth}(A) \leq 2^{-\ell} + \epsilon(Cost(\Pi) + 2^n) + \text{Adv}_{\Pi[\mathcal{E}]}^\text{prp}(B)$, where $\ell$ is the number of (full) blocks in the forgery output by $A$. Adversary $B$ has the same running time as $A$ and makes at most $(\text{Cost}(\Pi) + 1) \cdot \sigma/2$ queries, where $\sigma$ is the total number of (full) message blocks in the queries of $A$.

\textbf{24}
\begin{verbatim}
proc Decrypt[π](N,A,C)  // Experiments H1, H2
if |C| \neq \tau (mod 2n) then return ⊥
C_1 \cdots C_{2n} C_{2n+1} \parallel tag := C
// |C_i| = n for i ≤ 2m, |C_{2m+1}| < 2n, and |tag| = \tau
X := 0^{2n}; v := 1
for i = 1 to ℓ do
    T := (N,A,v); (Y,M_{2i−1}M_{2i}) := Dec^\tau,N,A,T (T,X,C_{2i−1}C_{2i})
    v := v + Cost(Π); X := Y
(\pi, M_{2m+1}) := \#Enc^\tau((N,A,v), X, C_{2m+1})
π ← Perm(T,n); v := v + ⌈|M_{2m+1}|/n⌉ − 1
T := (N,A,1−v); V := Tag^\tau(T,X)
if tag \neq V[1,\tau] then return ⊥
return M_1 \cdots M_{2m} M_{2m+1}
\end{verbatim}

\begin{verbatim}
proc Decrypt[π](N,A,C)  // Experiment H3
if |C| \neq \tau (mod 2n) then return ⊥
C_1 \cdots C_{2n} C_{2n+1} \parallel tag := C
// |C_i| = n for i ≤ 2m, |C_{2m+1}| < 2n, and |tag| = \tau
X := 0^{2n}; v := 1
for i = 1 to ℓ do
    T := (N,A,v); (Y,M_{2i−1}M_{2i}) := Dec^\tau,N,A,T (T,X,C_{2i−1}C_{2i})
    v := v + Cost(Π); X := Y
(\pi, M_{2m+1}) := \#Enc^\tau((N,A,v), X, C_{2m+1}); V ← \{0,1\}^n
if tag \neq V[1,\tau] then return ⊥
return M_1 \cdots M_{2m} M_{2m+1}
\end{verbatim}

Figure B.5: Experiments H1–H3 in the proof of Theorem B.2. Experiment H2 includes the corresponding boxed statement, but experiment H1 does not. Each experiment also has a procedure Encrypt[π], implementing the encryption algorithm of Π[π], that is not shown for simplicity.

Proof. Adversary B runs A. For each of A’s encryption queries, B runs the encryption scheme of Π[E] but with each call to E_K replaced by a query to B’s first oracle, and returns the ciphertext to A. When A outputs a forgery (N,A,C), adversary B runs the decryption scheme of Π[E] on (N,A,C), but with each call to E_K/E_K−1 replaced by a query to B’s oracles. Adversary B returns 1 if A output a valid forgery, and returns 0 otherwise. Let |π| be the ideal variant of Π[E], where calls to E_K/E_K−1 are replaced by queries to π/π−1, respectively, with π ← Perm(T,n). It suffices to show that Adv^{auth}_{Π[π]}(A) ≤ 2−τ + ℓ(Cost(Π) + 2)/2^n.

Consider experiments H1–H3 in Figure B.5. In H1, the adversary has oracle access to the encryption and decryption schemes of Π[π]. Experiment H2 is identical to H1, except that when running the decryption algorithm, we re-sample π ← Perm(T,n) before using it in Tag. Experiment H3 is identical to H2, except that instead of using Tag to generate the tag, we sample the tag uniformly.

Let (N,A,C) be the forgery output by A. Suppose that there is no encryption query (N,A,M') with ⌈|M'|/n⌉ = ⌊(|C| − τ)/n⌋. Since decryption of the forgery query involves calling Tag with a tweak that has never been used before, we have Pr[A forges in H1] = Pr[A forges in H2]. An application of Lemma 3.1 to lines 12–13 of Auth, shows that the string V := Tag^\tau(T,X) is uniform. Thus Pr[A forges in H2] is equal to Pr[A forges in H3], which is in turn at most 2−τ. Hence Adv^{auth}_{Π[π]}(A) ≤ 2−τ.

Now, suppose that there is an encryption query (N,A,M') such that ⌈|M'|/n⌉ = ⌊(|C| − τ)/n⌋. (Note that there can be at most one such query, since the attacker is not allowed to re-use a nonce value in two encryption queries.) Let C' be the corresponding ciphertext output by this encryption query. We say that a ciphertext is complete if its length (excluding the tag) is a multiple of n. First consider the case that one of C and C' is complete and the other is incomplete. For example, suppose that C is complete and C' is incomplete. Let S be the subset of Perm(T,n) such that for any f ∈ S and for any query (T,X) that
Figure B.6: Experiments $P_1, \ldots, P_{4m-r}$ in the proof of Theorem B.2. Experiment $P_2$ includes the corresponding boxed statement, but $P_1$ does not. Each experiment also has a procedure $\text{Encrypt}[^{\Delta, \pi}](N, A, C)$, implementing the encryption algorithm of $\Pi[^{\pi}]$, that is not shown for simplicity. Here $S$ is the set of $f \in \text{Perm}(T, n)$ such that for any query $(T, X)$ that $\text{Encrypt}[^{\Delta, \pi}](N, A, M')$ makes to $\pi$, $f(T, X) = \pi(T, X)$.

We have (i) the outputs of the encryption oracle are independent of the key $\Delta$, and (ii) if $\pi$ is deterministic, the forgery is invalid.

Case 1: $C_j = C'_j$ for every $j \leq 2m + 1$. Then tag and tag' must be different and thus, since $\text{Tag}$ is deterministic, the forgery is invalid.

Case 2: $C_j = C'_j$ for every $j \leq 2m$, but $C_{2m+1} \neq C_{2m+1}'$. Let $X$ be the state that $\text{Tag}$ receives on querying $(N, A, C)$, and let $X'$ be the corresponding state on querying $(N, A, C')$. Then $X[1, n] = X'[1, n] \not\equiv C_{2m+1} \oplus C_{2m+1}' \neq X'[1, n]$. By applying Lemma 3.2 to lines 20–22 of procedure $\text{Auth}$, the tag of $C$ is within
Case 3: There is $r \leq m$ such that $C_{2r-1}C_{2r} \neq C'_{2r-1}C'_{2r}$, and $C_j = C'_j$ for every $j < 2r - 1$. Consider experiments $P_1, \ldots, P_{m-r+4}$ in Figure B.6. In $P_1$, the adversary has two oracles: $\text{Encrypt}$ and $\text{Decrypt}$. The first implements the encryption scheme of $\Pi[\pi]$, and the second implements the decryption scheme of $\Pi[\pi]$ but returns false if the decrypted value is $\bot$ and returns true otherwise. Experiment $P_2$ is identical to $P_1$, except that in procedure $\text{Decrypt}$, each time we call $\text{Dec}$ or $\text{Tag}$ we resample $\pi \leftarrow S$. Since in the forgery query we do not repeat the tweak of any encryption query other than $(N,A,M')$, and $\pi$ and $\pi^{-1}$ are called with distinct tweaks, we have $\Pr[A \text{ forges in } P_1] = \Pr[A \text{ forges in } P_2]$. In experiment $P_3$ we sample $Y$ uniformly instead of computing $Y := \text{Dec}^{\pi^{-1}}(T,X,C_{2r-1}C_{2r})$. Applying Lemma 3.2 to lines 14–16 of procedure $\text{Auth}$, we have $\Pr[A \text{ forges in } P_2] - \Pr[A \text{ forges in } P_3] \leq \frac{2\text{Cost}(\Pi)+2}{2^n}$.

For $j = 1, \ldots, m-r$, experiment $P_{3+j}$ is identical to $P_{2+j}$, except that we sample $Y$ uniformly instead of computing $Y := \text{Dec}^{\pi^{-1}}(T,X,C_{2r-2j}C_{2r-2j+2})$. Applying Lemma 3.2 to lines 17–19 of procedure $\text{Auth}$, we can conclude that $\Pr[A \text{ forges in } P_{2+j}] - \Pr[A \text{ forges in } P_{3+j}] \leq \frac{2\text{Cost}(\Pi)+2}{2^n}$.

Experiment $P_{m-r+3}$ is identical to $P_{m-r+4}$ except that we sample $Y$ uniformly when checking validity of the forgery instead of computing $V := \text{Tag}^{\pi}(T,X)$. Let $X$ be the incoming state for $\pi \text{Dec}$ on querying $(N,A,C)$ and let $Y$ be the outgoing state. Let $X'$ and $Y'$ be the corresponding states on querying $(N,A,C')$ respectively. Then $X$ is a random string, independent of prior ciphertexts and $C,X,Y$. Moreover, $Y[1,n] = X[1,n] \oplus M'_{2m+1} \oplus C'_{2m+1} \oplus C_{2m+1}$, which is independent of $Y'[1,n]$. Hence $Y[1,n] \neq Y'[1,n]$ with probability at least $1 - 2^{-n}$. Note that $Y$ and $Y'$ are the state given to $\text{Tag}$ in $\text{Encrypt}\{\Delta, \pi\}(N,A,C)$ and $\text{Encrypt}\{\Delta, \pi\}(N,A,C')$ respectively. If $Y[1,n] \neq Y'[1,n]$, by applying Lemma 3.2 to lines 20–22 of procedure $\text{Auth}$, we have $\Pr[A \text{ forges in } P_{m-r+3}] - \Pr[A \text{ forges in } P_{m-r+4}] \leq \frac{2}{\pi^2}$. Finally, $\Pr[A \text{ forges in } P_{m-r+4}] \leq \frac{\pi}{2^\tau}$. Summing up,

$$\text{Adv}_{\Pi[\pi]}^\text{auth}(A) \leq 2^{-2\tau} + \frac{(m-r+1)(\text{Cost}(\Pi) + 1) + 3}{2^n} \leq 2^{-\tau} + \frac{\ell(\text{Cost}(\Pi) + 2)}{2^n}. \tag*{\square}$$

C Generating Attacks Automatically

In this section we show how to automatically generate attacks on schemes that do not pass the $\text{Priv}$ and $\text{Auth}$ tests. The attacks are sound, but it does not mean that schemes that generate no attack are secure. We consider AE schemes that use the extension in Appendix B to handle fragmentary strings. Below, we categorize the attacks based on the graphs ($\text{Tag}$, $\text{Enc}$, and $\text{Dec}$) in which the schemes fail to pass the required tests. We begin by describing a simple property on graphs, which is a cornerstone to our attacks.

A simple linear relation. Let $G = (d,r,F,P,L)$ be a graph and let $E$ be a tweakable blockcipher with key space $K$. Let $S = \{v_1, \ldots, v_s\} \subseteq \{1, \ldots, r\}$ be the set of all $\text{IN}$, $\text{IN}^{-1}$, and $\text{TBC}$ nodes. Without loss of generality, assume that $v_j = j$ for every $j \leq d$. Then for any node $i$, there are constants $a_1, \ldots, a_s \in \{0,1\}$ such that the $n$-bit value of node $i$ is the linear combination of the values at nodes $v_1, \ldots, v_s$ with coefficients $a_1, \ldots, a_s$ respectively. That is, for any $Z_1, \ldots, Z_d \in \{0,1\}^n$, any key $K \in K$, and any tweak $T$, if $(Z_1, \ldots, Z_r) := \text{Eval}^{E_K,F_K}(G,T,Z_1, \ldots, Z_d)$ then $Z_i = a_1 \cdot Z_{v_1} + \cdots + a_s \cdot Z_{v_s}$, where $1 \cdot X = X$ and $0 \cdot X = 0^n$.

To justify this claim, we give an induction proof on $i$. If $i \leq d$ then let $a_j = 1$ if $v_j = i$, and let $a_j = 0$ otherwise. The claim trivially holds for this base case. Suppose that the claim holds for $1, \ldots, i-1$. We now show that it holds for $i$ as well.

- If $i$ is a TBC node then let $a_j = 1$ if $v_j = i$ and let $a_j = 0$ otherwise.
- If $i$ is a FIN/DUP/OUT node then let $p$ be the parent of $i$. Due to the induction hypothesis, there are constants $b_1, \ldots, b_s$ such that the claim holds for $p$. Let $a_j = b_j$ for every $j \leq s$.

27
If $i$ is an XOR node then let $p_1$ and $p_2$ be the parents of $i$. Due to the induction hypothesis, there are constants $b_1, \ldots, b_s$ such that the claim holds for $p_1$, and constants $c_1, \ldots, c_s$ such that the claim holds for $p_2$. Let $a_j = b_j + c_j$ for every $j \leq s$.

This completes the proof.

We can thus conclude that node $i$ can be represented by a corresponding vector $(a_1, \ldots, a_s)$. Note that the above inductive proof also gives an algorithm to identify the representative vectors of all nodes.

**Attacks based on a Tag graph.** There are two tests that Priv and Auth perform on a Tag graph. First, in lines 02–03 (and also lines 12–13) of Figure 3.6, we check if the output value of the graph is random for a fresh TBC tweak. If this test fails, then the representative vector $(a_1, a_2, a_3)$ of the FIN node must have $a_3 = 0$, since this coordinate corresponds to the TBC node (here we assume that Tag has only one TBC instruction). We consider the following cases.

**Case 1:** $(a_1, a_2) = (0, 0)$, meaning that the output value of the Tag graph is always $0^n$, regardless of the input. This immediately leads the following privacy attack: query $0^{2n}$ and return 1 if the tag of the answer is $0^s$, and return 0 otherwise. The adversary wins with advantage at least $1 - 2^{-s}$.

**Case 2:** $(a_1, a_2)$ is $(1, 0)$ or $(0, 1)$. Without loss of generality, assume that $(a_1, a_2) = (1, 0)$, meaning that the output value of $G_1$ is always the same as the value of the first INI node. This is a vulnerability if the first input value of the Tag graph is a linear combination of the plaintext blocks, as in OCB or OTR. To determine this linearity, we check if in the Enc graph, the representative vector $(b_1, \ldots, b_s)$ of the first FIN node satisfies $b_j = 0$ for every $j > 4$. If this happens then we can also launch the privacy attack in Case 1. In the Enc graph, the values of the IN and INI nodes are $0^n$, and thus the value at the first FIN node must be $0^s$ as well. Hence the output value of the Tag graph is also $0^n$. The adversary wins with advantage at least $1 - 2^{-s}$.

**Case 3:** $(a_1, a_2) = (1, 1)$, meaning that the output value of the Tag graph is always the same as the XOR of the values at the two INI nodes. This is a vulnerability if this XOR value is a linear combination of the plaintext blocks. We check if in the Enc graph, the representative vectors $(b_1, \ldots, b_s)$ and $(c_1, \ldots, c_s)$ of the two FIN nodes satisfy $b_j = c_j$ for every $j > 4$. If this happens then we can also launch the privacy attack in Case 1. The adversary wins with advantage at least $1 - 2^{-s}$.

In the second test, in lines 20–22 of Figure 3.6, we check if the output values are independent when we evaluate the Tag graph twice on the same tweak but different inputs. (Actually, in the test, the two inputs must differ only in the first half.) Without loss of generality, assume that the Tag graph passes the test at line 02–03. Then the representative vector $(a_1, a_2, a_3)$ of the OUT node must have $a_3 = 1$. Let $p$ be the parent of the TBC node, and let $(d_1, d_2, d_3)$ be the representative vector of $p$. Then $d_3 = 0$ because the Tag graph has only a single TBC node, and this node is not an ancestor of $p$. We consider the following cases.

**Case 1:** $d_1 = 0$, meaning that if we evaluate the Tag graph twice, on the same tweak, such that the inputs agree on the second half, then the two results agree on the value at node $p$. We then check if $a_1 = 0$. If this happens then in the two evaluations above, they also agree on the value at the OUT node. We now give an authenticity attack. First query $0^n$ to get answer $C \parallel T$, with $|C| = n$. (Note that this query is a fragmentary string.) Next produce $C^* \parallel T$ as a forgery, for any arbitrary $C^* \neq C$. In both the encryption query and the forgery query, the values at the second INI node of the Tag graph are $0^n$. Hence tag $T$ is also a valid tag for $C^*$, and thus the adversary wins with advantage 1.

**Case 2:** $d_1 = d_2 = 1$. We first check if $a_1 \oplus a_1 = 0$. If this happens then the output value of the Tag graph depends solely on the XOR of the values at the two INI nodes. This will be a vulnerability if the XOR value can be linearly determined from the ciphertext blocks. We check if in the Dec graph, the representative vectors $(x_1, \ldots, x_s)$ and $(y_1, \ldots, y_s)$ of the two FIN nodes satisfy $x_j = y_j$ for every $j > 4$. If this happens then we launch the following authenticity attack. First query $0^{2n}$ to get ciphertext $C_1 C_2 \parallel T$, with $|C_1| = |C_2| = n$. Recall that $(x_3, x_4)$ and $(y_1, y_4)$ are coordinates corresponding to the ciphertext blocks in the FIN nodes of the Dec graph. Let $b = x_3 \oplus x_4 \oplus y_3 \oplus y_4$. If $b = 0$ then output $(C_1 \oplus 1^n) \parallel (C_2 \oplus 1^n) \parallel T$ as
a forgery. Note that in the encryption query, the XOR of the values at the two INI nodes of the Tag graph is
\[ x_3 \cdot (C_1 \oplus 1^n) \oplus x_4 \cdot (C_2 \oplus 1^n) \oplus y_3 \cdot (C_1 \oplus 1^n) \oplus y_4 \cdot (C_2 \oplus 1^n) = x_3 \cdot C_1 \oplus x_4 \cdot C_2 \oplus y_3 \cdot C_1 \oplus y_4 \cdot C_2. \]
If \( b = 1 \) then one of \( x_3 \oplus y_3 \) and \( x_4 \oplus y_4 \) must be 0 while the other is 1. Without loss of generality, assume that \( x_3 \oplus y_3 = 0 \). Then output \((C_1 \oplus 1^n) \parallel C_2 \parallel T\) as a forgery. Now in the forgery query, the XOR of the values at the two INI nodes of the Tag graph is
\[ x_3 \cdot (C_1 \oplus 1^n) \oplus x_4 \cdot C_2 \oplus y_3 \cdot (C_1 \oplus 1^n) \oplus y_4 \cdot C_2 = x_3 \cdot C_1 \oplus x_4 \cdot C_2 \oplus y_3 \cdot C_1 \oplus y_4 \cdot C_2. \]
Hence, regardless of the value of \( b \), the encryption query and the forgery query always agree on the XOR of the values at the two INI nodes of the Tag graph. The adversary thus wins with advantage 1.

Note that the case \((d_1, d_2) = (1, 0)\) cannot happen, because it passes the test at line 20–22: if the two inputs of the Tag graph differ in the first half then their values at the TBC node are independent random strings and so are the values at the OUT nodes.

**Attacks based on an Enc graph.** There is a single test that Priv performs on an Enc graph. At lines 04–06 of Figure 3.6, we check if the output values of the graph are random and independent, for a fresh TBC tweak. If the Enc graph fails this test, let \((b_1, \ldots, b_n)\) and \((c_1, \ldots, c_n)\) be the representative vectors of the two OUT nodes of the Enc graph. We consider the following cases.

**Case 1:** \( b_j = 0 \) for every \( j > 4 \), meaning that the value of the first OUT node can be linearly determined from the values at the IN and INI nodes. We launch the following privacy attack: query \( 0^{2n} \) to get the answer \( C \), and then output 1 if the first \( n \)-bit block of \( C \) is \( 0^n \), and output 0 otherwise. The adversary wins with advantage \( 1 - 2^{-\tau} \).

**Case 2:** \( c_j = 0 \) for every \( j > 4 \). This case is similar to Case 1.

**Case 3:** \( b_j = c_j \) for every \( j > 4 \). Then the XOR of the values at two OUT nodes can be linearly determined from the values at the IN and INI nodes. We launch the following privacy attack. First query \( 0^{2n} \) to get the answer \( C_1C_2T \), with \(|C_1| = |C_2| = n\). Next, output 1 if \( C_1 = C_2 \), and output 0 otherwise. The adversary wins with advantage \( 1 - 2^{-\tau} \).

**Attacks based on a Dec graph.** There are two tests that Auth performs on a Dec graph. First, in lines 14–16 of Figure 3.6, we evaluate the graph twice on the same tweak with distinct inputs that agree on the values of the INI nodes to check if the values at the first FIN node are random and independent.

A possible way to fail this test is that the value in the first FIN node depends only on at most a single IN node. We thus check if both IN nodes are ancestors of the first FIN node. If this does not happen then without loss of generality suppose that the first IN node is not an ancestor of the first FIN node. We can launch the following authenticity attack. First query \( 0^{2n} \) to get \( C_1C_2T \), with \(|C_1| = |C_2| = n\). Next, output \((C_1 \oplus 1^n) \parallel C_2 \parallel T\) as the forgery attempt. The adversary wins with advantage 1.

Now suppose that both IN nodes are ancestors of the first FIN node. An example of such a scheme that still fails the test is the insecure variant of OTR as illustrated in Figure 3.8. We can exploit this if the Dec graph satisfies the following additional constraints:

(i) There are \( b_1, b_2, c_1, c_2 \in \{0, 1\} \) such that \((b_1, b_2) \neq (0, 0)\) and XORing \( b_1 \cdot \Delta \) and \( b_2 \cdot \Delta \) to the values of the two IN nodes, respectively, while keeping the same values of the INI nodes, results in \((c_1 \cdot \Delta)\)- and \((c_2 \cdot \Delta)\)-change in the values of the first and second FIN nodes, respectively. For example, in the insecure variant of OTR, (i) holds for \( b_1 = c_1 = 1 \) and \( b_2 = c_2 = 0 \).

(ii) There exist \( a_1, a_2 \in \{0, 1\} \) such that for any \( \Delta \in \{0, 1\}^n \), XORing \( c_1 \cdot \Delta, c_2 \cdot \Delta, a_1 \cdot \Delta, a_2 \cdot \Delta \) to the values of the INI nodes and IN nodes, respectively, will not change the value of the first FIN node. For example, in the insecure variant of OTR, (ii) holds for \( a_1 = 1 \) and \( a_2 = 0 \).

To check for these properties, we do the following:
• Consider all \((r_1, r_2) \in \{(1, 0), (0, 1), (1, 1)\}\). If all TBC ancestors \((x_1, \ldots, x_s)\) of at least one FIN node satisfy \(x_3 r_1 = x_4 r_2\) then let \(b_1 = r_1\) and \(b_2 = r_2\).

• Let \((u_1, \ldots, u_s)\) and \((w_1, \ldots, w_s)\) be the representative vectors of the first and second FIN nodes, respectively. Let \(c_1 = u_3 b_1 \oplus u_4 b_2\) and \(c_2 = w_3 b_1 \oplus u_4 b_2\).

• Consider all \(t_1, t_2 \in \{0, 1\}\). If \(c_1 u_1 \oplus c_2 u_2 = t_1 u_3 \oplus t_2 u_4\), and all TBC ancestors \((x_1, \ldots, x_s)\) of the first FIN node satisfy \(c_1 x_1 \oplus c_2 x_2 = t_1 x_3 \oplus t_2 x_4\), then let \(a_1 = t_1\) and \(a_2 = t_2\).

If in the first check, there are multiple pairs \((r_1, r_2)\) then we find corresponding \((t_1, t_2)\) for each such pair, but only choose a tuple \((r_1, r_2, t_1, t_2)\) to determine \((a_1, a_2, b_1, b_2, c_1, c_2)\).

If those checks pass, we launch the following authenticity attack. Query \(0^4 n\) to get answer \(C_1 C_2 C_3 C_4 T\), where \(|C_1| = |C_2| = |C_3| = |C_4| = n\). Then output \((C_1 \oplus b_1 \cdot 1^n) \parallel (C_2 \oplus b_2 \cdot 1^n) \parallel (C_3 \oplus a_1 \cdot 1^n) \parallel (C_4 \oplus a_2 \cdot 1^n) \parallel T\) as the forgery attempt. The adversary wins with advantage 1.

In the second test, in lines 20–22 of Figure 3.6, we evaluate the graph twice, on the same tweak, and check if the values at the first FIN node are random and independent, where in one input, the value at the first INI is chosen at random. If the Dec graph fails this test, we’ll determine if the value at the first FIN node can be computed from just the input and the tweak. To achieve this, initially, in the Dec graph, we color the two IN nodes red while all other nodes are blue. Our goal is to recolor some blue nodes of the graphs to red, so that the value of any red node can be computed from the tweak and the input without knowing the values at the INI nodes. We’ll traverse the graph in an arbitrary topological order. At each node, if it’s a FIN/OUT/DUP/TBC node and its parent is red then color it red. If it’s an XOR node then let \((y_1, \ldots, y_s)\) be the representative vector of this node. If the corresponding nodes for all coordinates \(i\) such that \(y_i = 1\) are red then color the current node red. When the algorithm terminates, if the first FIN node is red then we launch the following authenticity attack. First query \(0^4 n\) to get answer \(C_1 C_2 C_3 C_4 T\), where \(|C_1| = |C_2| = |C_3| = |C_4| = n\). Next, query \(C_1^* C_2 C_3 C_4 T\), for any \(C_1^* \neq C_1\), as a forgery attempt. The adversary wins with advantage 1.
Changelog

- Version 2.0 (August 12, 2015): Major revision, with additional illustrations and an approach for generating concrete attacks on schemes not found secure by our analysis. The running time of OCB and the three schemes in Figure 4.3 is updated: TurboBoost has to be disabled for accurate timing.
- Version 1.0 (June 23, 2015): First release.