Faster Yet Safer: Logging System Via Fixed-Key Blockcipher

Viet Tung Hoang
Florida State University
tvhoang@cs.fsu.edu

Cong Wu
Florida State University
wu@cs.fsu.edu

Xin Yuan
Florida State University
xyuan@cs.fsu.edu

Abstract

System logs are crucial for forensic analysis, but to be useful, they need to be tamper-proof. To protect the logs, a number of secure logging systems have been proposed from both academia and the industry. Unfortunately, except for the recent KennyLoggings construction, all other logging systems are broken by an attack of Paccagnella *et al.* (CCS 2020). In this work, we build a secure logging system that improves KennyLoggings in several fronts: adoptability, security, and performance. Our key insight for performance gain is to use AES on a *fixed, known* key. While this trick is widely used in secure distributed computing, this is the first time it has found an application in the area of symmetric-key cryptography.

1 Introduction

LOGGING MATTERS. Security breaches happen frequently, with millions of victims and widespread damages [15]. To investigate and recover from such incidents, people crucially rely on system logs for information of system execution history; there is now a cottage industry on how to use system logs for forensic analysis [26, 30, 33, 35]. Indeed, a recent survey from VMware Carbon Black [45] reports that 75% of the security specialists consider system logs to be the most valuable artifact during an incident investigation. Due to this demand, the global market of log management software is now a billion-dollar business, and it keeps growing steadily [1].

LOGGING SYSTEMS ARE BROKEN. Still, logs are only useful as long as attackers cannot tamper with them. But system logs are usually the first target of an experienced attacker to hide the traces of the intrusion. In the same VMware Carbon Black survey above, 87% of the respondents indicate that log tampering is a top evasion tactic.

To cope with the situation above, there have been a plethora of secure logging systems, from both academia and the industry [14, 17, 19, 28, 34, 37, 38, 41, 44]. These solutions either (i) produce integrity proofs for the logs and protect the signing key via forward security [17, 37], or (ii) rely on

trusted hardware [28, 38]. In this line of research, the focus is to give further utilities (such as public verifiability) or improve performance. Security of these schemes is generally not questioned; the consensus is that they *work*. Surprisingly, in a recent work [39], Paccagnella *et al.* challenge this belief, giving an attack that breaks *all* current logging systems in Linux.

Specifically, there is often a non-negligible window of time between (i) the moment that a system event occurs and its log is generated, and (ii) the time when the log is committed and becomes tamper-proof. Even worse, this time interval grows with the system load. As a result, under an attack that causes a burst of logging events, there is enough time for the attacker to gain root privilege, and then undetectably modify the logs of the intrusion activities before they are committed.

What causes those time intervals? In Linux, when a system event occurs, the kernel creates a log message and puts it in a queue. Later the log will be dequeued and sent to the user space for a secure logging system to commit and store in the log entries. For performance reason, messages in the queue must be processed *asynchronously*, resulting in such time intervals.

STATE OF THE ART: KENNYLOGGINGS. To thwart the attack above, Paccagnella *et al.* suggest that a secure logging system should operate within the kernel space, creating integrity proofs for logs *before* they are put in the queue. This approach however imposes stringent performance requirements on the logging systems, excluding prior designs that are based on public-key cryptography [34] or have non-constant complexity [19, 41]. Paccagnella *et al.* thus opt to follow a design of Bellare and Yee [14] for building KennyLoggings, the first kernel-based secure logging system. Their benchmarks show that KennyLoggings has just 8%–11% overhead on log-intensive applications.

Conceptually, KennyLoggings produces integrity proofs for log messages via a Message Authentication Code (MAC) scheme. To ensure that an attack cannot steal a signing key, once we sign for a message under the current key K_i, we

derive a new key $K_{i+1} \leftarrow H(K_i)$ and erase K_i, where H is a cryptographic hash function that is instantiated via Blake2 [4]. Instantiating the MAC scheme is however tricky, as log messages are usually short, but standard MAC schemes (such as GMAC [21]) are only optimized for long data. Paccagnella *et al.* thus opt to use SipHash [3], a dedicated MAC design for short messages. To improve performance, instead of deriving keys on the fly, KennyLoggings *precomputes* 200,000 keys and refills when half of them are used and deleted.

COULD WE DO BETTER? Despite the impressive performance statistics of KennyLoggings, a closer look reveals that it still leaves much to be desired.

First, KennyLoggings is based on Blake2 and SipHash. While these cryptographic tools have been used in a number of software such as the Linux kernel, they are not NIST-approved algorithms, preventing government agencies from using KennyLoggings.[1] It is therefore desirable to build an AES-based alternative for better adoptability from government agencies and companies.

Next, KennyLoggings may significantly increase the latency of system calls in a busy multi-threading environment, from 40% to 53%, according to the benchmarks in [39].

In addition, due to the key precomputation, KennyLoggings has to hoard 200,000 keys, occupying 3.2 MB in kernel memory. Storing lots of sensitive information makes it vulnerable to side-channel attacks such as [32]. It is therefore advisable to reduce the amount of keying material to a few hundred bytes. Turning off key precomputation in KennyLoggings however decreases the performance by 8.6% on average [39].

Also, KennyLoggings has to store an integrity proof per log message, resulting in substantial overheads in storing the logs and transmitting them to the auditor. In particular, while KennyLoggings uses an 8B tag, the actual overhead is 19B per message, because the binary tag is encoded as a 16B string of ASCII characters, and there is an additional 3B label for preserving the semantics of Linux Audit records. Given that an average log message is roughly 378B [36], the storage overhead of KennyLoggings is estimated to be around 5.14% [39].

Finally, while the signing and verification of KennyLoggings are already fast, it is still desirable to rev up the speed of these operations.

WARMUP: QUICKLOG. Following the blueprint in [14], we first build QuickLog, an AES-based logging system whose signing is on par with KennyLoggings, but its verification is 6–8 times faster. For example, it takes QuickLog just 73 ns to verify a 256-byte log message, whereas KennyLoggings needs about 529 ns. Moreover, QuickLog only has 32-byte keying material, which is much easier to protect against side-channel attacks. We note that our experimental server

---

[1]For example, according to NIST SP 800-175B [6], "when a federal agency requires the use of cryptography (e.g., for encryption), an **approved** algorithm must be used; approval is indicated by inclusion in a FIPS or SP."

only supports 128-bit AES-NI; the performance gap between QuickLog and KennyLoggings is likely to widen for latest machines that support 512-bit VAES instructions.

There are several obstacles in building an AES-based logging system whose performance is on par with KennyLoggings. As mentioned earlier, log messages are usually short, but standard blockcipher-based MAC schemes are not optimized on short messages. Even worse, they have a relatively expensive setup cost. In many MAC applications, this one-time cost can be amortized over many uses under the same key. Unfortunately, here we only sign one message per key, and thus we have to pay the cost for every single use. Worse still, if we want to run AES-NI from the Linux kernel, we have to save all floating-point unit (FPU) registers first, make AES calls, and then later restore those registers. This FPU context-switching incurs a heavy performance penalty, which is particularly problematic if all we need is to sign just one short message.

To deal with the issues above, instead of using AES with a secret key, we use AES with a *fixed, known* key and model it as an ideal permutation. The MAC key is instead used to whiten the input and output of AES, following the Even-Mansour design of blockcipher [20, 22]. As a result, although we keep evolving the MAC key, we always use AES with the same fixed key.

The trick of a fixed-key blockcipher is widely used in secure distributed computing to improve the performance of circuit garbling [10, 11]. By using a fixed key instead of changing key per operation, we can avoid disrupting the AES-NI pipeline and do not have to pay the key setup cost. Thanks to the pipelining of AES-NI, the verification cost of QuickLog is considerably faster than its signing cost, because we have only one message to sign at a time, but lots of data to verify.

On the other hand, given that we only need to sign one message per key, we realize that we do not need a fully-fledged MAC. Instead, what we need is just a *one-time* MAC. The classic, theoretical way to realize a one-time MAC is via a pairwise independent hash function [46], but existing constructions unfortunately do not meet our speed demand. We instead build a one-time MAC that we call XMAC by borrowing design ideas of the XOR MAC construction [8]. There are however fundamental differences between XMAC and XOR MAC. For example, XMAC is based on a fixed-key blockcipher whereas under XOR MAC, the blockcipher key is secret. Moreover, XMAC is stateless because we only need a one-time MAC; in contrast XOR MAC is either stateful or probabilistic.

The security proof of XMAC is nontrivial if one wants a strong bound. A naive approach is to rely on the known result that the Even-Mansour construction is a good PRF [20, 22] but one would end up with an inferior bound $\ell(p+\ell)/2^{128}$, where p is the number of queries to the ideal permutation and ℓ is the block length of the message that we sign. Our proof instead yields a stronger bound $(p+\ell)/2^{128}$. We also give a matching

attack, showing that our security bound is tight.

The XMAC construction is so fast that despite the heavy penalty of the FPU context switching above, our signing cost is faster than KennyLoggings for messages longer than 256B. Thus for applications whose average log length is beyond 256B, one can heuristically predict that QuickLog is better than KennyLoggings. This 256B threshold holds for many real-world situations. For example, a study by Ma *et al.* [36] observes that under realistic conditions, the average log length of a web server is about 378B.

While QuickLog follows the blueprint of Bellare and Yee [14], a proof is needed to warrant its security. Indeed, although KennyLoggings claims to inherit the security in [14], we point out a glitch in its design that completely voids the correct proof in [14]. While one can give a new (nontrivial) proof to justify the security of KennyLoggings, it indicates that a rigorous treatment is needed for QuickLog. To achieve this goal, we give a simpler framework for the security of logging system than the original treatment of Bellare and Yee. (The latter is complex because it aims to deal with a more general situation where MAC keys evolve over time intervals, and messages within the same interval are signed under the same key.) We then show that QuickLog has 96-bit security under our threat model, which is stronger than the conventional 64-bit security of many NIST standards such as GCM [21].

EVEN BETTER: QUICKLOG2. In both QuickLog and KennyLoggings, each log message has a corresponding tag, resulting in substantial storage overheads. To eliminate this storage cost, we build QuickLog2, an optimized version of QuickLog with a single *aggregate* tag that authenticates all log messages. Even better, with aggregate authentication, QuickLog2 can directly thwart the *truncation attack* where an adversary tries to delay the detection of its intrusion by deleting some recent log messages with their tags. In contrast, in QuickLog and KennyLoggings, one has to resort to additional mechanisms (such as requesting signing a known message immediately prior to an audit).

In addition, by avoiding appending an integrity proof to each logging event, QuickLog2 also improves the signing time, yet still retains the same verification speed as QuickLog. For example, to sign a 256-byte message, QuickLog and KennyLoggings need 366 ns and 362 ns respectively, whereas QuickLog2 only needs 205 ns. Our benchmarks also show that in busy multi-threading environments, QuickLog2 introduces much less overhead than both QuickLog and KennyLoggings. On the other hand, QuickLog2 has 48B of sensitive material to protect against side-channel attacks, which is slightly higher than QuickLog (32B), but still much smaller than KennyLoggings (3.2MB).

We now describe how to add aggregate authentication to QuickLog. A prior work of Ma and Tsudik [34] suggests the following way to update the aggregate tag T when we have a new log message M_i. First sign M_i to get a correspond-

ing tag T_i as usual, but then update T via $T \leftarrow H(T \parallel T_i)$, where H is a collision-resistant hash function such as SHA-2. Unfortunately, if we incorporate this method into QuickLog, it will significantly increase both the signing and verification time. Borrowing ideas from [29], we instead update T via $T \leftarrow T \oplus T_i$, making the overhead of the aggregation negligible. Note that under our approach, the verification algorithm is fully parallelizable. In contrast, the method of [34] forces the auditor to make a long chain of hashing.

The xor trick above first appears in the work of Katz and Lindell [29] for aggregating MAC signatures. Yet it has never been used for prior logging systems due to an obvious attack. In particular, for prior designs, one evolves keys over time intervals, and in each interval, multiple messages are signed under the same key. Thus the xor trick will fail to detect if log messages of the same interval are reordered. It however *does* work for our setting, as there is only a single message to sign per key. Still, proving that this method can cope with the truncation attack goes beyond what the abstraction of the xor trick in [29] can deliver.

We formalize a game-based framework to capture the security of logging protocols with aggregate authentication. We then generically show that using the xor trick on a secure logging scheme in the Bellare-Yee blueprint (such as QuickLog) meets the new notion. In particular, our proof implies that QuickLog2 also has 96-bit security.

A PERSPECTIVE. The move from standard secret-key block-cipher to fixed-key blockcipher is simple in hindsight but indicates a major conceptual leap. Indeed, while the trick of using a fixed-key blockcipher has seen widespread use in secure distributed computing, it has found no application in the area of symmetric-key cryptography. The reason is simple: one usually encrypts or authenticates many messages per key, and for such settings, there is no performance advantage in using AES on a fixed key compared to the standard way of running AES on a secret key. Our paper introduces the first application in the area of symmetric-key cryptography that benefits from using a fixed-key blockcipher. Finding other applications of this technique is an interesting direction for future work.

CONCURRENT RELATED WORK. In a concurrent and independent work, Ahmad, Lee, and Peinado [2] build HardLog, a logging system that employs a novel audit device to *synchronously* store critical log messages. Non-critical logs are still processed asynchronously, but HardLog ensures that the delay is bounded within 15 ms. This approach is complementary to ours; combining the two solutions will lead to the best of both worlds, namely the fine-grained log availability of HardLog, and the synchronous authentication of QuickLog2.

Game $\mathbf{G}_F^{\mathrm{prf}}(\mathcal{A})$	**procedure** $\mathrm{FN}(M)$
$b \leftarrow\!\!\$ \{0,1\};\ K \leftarrow\!\!\$ \mathcal{K}$	If $b = 1$ then return $F_K(M)$
$f \leftarrow\!\!\$ \mathrm{Func}(\mathrm{Dom},\mathrm{Rng})$	Else return $f(M)$
$b' \leftarrow\!\!\$ \mathcal{A}^{\mathrm{FN}};\ \mathrm{Return}\ (b' = b)$	

Figure 1: Game defining PRF security of a function $F : \mathcal{K} \times \mathrm{Dom} \to \mathrm{Rng}$.

2 Preliminaries

NOTATION AND TERMINOLOGY. Let ε denote the empty string. For a string x we write $|x|$ to refer to its bit length, and $x[i : j]$ is the bits i through j (inclusive) of x, for $1 \leq i \leq j \leq |x|$. By $\mathrm{Func}(\mathrm{Dom},\mathrm{Rng})$ we denote the set of all functions $f : \mathrm{Dom} \to \mathrm{Rng}$ and by $\mathrm{Perm}(\mathrm{Dom})$ the set of all permutations $\pi : \mathrm{Dom} \to \mathrm{Dom}$. We use $\perp$ as a special symbol to denote rejection, and it is assumed to be outside $\{0,1\}^*$.

If $\mathcal{X}$ is a finite set, we let $x \leftarrow\!\!\$ \mathcal{X}$ denote picking an element of $\mathcal{X}$ uniformly at random and assigning it to x. If A is an algorithm, we let $y \leftarrow A(x_1,\ldots;r)$ denote running A on inputs $x_1,\ldots$ and coins r, and assigning the output to y. By $y \leftarrow\!\!\$ A(x_1,\ldots)$ we denote picking r at random and letting $y \leftarrow A(x_1,\ldots;r)$. We write $\mathcal{A}^f$ to indicate that adversary $\mathcal{A}$ has oracle access to a function f.

GAMES. We use the game-playing framework of [13]. (See Fig. 1 for an example.) We write $G(\mathcal{A}) \Rightarrow b$ to denote the event of running game G with an adversary $\mathcal{A}$ that results in b. We also write $G(\mathcal{A})$ to abbreviate $G(\mathcal{A}) \Rightarrow \mathsf{true}$.

PRF. Let $F : \mathcal{K} \times \mathrm{Dom} \to \mathrm{Rng}$ be a function. For an adversary $\mathcal{A}$, define its advantage in breaking the PRF security of F as

$$\mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{A}) = 2 \cdot \Pr[\mathbf{G}_F^{\mathrm{prf}}(\mathcal{A})] - 1 \ ,$$

where game $\mathbf{G}_F^{\mathrm{prf}}(\mathcal{A})$ is defined in Fig. 1. If the function F is built on top of an ideal permutation π then $\mathcal{A}$ is also given oracle access to both π and π^{-1}.

(ONE-KEY) EVEN-MANSOUR SCHEME. The Even-Mansour constructions [20, 22] is a well-known way to build a blockcipher $\mathsf{EM}[\pi] : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ on top of a permutation $\pi : \{0,1\}^n \to \{0,1\}^n$. In particular, $\mathsf{EM}[\pi](K,M) = \pi(K \oplus M) \oplus K$. The following Lemma 2.1 gives a bound on the PRF security of $\mathsf{EM}[\pi]$ in the ideal-permutation model; it combines the well-known PRP bound of EM [20, 22] with the PRP/PRF Switching Lemma [12].

Lemma 2.1. *Let* $\pi : \{0,1\}^n \to \{0,1\}^n$ *be a permutation that we will model as an ideal permutation. Define* $\mathsf{EM}[\pi]$ *as above. Then for an adversary* $\mathcal{A}$ *making at most* q FN *queries and at most* p *queries to* π *and* π^{-1},

$$\mathbf{Adv}_{\mathsf{EM}[\pi]}^{\mathrm{prf}}(\mathcal{A}) \leq \frac{q(p+q-1)}{2^n} \ .$$

SYSTEMS AND TRANSCRIPTS. Following the notation from [25], it is convenient to consider interactions of a distinguisher $\mathcal{A}$ with an abstract system $\mathbf{S}$ which answers $\mathcal{A}$'s queries. The resulting interaction then generates a transcript $\theta = \big((X_1,Y_1),\ldots,(X_q,Y_q)\big)$ of query-answer pairs. It is known that $\mathbf{S}$ is entirely described by the probabilities $\mathsf{p}_{\mathbf{S}}(\theta)$ that correspond to the system $\mathbf{S}$ responding with answers as indicated by θ when the queries in θ are made.

We will generally describe systems informally, or more formally in terms of a set of oracles they provide, and only use the fact that they define corresponding probabilities $\mathsf{p}_{\mathbf{S}}(\theta)$ without explicitly giving these probabilities.

THE H-COEFFICIENT TECHNIQUE. The H-coefficient technique considers a deterministic distinguisher $\mathcal{A}$ that tries to distinguish a "real" system $\mathbf{S}_1$ from an "ideal" system $\mathbf{S}_0$. The adversary's interactions with those systems define transcripts $\mathcal{T}_1$ and $\mathcal{T}_0$, respectively. The following result bounds the distinguishing advantage of $\mathcal{A}$.

Lemma 2.2. [18, 40] *Suppose we can partition the set of valid transcripts for the ideal system into good and bad ones. Further, suppose that there exists a constant* $\varepsilon \geq 0$ *such that* $1 - \frac{\mathsf{p}_{\mathbf{S}_1}(\theta)}{\mathsf{p}_{\mathbf{S}_0}(\theta)} \leq \varepsilon$ *for every good transcript* θ. *Then, the advantage of* $\mathcal{A}$ *in distinguishing* $\mathbf{S}_1$ *and* $\mathbf{S}_0$ *is at most* $\varepsilon + \Pr[\mathcal{T}_0 \text{ is bad}]$.

3 One-time MAC

In this section we'll formalize a notion of *one-time MAC*, and show how to realize it via a fixed-key blockcipher.

3.1 Security Notions

ONE-TIME MAC. A one-time MAC is a function $F : \mathcal{K} \times \mathrm{Dom} \to \mathrm{Rng}$. It takes as input a key $K \in \mathcal{K}$ and a message $M \in \mathrm{Dom}$, and then deterministically produces a tag $T \leftarrow F(K,M)$.

For an adversary $\mathcal{A}$, we define its advantage in breaking security of F as

$$\mathbf{Adv}_F^{\mathrm{mac1}}(\mathcal{A}) = \Pr[\mathbf{G}_F^{\mathrm{mac1}}(\mathcal{A})] \ ,$$

where game $\mathbf{G}_F^{\mathrm{mac1}}(\mathcal{A})$ is in Fig. 2. Informally, the adversary $\mathcal{A}$ first specifies a message M and stores its state in a string σ. It is then given back its state σ and the genuine tag T of M. The goal of the adversary is to forge a message $M' \neq M$ and its corresponding tag T'.

The security notion above is a weaker variant of the standard MAC notion where the adversary can specify many messages and learn their corresponding tags. A standard MAC construction therefore can be used to sign many messages. In contrast, a one-time MAC construction is intended to sign a single message. Aiming for just a one-time MAC allows us to realize

$$
\begin{array}{|l|}
\hline
\textbf{Game } \mathbf{G}_F^{\mathrm{mac1}}(\mathcal{A}) \\
\hline
(M,\sigma) \leftarrow\!\!\$\ \mathcal{A};\ K \leftarrow\!\!\$\ \mathcal{K};\ T \leftarrow F(K,M) \\
(M',T') \leftarrow\!\!\$\ \mathcal{A}(T,\sigma);\ T^* \leftarrow F(K,M') \\
\text{return } (M' \neq M \text{ and } T' = T^*) \\
\hline
\end{array}
$$

Figure 2: Game defining single-user security of a one-time MAC F.

$$
\begin{array}{|l|}
\hline
\textbf{Game } \mathbf{G}_F^{\mathrm{mu-mac1}}(\mathcal{A}) \\
\hline
v \leftarrow 0;\ (i,M',T') \leftarrow\!\!\$\ \mathcal{A}^{\mathrm{SIGN}} \\
T^* \leftarrow F(K_i,M');\ \text{Return } (M' \neq M_i \text{ and } T' = T^*) \\
\hline
\text{Procedure } \mathrm{SIGN}(M) \\
v \leftarrow v+1;\ K_v \leftarrow\!\!\$\ \mathcal{K};\ M_v \leftarrow M;\ T \leftarrow F(K_v,M) \\
\text{Return } T \\
\hline
\end{array}
$$

Figure 3: Game defining multi-user security of a one-time MAC F. If the adversary makes q signing queries then its forgery output (i,M',T') must satisfy $i \in \{1,\dots,q\}$.

this goal with a very simple and efficient construction via a fixed-key blockcipher.

<u>MULTI-USER SECURITY.</u> In practice one would use a one-time MAC to sign many messages, with one fresh random key for each message. This is the *multi-user* setting, introduced by Biham [16] in symmetric cryptanalysis and by Bellare, Boldyreva, and Micali [7] in public-key cryptography. To capture the multi-security of a one-time MAC F, for an adversary $\mathcal{A}$, we let

$$
\mathbf{Adv}_F^{\mathrm{mu-mac1}}(\mathcal{A}) = \Pr[\mathbf{G}_F^{\mathrm{mu-mac1}}(\mathcal{A})]\ ,
$$

where game $\mathbf{G}_F^{\mathrm{mu-mac1}}(\mathcal{A})$ is defined in Fig. 3. Informally, $\mathcal{A}$ is given a signing oracle. For each v-th query M_v, the oracle creates a secret fresh random key K_v and returns the tag $T \leftarrow F(K_v,M_v)$. The adversary then creates a forgery (i,M',T'). It wins the game if $M' \neq M_i$ and $T' = F(K_i,M')$.

3.2 The XMAC Construction

We now show how to build a one-time MAC via a fixed-key blockcipher; we call this construction XMAC.

<u>THE XMAC CONSTRUCTION.</u> Let n be a multiple of 8 and let $\pi: \{0,1\}^n \to \{0,1\}^n$ be a permutation. Let $\tau, r \in \{1,\dots,n\}$ such that r is a multiple of 8. For an integer $i \in \{1,\dots,2^r\}$, let $[i]_r$ denote an r-bit encoding of i. The XMAC construction only processes byte strings up to $(2^r - t) \cdot t$ bytes, where $t = (n-r)/8$, its tag length is τ bits, and its key space is $\{0,1\}^n$. It is described in Fig. 4. Note that the list of counters we use in signing a message uniquely encodes its byte length. This ensures that the signing message and the forgery one will result in different sets of AES inputs.

To implement XMAC, we can instantiate π via AES with a constant key, meaning that $n = 128$. In our construction, we pick $r = 16$. In other words, our XMAC implementation can digest messages up to $(2^{16} - 14) \cdot 14 = 917{,}308$ bytes. This is enough if one wants to use XMAC for system logging, as log messages are short. In practice, log size rarely goes beyond 1KB; in our benchmarks, log size in fact never exceeds 400B. The default maximum log size of Linux Audit is 8KB, way below the limit of XMAC.

The design of XMAC takes inspiration from the XOR MAC construction [8], but there are major differences between the two. In particular, while XOR MAC is either stateful or randomized, XMAC is stateless and deterministic. As a result, not only is XMAC simpler than XOR MAC, it is also faster. Moreover, XMAC is based on a fixed-key blockcipher whereas under XOR MAC, the blockcipher key is secret. Thus XMAC is preferred in the setting where one has to update key per signing, as there is no key setup and we can avoid disrupting the AES-NI pipeline.

<u>DISCUSSION.</u> Traditional blockcipher-based MAC (such as GMAC [21]) typically aims to minimize the number of blockcipher calls. Such designs strive to use $\lceil m/16 \rceil$ parallel calls to authenticate an m-byte message, at the expense of some setup cost (such as building a look-up table for fast finite-field multiplications). This approach is not suitable for our setting, because (i) having an expensive setup cost is undesirable when one only authenticates a single message per key, and (ii) for short messages, a few extra blockcipher calls have little impact on the running time, thanks to the pipelining of AES-NI. The XMAC construction thus uses $\lceil m/14 \rceil$ parallel blockcipher calls for an m-byte message, with no setup cost.

<u>CAVEAT.</u> We note that XMAC must not be used to sign more than one message per key. For example, suppose that we use XMAC to sign messages $M_1 = 0^{n-r} \| 0^{n-r}$, $M_2 = 1^{n-r} \| 1^{n-r}$, and $M_3 = 0^{n-r} \| 1^{n-r}$ on the same key. Let T_1, T_2, T_3 be the corresponding tags. Then one can forge the tag of $M_4 = 1^{n-r} \| 0^{n-r}$ via $T_4 = T_1 \oplus T_2 \oplus T_3$.

<u>SINGLE-USER SECURITY OF XMAC.</u> The following result shows that XMAC is a good one-time MAC in the ideal-permutation model; the proof is in Section 3.3. Note that if we use a variant of XMAC that is based on the Even-Mansour construction and rely on Lemma 2.1 then we will end up with an inferior bound $\frac{\ell(p+\ell)}{2^n} + \frac{1}{2^\tau}$.

Theorem 3.1. *Let $\pi: \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Define $\mathrm{XMAC}[\pi,\tau]$ as above. Consider an adversary $\mathcal{A}$ that makes at most p ideal-permutation queries, and its two messages are of at most ℓ blocks (each of $n-r$ bits). Then*

$$
\mathbf{Adv}_{\mathrm{XMAC}[\pi,\tau]}^{\mathrm{mac1}}(\mathcal{A}) \leq \frac{4p+2\ell}{2^n} + \frac{1}{2^\tau}\ .
$$

```
procedure XMAC[π, τ](K, M)
M₁ ‖ ⋯ ‖ Mₘ ← M  // |Mₘ| ≤ n − r, and other |Mᵢ| = n − r
v ← n − r − |Mₘ|; u ← v/8; R ← K
For i = 1 to m − 1 do Xᵢ ← [i]ᵣ ‖ Mᵢ;  R ← R⊕π(Xᵢ⊕K)
Xₘ ← [m + u]ᵣ ‖ (Mₘ0ᵛ);  R ← R⊕π(Xₘ⊕K)
Return R[1 : τ]
```
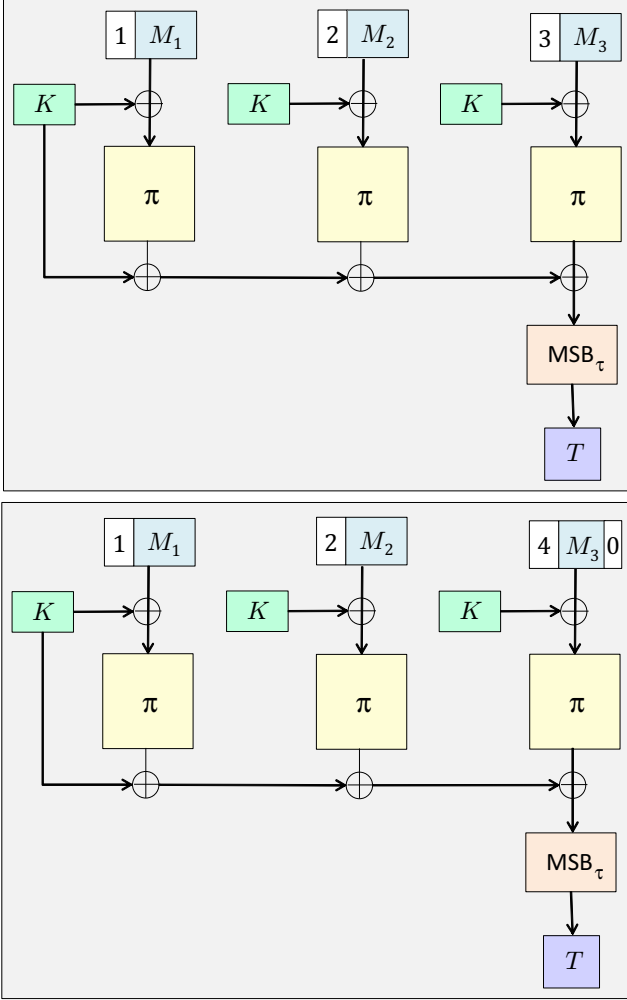

Figure 4: The one-time MAC construction XMAC (top) and an illustration of XMAC for the 3-block case, where the message is either full-block (middle), or fragmentary (bottom) with a one-byte padding. For the last block, we will pad 0's if necessary, and increase the counter with the byte length of the padding. The box MSB_τ outputs the τ-bit prefix of the input.

MULTI-USER SECURITY OF XMAC. The following result shows that XMAC also has good multi-user one-time MAC security in the ideal-permutation model. The proof is in Section 3.4.

Theorem 3.2. *Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Define XMAC$[\pi, \tau]$ as above. Consider an adversary $\mathcal{A}$ that makes at most p ideal-permutation queries and $q \geq 1$ signing queries, and its messages are of at most s blocks (each of $n − r$ bits). Then*

$$\mathbf{Adv}^{\mathrm{mu-mac1}}_{\mathrm{XMAC}[\pi, \tau]}(\mathcal{A}) \leq \frac{4q(p + s)}{2^n} + \frac{1}{2^\tau} \ .$$

MATCHING ATTACKS. The bound in Theorem 3.2 consists of two important terms $1/2^\tau$ and $pq/2^n$ that correspond to actual attacks. To have advantage $1/2^\tau$, one simply picks an arbitrary message M', samples $T' \leftarrow \!\!\$ \{0,1\}^\tau$, and then outputs $(1, M', T')$. For the term $pq/2^n$, consider the following attack for the case $\tau = n$.

- First, pick arbitrary distinct $(n − r)$-bit strings M and M'. Let $X = [1]_r \| M$ and $X' = [1]_r \| M'$.
- Next, for every $i \leq q$, query SIGN(M) to get answer T_i. Note that each T_i is $\pi(K_i \oplus X) \oplus K_i$, where K_i is the i-th key.
- Pick arbitrary distinct $L_1, \ldots, L_p \in \{0,1\}^n$. For every $k \leq p$, compute $V_k \leftarrow \pi(L_k \oplus X) \oplus L_k$. If there are some i and k such that $T_i = V_k$ then we guess the key K_i as L_k, and then output the forgery as $(i, M', \pi(L_k \oplus X') \oplus L_k)$.

To analyze the attack above, we will use the following inequality.

Lemma 3.3. [9] *Let $q \geq 1$ be an integer and $a \geq 0$ a real number. Assume $aq \leq 1$. Then $(1 − a)^q \leq 1 − aq/2$.*

First, since $L_1, \ldots, L_p$ are distinct and $K_1, \ldots, K_q \leftarrow \!\!\$ \{0,1\}^n$, the chance that there is a collision $L_k = K_i$ is $1 − (1 − p/2^n)^q$. Using Lemma 3.3 with $a = p/2^n$, the chance that there is a collision $L_k = K_i$ (and thus a matching $V_k = T_i$) is at least

$$1 − (1 − p/2^n)^q \geq pq/2^{n+1} \ .$$

Still, given a matching $T_i = V_k$, we need to analyze the conditional probability that $L_k = K_i$, since there might be false positives. Fix i and k. Let Hit be the event that $K_i = L_k$ and let Bad be the event that $K_i \neq L_k$ but somehow $T_i = V_k$. Note that the events Hit and Bad are disjoint. Our goal is to bound Pr[Hit | Hit ∪ Bad]. From Bayes' theorem,

$$
\begin{aligned}
\Pr[\mathsf{Hit} \mid \mathsf{Hit} \cup \mathsf{Bad}] &= \frac{\Pr[\mathsf{Hit} \cup \mathsf{Bad} \mid \mathsf{Hit}] \cdot \Pr[\mathsf{Hit}]}{\Pr[\mathsf{Hit} \cup \mathsf{Bad}]} \\
&= \frac{\Pr[\mathsf{Hit}]}{\Pr[\mathsf{Hit}] + \Pr[\mathsf{Bad}]} \\
&= \frac{1}{1 + \Pr[\mathsf{Bad}]/\Pr[\mathsf{Hit}]} \ .
\end{aligned}
$$

For fixed i and k, the chance that $K_i = L_k$ is $1/2^n$, whereas given that $K_i \neq L_k$, the conditional probability that $T_i = \pi(K_i \oplus X) \oplus K_i$ and $V_k = \pi(L_k \oplus X) \oplus L_k$ are the same is at most $1/(2^n − 1) \leq 2/2^n$. Therefore

$$
\begin{aligned}
\Pr[\mathsf{Bad}] &= \Pr[K_i \neq L_k \text{ and } T_i = V_k] \\
&\leq \Pr[T_i = V_k \mid K_i \neq L_k] \leq \frac{2}{2^n} \ .
\end{aligned}
$$

6

Game $\mathbf{G}_F^{\mathrm{prf2}}(\mathcal{A})$	procedure $\mathrm{FN}(M)$
$b \leftarrow\!\!{\scriptscriptstyle\$}\, \{0,1\}$; $K \leftarrow\!\!{\scriptscriptstyle\$}\, \mathcal{K}$	$count \leftarrow count + 1$
$f \leftarrow\!\!{\scriptscriptstyle\$}\, \mathrm{Func}(\mathrm{Dom},\mathrm{Rng})$	If $count > 2$ then return $\perp$
$count \leftarrow 0$; $b' \leftarrow\!\!{\scriptscriptstyle\$}\, \mathcal{A}^{\mathrm{FN}}$	If $b = 1$ then return $F_K(M)$
Return $(b' = b)$	Else return $f(M)$

Figure 5: Game defining two-time PRF security of a function $F : \mathcal{K} \times \mathrm{Dom} \to \mathrm{Rng}$.

Hence $\Pr[\mathsf{Bad}]/\Pr[\mathsf{Hit}] \le 2$, and thus $\Pr[\mathsf{Hit} \mid \mathsf{Hit} \cup \mathsf{Bad}] \ge 1/3$. As a result, the attack above wins with advantage at least $pq/6 \cdot 2^n$.

TIGHTNESS OF THE BOUND. The matching attack for the term $pq/2^n$ above can be extended to work with a general τ, but the advantage will dwindle by a factor of $2^{n-\tau}$. Thus while our bound is tight if τ is close to n, there might be room for improvement for the case that τ is much smaller than n (such as $n = 128$ and $\tau = 64$). We leave this as an open problem.

3.3 Proof of Theorem 3.1

PROOF OUTLINE. As a stepping stone, we will define an intermediate notion that we call *two-time PRF*. It is an analogue of the standard PRF notion but (i) the adversary is now allowed only two PRF queries, and (ii) if the PRF is built on top of an ideal permutation π then the adversary is given access to both π and π^{-1}, but it is prohibited from querying them after the second PRF query. We will show that a two-time PRF is also a good one-time MAC; the proof is similar to the classic proof that PRF security implies MAC security. We will then prove that XMAC is a good two-time PRF via the H-coefficient technique [18, 40], and thus XMAC is also a good one-time MAC.

DEFINING TWO-TIME PRF. Let $F : \mathcal{K} \times \mathrm{Dom} \to \mathrm{Rng}$ be a function. For an adversary $\mathcal{A}$, define its advantage in breaking the two-time PRF security of F as

$$\mathbf{Adv}_F^{\mathrm{prf2}}(\mathcal{A}) = 2 \cdot \Pr[\mathbf{G}_F^{\mathrm{prf2}}(\mathcal{A})] - 1 \;,$$

where game $\mathbf{G}_F^{\mathrm{prf2}}(\mathcal{A})$ is defined in Fig. 5. This game is similar to the standard PRF game in Fig. 1, but after the second query, the FN oracle only returns $\perp$. In other words, the adversary is effectively limited to two queries. If the function F is built on top of an ideal permutation π then $\mathcal{A}$ is also given oracle access to both π and π^{-1}, but it is prohibited from querying them after the second PRF query.

We stress that in defining two-time PRF security, it is crucial that the adversary is not allowed to query π and π^{-1} after the second PRF query. Dropping this restriction will lead to the following devastating attack on XMAC for the typical case $\tau = n$. In particular, pick an arbitrary $(n-r)$-bit string M. Query M and $M' = M \,\|\, M$ to the FN oracles

to get answers T and T' respectively. Let $B_1 = [1]_r \,\|\, M$ and $B_2 = [2]_r \,\|\, M$. Note that in the real world, $T = K \oplus \pi(K \oplus B_1)$ and $T' = K \oplus \pi(K \oplus B_1) \oplus \pi(K \oplus B_2)$. One then can recover the key K via $\pi^{-1}(T \oplus T') \oplus B_2$, and return 1 if $T = K \oplus \pi(K \oplus B_1)$. This attack wins with advantage $1 - 1/2^n$.

TWO-TIME PRF IMPLIES ONE-TIME MAC. The following result shows that any good two-time PRF is also a good one-time MAC; the proof is in Appendix B. As a result, below we will focus on proving two-time PRF security of XMAC.

Proposition 3.4. *Let $F : \mathcal{K} \times \mathrm{Dom} \to \mathrm{Rng}$ be a function. For any adversary $\mathcal{A}$, we can construct an adversary $\mathcal{B}$ of about the same time such that*

$$\mathbf{Adv}_F^{\mathrm{mac1}}(\mathcal{A}) \le \mathbf{Adv}_F^{\mathrm{prf2}}(\mathcal{B}) + \frac{1}{|\mathrm{Rng}|} \;.$$

Adversary $\mathcal{B}$ uses the same amount of resources as $\mathcal{A}$, and its messages are the same as those of $\mathcal{A}$.

XMAC IS A GOOD TWO-TIME PRF. The following result shows that XMAC is a good two-time PRF in the ideal-permutation model.

Proposition 3.5. *Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Define $\mathrm{XMAC}[\pi,\tau]$ as above. Consider an adversary $\mathcal{B}$ that makes at most p ideal-permutation queries, and its two messages are of at most ℓ blocks (each of $n-r$ bits). Then*

$$\mathbf{Adv}_{\mathrm{XMAC}[\pi,\tau]}^{\mathrm{prf2}}(\mathcal{B}) \le \frac{4p + 2\ell}{2^n} \;.$$

The proof of Proposition 3.5 is in Appendix C. Here we sketch some high-level intuitions. Let M and M' be the PRF queries, with $|M| \ge |M'|$, and let T and T' be the corresponding PRF outputs. In the real world, these queries internally result in calling $\pi(X_1 \oplus K), \ldots, \pi(X_m \oplus K)$ for M, and $\pi(X_1' \oplus K), \ldots, \pi(X_s' \oplus K)$ for M'. Since the list of counters uniquely encodes the message byte length, and $M \ne M'$ due to the definitional restriction, there must be some $X_t \notin \{X_1', \ldots, X_s'\}$.

Intuitively, we want to show that in the real world (where outputs are generated via XMAC), it is unlikely that the adversary can query $\pi(X)$ with $X \in \{X_t \oplus K, X_s' \oplus K\}$, or query $\pi^{-1}(Y)$ that ends up with answer $X_t \oplus K$ or $X_s' \oplus K$. If this does not happen then $R \leftarrow \pi(X_t \oplus K)$ and $R' \leftarrow \pi(X_s' \oplus K)$ serve as one-time pads to make T and T' pseudorandom.[2] Using the H-coefficient technique [18, 40], we can instead consider the chance this bad event happens in the *ideal* world, for $K \leftarrow\!\!{\scriptscriptstyle\$}\, \{0,1\}^n$ independent of the adversary's view.

The convenience above however comes with a cost. Note that in both worlds, we can reconstruct the one-time pads R

---

[2]It is possible that $X_s' \in \{X_1, \ldots, X_m\}$, meaning that we also use R' in producing T, but it is protected by R.

and R' via (T, T', K), and all $\pi(X_i)$ with $i \in \{1, \ldots, m\} \setminus \{t\}$, and all $\pi(X'_j)$ with $j \leq s-1$. Using the H-coefficient technique requires us to show that in the ideal world, it is unlikely that the adversary can query $\pi^{-1}(Y)$ with $Y \in \{R, R'\}$, or query $\pi(X)$ that ends with answer R or R'. This bad event leads to inconsistency, as in the ideal world, in general $R \neq \pi(X_t \oplus K)$ and $R' \neq \pi(X'_s \oplus K)$. To bound the chance this event happens, we rely on the fact that in the ideal world, the string K and the second PRF output are uniformly random, independent of the adversary's ideal-cipher queries and their answers.

XMAC IS A GOOD ONE-TIME MAC. Consider an adversary $\mathcal{A}$ attacking the one-time MAC security of $\mathsf{XMAC}[\pi, \tau]$ that makes at most p ideal-permutation queries, and its two messages are of at most ℓ $(n-r)$-bit blocks. From Proposition 3.4 for $\mathrm{Rng} = \{0,1\}^\tau$, we can construct an adversary $\mathcal{B}$ such that

$$\mathbf{Adv}_{\mathsf{XMAC}[\pi,\tau]}^{\mathrm{mac1}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathsf{XMAC}[\pi,\tau]}^{\mathrm{prf2}}(\mathcal{B}) + \frac{1}{2^\tau} \ .$$

Adversary $\mathcal{B}$ makes at most p ideal-permutation queries and its two messages are of at most ℓ $(n-r)$-bit blocks. Then from Proposition 3.5,

$$\mathbf{Adv}_{\mathsf{XMAC}[\pi,\tau]}^{\mathrm{prf2}}(\mathcal{B}) \leq \frac{4p + 2\ell}{2^n} \ .$$

Hence

$$\mathbf{Adv}_{\mathsf{XMAC}[\pi,\tau]}^{\mathrm{mac1}}(\mathcal{A}) \leq \frac{4p + 2\ell}{2^n} + \frac{1}{2^\tau} \ .$$

3.4 Proof of Theorem 3.2

The key idea of the proof is to reduce the multi-user one-time MAC security to the notion of two-time PRF defined in Section 3.3 using a standard hybrid argument. This is established in Proposition 3.6 below; the proof is in Appendix A.

Proposition 3.6. *Let $F : \mathcal{K} \times \mathrm{Dom} \to \mathrm{Rng}$ be a function. For any adversary $\mathcal{A}$ making $q \geq 1$ signing queries, we can construct an adversary $\mathcal{B}$ of about the same time such that*

$$\mathbf{Adv}_F^{\mathrm{mu-mac1}}(\mathcal{A}) \leq q \cdot \mathbf{Adv}_F^{\mathrm{prf2}}(\mathcal{B}) + \frac{1}{|\mathrm{Rng}|} \ .$$

Adversary $\mathcal{B}$ runs $\mathcal{A}$ and calls F to compute the tags for all but one signing messages of $\mathcal{A}$, and queries the remaining signing message and the forgery message of $\mathcal{A}$ to its PRF oracle.

Back to the multi-user security of XMAC, from Proposition 3.6, we can construct an adversary $\mathcal{B}$ of at most Q ideal-permutation queries whose two messages are of at most ℓ blocks, where $Q + \ell = p + s$, such that

$$\mathbf{Adv}_{\mathsf{XMAC}[\pi,\tau]}^{\mathrm{mu-mac1}}(\mathcal{A}) \leq q \cdot \mathbf{Adv}_{\mathsf{XMAC}[\pi,\tau]}^{\mathrm{prf2}}(\mathcal{B}) + \frac{1}{2^\tau} \ . \quad (1)$$

From Proposition 3.5,

$$\mathbf{Adv}_{\mathsf{XMAC}[\pi,\tau]}^{\mathrm{prf2}}(\mathcal{B}) \leq \frac{4Q + 2\ell}{2^n} \leq \frac{4(Q+\ell)}{2^n} = \frac{4(p+s)}{2^n} \ . \quad (2)$$

Combining Eq. (1) and Eq. (2) gives us the claimed bound.

4 Warmup: QuickLog

4.1 Formalizing Security of Logging Systems

THREAT MODEL. Following prior work [14, 39], we consider an adversary that initially has non-privileged access to a machine via, say stolen credentials, and then mounts an attack to escalate privilege. We assume that this attack invokes some system calls that are recorded in the system's audit logs. However, after gaining full system control, the adversary can modify the logs to hide the traces of the attack and avoid detection.

We assume that the audit logs are periodically sent to a trusted machine for analysis. For the auditor to detect tampering of logs, the logging system will include a short tag for each log as a proof of integrity. Initially, the logging system and the auditor share a secret short state; the logging system will then update the state for each tag it signs and erase the prior state from the host's memory. We assume that (i) it is impossible for the adversary to recover deleted states, and (ii) before gaining privileged access, the adversary cannot retrieve partial information of secrets in kernel memory, say using side-channel attacks such as [32].

SYNTAX OF LOGGING PROTOCOL. Syntactically, a *logging protocol* Π consists of a pair of deterministic algorithms $(\mathsf{Update}, \mathsf{Sign})$ and is associated with a state space $\mathcal{S}$.

- Initially, a root state $S_0 \leftarrow_\$ \mathcal{S}$ is sampled. Derive $(K_1, S_1) \leftarrow \mathsf{Update}(S_0)$, share S_0 with the auditor, and erase S_0 from the host's memory.

- When we need to sign the i-th log message M_i, we retrieve (S_i, K_i) and generate the tag $T_i \leftarrow \mathsf{Sign}(K_i, M_i)$. We then update $(K_{i+1}, S_{i+1}) \leftarrow \mathsf{Update}(S_i)$ and delete (K_i, S_i) from the host's memory so that they are no longer available for the adversary after it gains full system control.

- Given a root state S_0 and message-tag pairs $(M_1, T_1), \ldots, (M_q, T_q)$, an auditor can verify the integrity of these logs by iteratively deriving $(K_i, S_i) \leftarrow \mathsf{Update}(S_{i-1})$ and checking if $T_i = \mathsf{Sign}(K_i, M_i)$ for every $i \leq q$.

The syntax above is a simplified version of the framework of Bellare and Yee [14]; the latter instead evolves keys over time intervals. As a result, messages within the same interval are signed under the same key. However, as pointed out by Paccagnella *et al.* [39], if an attack happens within an interval, the adversary can get the signing key of the current interval and undetectably hide the traces of its intrusion. We follow their conservative recommendation, signing just one message per key. This also substantially simplifies the syntax, since

Game $\mathbf{G}_\Pi^{\mathrm{fa}}(\mathcal{A})$	**procedure** $\mathrm{SIGN}(M)$
$S_0 \leftarrow_\$ \mathcal{S}; \; v \leftarrow 0; \; (i,\sigma) \leftarrow_\$ \mathcal{A}^{\mathrm{SIGN}}$	$v \leftarrow v+1; \; M_v \leftarrow M$
$(M',T') \leftarrow_\$ \mathcal{A}(S_i,\sigma)$	$(K_v,S_v) \leftarrow \mathrm{Update}(S_{v-1})$
$T^* \leftarrow \mathrm{Sign}(K_i,M')$	$T_v \leftarrow \mathrm{Sign}(K_v,M_v)$
Return ($M' \neq M_i$ and $T' = T^*$)	Return T_v

Figure 6: Game defining the forward authenticity of a logging protocol Π.

one does not have to worry about the deletion or reordering of messages within the same interval.

SECURITY NOTION FOR LOGGING PROTOCOLS. For an adversary $\mathcal{A}$ attacking a logging protocol Π, we define its advantage in breaking the *forward authenticity* (FA) of Π as

$$\mathbf{Adv}_\Pi^{\mathrm{fa}}(\mathcal{A}) = \Pr[\mathbf{G}_\Pi^{\mathrm{fa}}(\mathcal{A})] \;,$$

where game $\mathbf{G}_\Pi^{\mathrm{fa}}(\mathcal{A})$ is defined in Fig. 6. Initially the game samples a state $S_0 \leftarrow_\$ \mathcal{S}$, and runs $\mathcal{A}$ with access to a signing oracle. For the v-th signing query M_v, the oracle runs $\mathrm{Update}(S_{v-1})$ to get an updated key K_v and state S_v, and then returns the tag $T_v \leftarrow \mathrm{Sign}(K_v,M_v)$. When $\mathcal{A}$ finishes querying, its produces an index $i \geq 1$ and saves its internal state to a string σ. The adversary is then given back σ and the state S_i (but now without oracle access) to output a forgery (M',T'). It wins the game if $M' \neq M_i$ but $T' = \mathrm{Sign}(K_i,M')$.

The notion of forward authenticity is a simplified version of the notion of forward-secure MAC of Bellare and Yee [14] in which there is only one signing message per time interval. We now give an intuition for why achieving this goal makes the logs tamper-proof. Suppose that when the adversary gets full control of the system, it obtains the state S_q (and thus knows all the subsequent keys and states), but the system calls of its attack are already recorded in the logs $M_1,\ldots,M_q$. These messages are signed under the (now deleted) keys $K_1,\ldots,K_q$. Suppose that later an auditor receives the stream of tampered message-tag pairs $(M'_1,T'_1),(M'_2,T'_2),\ldots$. Let $i \leq q$ be the smallest index that $M_i \neq M'_i$. (This index must exist, otherwise the auditor will find out the intrusion.) The forward authenticity ensures that T'_i will be different from $\mathrm{Sign}(K_i,M_i)$, leading to a detection of the tampering.

We note that by mounting a truncation attack (that is, deleting the most recent log messages), an adversary may be able to *delay* detection until the next verification period, but will be ultimately detected. To avoid detection delay, Paccagnella *et al.* [39] suggest that an administrator should request signing a known message immediately prior to an audit.

4.2 The QuickLog System

DESCRIPTION OF QUICKLOG. Let $F : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a PRF and let $G : \{0,1\}^n \times \mathrm{Dom} \to \{0,1\}^\tau$ be a one-time MAC. The logging protocol $\mathrm{QuickLog}[F,G]$ with

procedure $\mathrm{QuickLog}[F,G].\mathrm{Update}(S)$
$S' \leftarrow F(S,[0]_n); \; K' \leftarrow F(S,[1]_n); \;$ Return (K',S')
procedure $\mathrm{QuickLog}[F,G].\mathrm{Sign}(K,M)$
$T \leftarrow G(K,M); \;$ Return T

Figure 7: The logging protocol QuickLog, built on top of a PRF F and a one-time MAC G. Here for each integer $a \in \{0,\ldots,2^n-1\}$, we let $[a]_n$ denote an n-bit encoding of a.

message space Dom is described in Fig. 7. We instantiate F via the Even-Mansour method $\mathrm{EM}[\pi]$ (that is described in Section 2) and G via $\mathrm{XMAC}[\pi,\tau]$ (that is described in Section 3.2). The underlying permutation π for both constructions is AES with the all-zero key.

QuickLog follows the blueprint in the work of Bellare and Yee [14] but uses a one-time MAC instead of a standard MAC. The key idea for performance improvement here is to build both F and G on top of a fixed-key blockcipher.

DISCUSSION. Paccagnella *et al.* [39] suggest that one should implement a logging protocol in the kernel to avoid their race attack. Unfortunately, to implement XMAC efficiently one needs to use vector instructions such as Intel SSE2, but calling them from the Linux kernel incurs a performance penalty. In particular, one has to save all floating-point unit (FPU) registers before using the vector instructions, and then later restoring those. We stress that this issue only affects the performance of the signing operation, as the verification is implemented in the user space. Despite this penalty, our experiments show that the signing part of QuickLog is on par with that of its competitor, KennyLoggings [39].

SECURITY OF QUICKLOG. The following result shows that QuickLog achieves forward authenticity for generic F and G.

Theorem 4.1. *Let $F : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a PRF and let $G : \{0,1\}^n \times \mathrm{Dom} \to \{0,1\}^\tau$ be a one-time MAC. For an adversary $\mathcal{A}$ making q signing queries, we can construct an adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}_{\mathrm{QuickLog}[F,G]}^{\mathrm{fa}}(\mathcal{A}) \leq q \cdot \mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{B}) + \mathbf{Adv}_G^{\mathrm{mu-mac1}}(\mathcal{A}) \;.$$

Adversary $\mathcal{B}$ runs $\mathcal{A}$ and makes $2(q-1)$ calls to F, $q+1$ calls to G on the messages of $\mathcal{A}$, plus two PRF queries.

Proof. For each $u \in \{0,\ldots,q\}$, consider game G_u in Fig. 8. Game G_q corresponds to game $\mathbf{G}_{\mathrm{QuickLog}[F,G]}^{\mathrm{fa}}(\mathcal{A})$. In contrast, in game G_0, all the keys and states are sampled uniformly at random, and thus this game corresponds to game $\mathbf{G}_G^{\mathrm{mu-mac1}}(\mathcal{A})$.

We now construct the adversary $\mathcal{B}$. It picks $U \leftarrow_\$ \{1,\ldots,q\}$ and then runs $\mathcal{A}$. It tries to simulate game $G_U(\mathcal{A})$, but for the U-th signing query M, it will instead use its own PRF oracle

Game $G_u(\mathcal{A})$	procedure $\text{SIGN}(M)$
$S_0 \leftarrow\!\!\$\, \mathcal{S};\ v \leftarrow 0;\ (i,\sigma) \leftarrow\!\!\$\, \mathcal{A}^{\text{SIGN}}$	$v \leftarrow v+1;\ M_v \leftarrow M$
$(M',T') \leftarrow\!\!\$\, \mathcal{A}(S_i,\sigma)$	$K_v, S_v \leftarrow\!\!\$\, \{0,1\}^n$
$T^* \leftarrow G(K_i, M')$	If $v \leq u$ then
Return $(M' \neq M_i \text{ and } T' = T^*)$	$\quad K_v \leftarrow F(S_{v-1},[0]_n)$
	$\quad S_v \leftarrow F(S_{v-1},[1]_n)$
	$T_v \leftarrow G(K_v, M_v)$
	Return T_v

Figure 8: Game G_u in the proof of Theorem 4.1.

FN to compute $K_U \leftarrow \text{FN}([0]_n)$ and $S_U \leftarrow \text{FN}([1]_n)$. Then

$$
\begin{aligned}
\mathbf{Adv}_F^{\text{prf}}(\mathcal{B}) &= \mathbf{E}_{U \leftarrow\!\$\,\{1,\dots,q\}} \Big[\Pr[G_U(\mathcal{A})] - \Pr[G_{U-1}(\mathcal{A})] \Big] \\
&= \frac{1}{q} \sum_{u=1}^{q} \Pr[G_u(\mathcal{A})] - \Pr[G_{u-1}(\mathcal{A})] \\
&= \frac{1}{q} \Big(\Pr[G_q(\mathcal{A})] - \Pr[G_0(\mathcal{A})] \Big) \\
&= \frac{1}{q} \Big(\mathbf{Adv}_{\text{QuickLog}[F,G]}^{\text{fa}}(\mathcal{A}) - \mathbf{Adv}_G^{\text{mu-mac1}}(\mathcal{A}) \Big)
\end{aligned}
$$

as claimed. □

The following result establishes the security bound for QuickLog in the special case that F is the Even-Mansour construction $\text{EM}[\pi]$ and G is $\text{XMAC}[\pi,\tau]$.

Corollary 4.2. *Let* $\pi : \{0,1\}^n \to \{0,1\}^n$ *be a permutation that we will model as an ideal permutation. Let F be the Even-Mansour construction* $\text{EM}[\pi]$, *and let G be* $\text{XMAC}[\pi,\tau]$. *Consider an adversary $\mathcal{A}$ making q signing queries and at most p ideal-permutation queries, and its messages are of at most s blocks. Then*

$$
\mathbf{Adv}_{\text{QuickLog}[F,G]}^{\text{fa}}(\mathcal{A}) \leq \frac{6pq + 4q^2 + 6sq}{2^n} + \frac{1}{2^\tau} \ .
$$

Proof. From Theorem 4.1, we can construct an adversary $\mathcal{B}$ such that

$$
\mathbf{Adv}_{\text{QuickLog}[F,G]}^{\text{fa}}(\mathcal{A}) \leq q \cdot \mathbf{Adv}_F^{\text{prf}}(\mathcal{B}) + \mathbf{Adv}_G^{\text{mu-mac1}}(\mathcal{A}) \ .
$$

Adversary $\mathcal{B}$ makes at most $p + 2(q-1) + s$ ideal-cipher queries, and two PRF queries. From Lemma 2.1,

$$
\mathbf{Adv}_F^{\text{prf}}(\mathcal{B}) \leq \frac{2(p + 2q + s)}{2^n} \ .
$$

Moreover, from Theorem 3.2,

$$
\mathbf{Adv}_G^{\text{mu-mac1}}(\mathcal{A}) \leq \frac{4q(p+s)}{2^n} + \frac{1}{2^\tau} \ .
$$

Summing up,

$$
\mathbf{Adv}_{\text{QuickLog}[F,G]}^{\text{fa}}(\mathcal{A}) \leq \frac{6pq + 4q^2 + 6sq}{2^n} + \frac{1}{2^\tau}
$$

as claimed. □

INTERPRETING THE BOUND. At the first glance, the bound in Corollary 4.2 looks like a typical birthday bound (also known as 64-bit security) of many NIST standards, such as GCM [21]. However, a closer look reveals that it is much stronger. Note that here q is the number of log messages in a single auditing round. It is reasonable to assume that $q \leq 2^{30}$, as a study by Ma *et al.* [36] observes just 2.76 million logging events per day on average for a web server under realistic conditions. Under this assumption, for $n = 128$ (the block length of AES), the bound is roughly $(p+s)/2^{96}$, meaning that we actually have 96-bit security.

REMARK. Had we followed the route in [14] to reduce the forward authenticity of QuickLog to the *single-user* one-time MAC security of XMAC, we would have ended up with an interior term $q/2^\tau$. Our approach instead yields a much better term $1/2^\tau$. This allows one to use a short tag length (say 8 bytes) for QuickLog, saving storage cost.

4.3 The KennyLoggings System

A GLANCE AT KENNYLOGGINGS. KennyLoggings [39] also follows the blueprint of Bellare and Yee [14], but the state S_i and the key K_i are conflated. In particular, one derives $K_i \leftarrow H(K_{i-1})$ via a cryptographic hash function H that is instantiated via Blake2b [4]. To sign a message M under key K, one uses a PRF that is instantiated from SipHash [3].

To speed up performance, KennyLoggings *pre-computes* 200,000 keys, and refills when half of the keys are used and deleted. As a result, KennyLoggings has to erase a prior key by, say writing zeros to this memory location. One has to be careful to ensure that a compiler will not remove these seemingly useless writes [47]. In contrast, QuickLog maintains just a single key and state, and during an update, it writes the new key and state to the memory location of the old one. Since the new key and state will be used later, this naturally avoids compilers' optimization of eliminating dead stores.[3]

A GLITCH IN KENNYLOGGINGS. While Paccagnella *et al.* [39] claim that KennyLoggings inherits the security proof from the work of Bellare and Yee [14], the conflation of the key and state actually voids the security guarantee from [14]. One can still justify the security of KennyLoggings by modeling Blake2b as a (programmable) random oracle (instead of a pseudorandom generator as in [14]), but the proof is non-trivial.

5 QuickLog2: Aggregate Authentication

The syntax of logging protocols in Section 4.1 requires that each log message has a corresponding τ-bit tag. This substan-

---

[3] KennyLoggings cannot use the new keys to overwrite the old ones. In particular, key erasure has to be performed immediately after each signing. Thus the signing thread of KennyLoggings cannot wait until its key-generating thread to wake up to overwrite the old keys.

tially increases the costs of storing the logs and transmitting them to the auditor. In this section, we formalize the syntax and security definition for logging protocols with *aggregate authentication* [34]. In such a logging protocol, there is only a single τ-bit aggregate tag to authenticate all log messages, cutting costs in storage and transmission of the logs. Even better, the use of aggregate authentication can directly thwart the truncation attack without employing additional mechanisms as suggested in [39]. We then show how to extend QuickLog to another scheme QuickLog2 of aggregate authentication with negligible overheads.

5.1 Formalizing Security

SYNTAX. A *logging protocol* Π *with aggregate authentication* consists of a triple of deterministic algorithms $(\mathrm{Update}, \mathrm{Sign}, \mathrm{Merge})$ and is associated with a state space $\mathcal{S}$ and a tag length τ.

- Initially, a root state $S_0 \leftarrow_\$ \mathcal{S}$ is sampled and an aggregate tag T is initialized to 0^τ. Derive $(K_1, S_1) \leftarrow \mathrm{Update}(S_0)$, share S_0 with the auditor, and erase S_0 from the host's memory.

- When we need to sign the i-th log message M_i, we retrieve (S_i, K_i) and generate the tag $T_i \leftarrow \mathrm{Sign}(K_i, M_i)$. We then update $T \leftarrow \mathrm{Merge}(T, T_i)$ and $(K_{i+1}, S_{i+1}) \leftarrow \mathrm{Update}(S_i)$. Finally, we delete (K_i, S_i, T_i) from the host's memory.

- Given a root state S_0, messages $(M_1, \ldots, M_q)$, and an aggregate tag T, an auditor can verify the integrity of these logs by (1) initializing $T^* \leftarrow 0^\tau$, and (2) iteratively deriving $(K_i, S_i) \leftarrow \mathrm{Update}(S_{i-1})$, $T_i \leftarrow \mathrm{Sign}(K_i, M_i)$, and $T^* \leftarrow \mathrm{Merge}(T^*, T_i)$ for every $i \leq q$, and (3) checking if the final T^* is the same as the given T.

DEFINING SECURITY. For an adversary $\mathcal{A}$ attacking a logging protocol Π with aggregate authentication, we define its advantage in breaking the *forward aggregate authenticity* (FA2) of Π as

$$\mathbf{Adv}_{\Pi}^{\mathrm{fa2}}(\mathcal{A}) = \Pr[\mathbf{G}_{\Pi}^{\mathrm{fa2}}(\mathcal{A})] \ ,$$

where game $\mathbf{G}_{\Pi}^{\mathrm{fa2}}(\mathcal{A})$ is defined in Fig. 9. Initially the adversary generates log messages $(M_1, \ldots, M_q)$ and an internal state σ. The game then samples a state $S_0 \leftarrow_\$ \mathcal{S}$, and generates the corresponding aggregate tag T for $(M_1, \ldots, M_q)$. The adversary is then given back its internal state σ, the aggregate tag T, and the state S_q to produce a forgery $(M'_1, \ldots, M'_r, T')$. It wins the game if (1) the forged tag T' is exactly the aggregate tag of $(M'_1, \ldots, M'_r)$ for the state S_0, and (2) if $r \geq q$ then we require $(M_1, \ldots, M_q) \neq (M'_1, \ldots, M'_q)$ to avoid trivial wins.

We now give an intuition for why achieving the FA2 goal makes the logs tamper-proof. Suppose that when the adversary gets full control of the system, it obtains the state S_q (and thus knows all the subsequent keys and states) and the current aggregate tag T, but the system calls of its attack are already

---

Game $\mathbf{G}_{\Pi}^{\mathrm{fa2}}(\mathcal{A})$
$(M_1, \ldots, M_q, \sigma) \leftarrow_\$ \mathcal{A};\ \ S_0 \leftarrow_\$ \mathcal{S}$
$T \leftarrow \mathrm{Agg}(S_0, M_1, \ldots, M_q)$
$(M'_1, \ldots, M'_r, T') \leftarrow_\$ \mathcal{A}(S_q, T, \sigma)$
$T^* \leftarrow \mathrm{Agg}(S_0, M'_1, \ldots, M'_r)$
If $r < q$ then return $(T^* = T')$
Else return $(T^* = T') \wedge ((M_1, \ldots, M_q) \neq (M'_1, \ldots, M'_q))$
procedure $\mathrm{Agg}(S_0, M_1, \ldots, M_v)$
$T \leftarrow 0^\tau$
For $i \leftarrow 1$ to v do
$\quad (K_i, S_i) \leftarrow \mathrm{Update}(S_{i-1});\ \ T_i \leftarrow \mathrm{Sign}(K_i, M_i)$
$\quad T \leftarrow \mathrm{Merge}(T, T_i)$
Return T

Figure 9: Game defining the FA2 security of a logging protocol Π with aggregate authentication.

recorded in the logs $M_1, \ldots, M_q$. These messages are signed under the (now deleted) keys $K_1, \ldots, K_q$. Suppose that later an auditor receives the tampered messages $(M'_1, \ldots, M'_r)$ and an aggregate tag T'. The auditor will compute the corresponding aggregate tag T^* of $(M'_1, \ldots, M'_r)$ and compares it with T'. Without loss of generality, assume that if $r \geq q$ then we must have $(M'_1, \ldots, M'_q) \neq (M_1, \ldots, M_q)$, otherwise the auditor can find out the intrusion. The FA2 notion ensures that $T' \neq T^*$ (even if the adversary performs a truncation attack, meaning $r < q$), and thus the auditor can detect the tampering.

5.2 The QuickLog2 System

DESCRIPTION OF QUICKLOG2. A simple construction for the Merge algorithm, suggested by Ma and Tsudik [34], is via $T \leftarrow H(T, T_i)$, where H is a collision-resistant hash function such as SHA-2 or SHA-3. Unfortunately, if we incorporate this method into QuickLog, it will significantly increase both the signing and verification time. We instead use $T \leftarrow T \oplus T_i$, making the overhead of the aggregation negligible. Note that under our approach, the verification algorithm is fully parallelizable. In contrast, the method of Ma and Tsudik [34] forces the auditor to make a long chain of hashing.

The idea for the xor trick is from the work of Katz and Lindell [29] for aggregating MAC signatures. Their mechanism however has never been used for prior logging systems due to an obvious attack. In particular, for prior designs, one evolves keys over time intervals, and in each interval, multiple messages are signed under the same key. Thus the xor trick will fail to detect if log messages of the same interval are reordered. It however *does* work for our setting, as there is only a single message to sign per key. Still, proving that this method can cope with the truncation attack goes beyond what the abstraction of the xor trick in [29] can deliver. In particular, unlike the the application in [29] that merely requires the individual tags to be unpredictable, here we need them to be pseudoran-

```
procedure QuickLog2[F,G].Update(S)
S' ← F(S,[0]_n);  K' ← F(S,[1]_n);  Return (K',S')

procedure QuickLog2[F,G].Sign(K,M)
T ← G(K,M);  Return T

procedure QuickLog2[F,G].Merge(T,T_i)
Return T⊕T_i
```

Figure 10: The logging protocol QuickLog2 with aggregate authentication, built on top of a PRF F and a two-time PRF G. Here for each integer $a \in \{0, \ldots, 2^n - 1\}$, we let $[a]_n$ denote an n-bit encoding of a.

dom.[4] To realize this goal, we exploit the fact that XMAC is a two-time PRF (as defined in Section 3.3).

Formally, we extend QuickLog to a scheme QuickLog2 of aggregate authentication as shown in Fig. 10. The algorithms Sign and Update of QuickLog2 remain the same as those of QuickLog, and its Merge algorithm is built on top of the xor trick. We stress that the primitive G is now required to be a two-time PRF instead of merely a one-time MAC.

SECURITY OF QUICKLOG2. The following result shows that QuickLog2 achieves FA2 security for generic F and G.

Theorem 5.1. *Let* $F : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ *be a PRF and let* $G : \{0,1\}^n \times \mathrm{Dom} \to \{0,1\}^\tau$ *be a two-time PRF. For an adversary* $\mathcal{A}$ *making q signing messages and r forgery messages, we can construct an adversaries $\mathcal{B}$ and $\mathcal{D}$ such that*

$$\mathbf{Adv}^{\mathrm{fa2}}_{\mathrm{QuickLog}[F,G]}(\mathcal{A}) \leq q \cdot \mathbf{Adv}^{\mathrm{prf}}_F(\mathcal{B}) + q \cdot \mathbf{Adv}^{\mathrm{prf2}}_G(\mathcal{D}) + \frac{1}{2^\tau} \ .$$

Adversary $\mathcal{B}$ runs $\mathcal{A}$ and makes $2(q-1)$ calls to F, $q+r$ calls to G on messages of $\mathcal{A}$, plus two PRF queries. Adversary $\mathcal{D}$ runs $\mathcal{A}$ and makes $r+q-2$ calls to G on $r-1$ forgery messages and $q-1$ signing messages of $\mathcal{A}$, plus two PRF queries on the remaining messages.

Proof. Since the adversary $\mathcal{A}$ receives the state S_q, it can compute the subsequent keys K_i (with $i > q$). Thus without loss of generality, we assume that $\mathcal{A}$ only generates at most q forgery messages. Indeed, suppose that $\mathcal{A}$ outputs $(M'_1, \ldots, M'_r, T')$ with $r > q$. Then it could instead produce $(M'_1, \ldots, M'_q, T' \oplus T_{q+1} \oplus \cdots \oplus T_r)$ to win with the same advantage, where $T_i \leftarrow G(K_i, M'_i)$. Hence from now on, we assume that $r \leq q$. For each $u \in \{0, \ldots, q\}$, consider game G_u in Fig. 11. Game G_q corresponds to game $\mathbf{G}^{\mathrm{fa2}}_{\mathrm{QuickLog2}[F,G]}(\mathcal{A})$. In contrast, in game G_0, all the keys and states are sampled uniformly at random.

We now construct the adversary $\mathcal{B}$. It picks $U \leftarrow^\$ \{1, \ldots, q\}$ and then runs $\mathcal{A}$. It tries to simulate game $G_U(\mathcal{A})$, but will

---
[4]To see why, observe that under the truncation attack, the adversary is given the checksum $T_1 \oplus \cdots \oplus T_q$ of the individual tags $T_1, \ldots, T_q$. It then needs to produce some $r < q$, together with $T_1 \oplus \cdots \oplus T_r$. Thus the tag T_q acts as a one-time pad to protect $T_1, \ldots, T_{q-1}$.

```
Game G_u(A)
(M_1,...,M_q,σ) ←$ A;  S_0 ←$ S
T ← Agg(S_0,M_1,...,M_q)
(M'_1,...,M'_r,T') ←$ A(S_q,T,σ)
T* ← Agg(S_0,M'_1,...,M'_r)
If r < q then return (T* = T')
Else return (T* = T')∧ ((M_1,...,M_q) ≠ (M'_1,...,M'_q))

procedure Agg(S_0,M_1,...,M_v)
T ← 0^τ
For i ← 1 to v do
    K_i ← F(S_{i-1},[0]_n);  S_i ← F(S_{i-1},[1]_n)
    If (i > u) then K_i,S_i ←$ {0,1}^n
    T_i ← G(K_i,M_i);  T ← T⊕T_i
Return T
```

Figure 11: Game G_u in the proof of Theorem 5.1.

use its own PRF oracle FN to compute $K_U \leftarrow \mathrm{FN}([0]_n)$ and $S_U \leftarrow \mathrm{FN}([1]_n)$. (Note that by storing K_U and S_U, adversary $\mathcal{B}$ only needs to make two PRF queries.) Then

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{prf}}_F(\mathcal{B}) &= \mathbf{E}_{U \leftarrow^\$ \{1,\ldots,q\}} \Big[\Pr[G_U(\mathcal{A})] - \Pr[G_{U-1}(\mathcal{A})] \Big] \\
&= \frac{1}{q} \sum_{u=1}^q \Pr[G_u(\mathcal{A})] - \Pr[G_{u-1}(\mathcal{A})] \\
&= \frac{1}{q} \Big(\Pr[G_q(\mathcal{A})] - \Pr[G_0(\mathcal{A})] \Big) \\
&= \frac{1}{q} \Big(\mathbf{Adv}^{\mathrm{fa2}}_{\mathrm{QuickLog2}[F,G]}(\mathcal{A}) - \Pr[G_0(\mathcal{A})] \Big) \ .
\end{aligned}$$

To bound $\Pr[G_0(\mathcal{A})]$, consider games P_u in Fig. 12 for every $u \in \{0, \ldots, q\}$. Game P_q coincides with game G_0, whereas in game P_0, all the individual tags are sampled uniformly at random, subject to the condition that if $M'_i = M_i$ then they will have the same individual tag. Note that in game P_0, the correct aggregate tag T^* of the forgery messages is uniformly random over $\{0,1\}^\tau$, independent of whatever the adversary $\mathcal{A}$ receives. Thus the chance that it can guess T^* correctly is $2^{-\tau}$. In other words, $\Pr[P_0(\mathcal{A})] = 2^{-\tau}$.

We now construct an adversary $\mathcal{D}$. It picks $U \leftarrow^\$ \{1, \ldots, q\}$ and then runs $\mathcal{A}$. It tries to simulate game $P_U(\mathcal{A})$, but uses its PRF oracle to sign messages M_U and M'_U. If the simulated game returns true then $\mathcal{D}$ returns 1, otherwise it returns 0. Note that if G is built on top of ideal primitives (as in the case of XMAC) then this constructed adversary $\mathcal{D}$ will not query them after the second PRF query. Then

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{prf2}}_G(\mathcal{D}) &= \mathbf{E}_{U \leftarrow^\$ \{1,\ldots,q\}} \Big[\Pr[P_U(\mathcal{A})] - \Pr[P_{U-1}(\mathcal{A})] \Big] \\
&= \frac{1}{q} \sum_{u=1}^q \Pr[P_u(\mathcal{A})] - \Pr[P_{u-1}(\mathcal{A})] \\
&= \frac{1}{q} \Big(\Pr[P_q(\mathcal{A})] - \Pr[P_0(\mathcal{A})] \Big) \\
&= \frac{1}{q} \Big(\Pr[G_0(\mathcal{A})] - \frac{1}{2^\tau} \Big) \ .
\end{aligned}$$

```
Game P_u(A)
(M_1,...,M_q,σ) ←$ A;  S_0 ←$ S;  T ← 0^τ
For i ← 1 to q do
    S_i,K_i ←$ {0,1}^n;  T_i ← Sign(i,K_i,M_i);  T ← T⊕T_i
(M'_1,...,M'_r,T') ←$ A(S_q,T,σ);  T^* ← 0^τ
For i ← 1 to r do
    If M'_i = M_i then T_i^* ← T_i else T_i^* ← Sign(i,K_i,M'_i)
    T^* ← T^*⊕T_i^*
If r < q then return (T^* = T')
Else return (T^* = T') ∧ ((M_1,...,M_q) ≠ (M'_1,...,M'_q))
procedure Sign(i,K,M)
If i ≤ u then V ← G(K,M) else V ←$ {0,1}^n
Return V
```

Figure 12: Game P_u in the proof of Theorem 5.1.

Summing up,

$$\mathbf{Adv}^{\text{fa2}}_{\text{QuickLog2}[F,G]}(\mathcal{A}) = q \cdot \mathbf{Adv}^{\text{prf}}_F(\mathcal{B}) + q \cdot \mathbf{Adv}^{\text{prf2}}_G(\mathcal{D}) + \frac{1}{2^\tau}$$

as claimed. $\qquad\square$

In the case that F is the Even-Mansour construction $\mathsf{EM}[\pi]$ and G is $\mathsf{XMAC}[\pi,\tau]$, the following result derives the security bound for QuickLog2. The bound is comparable to that of QuickLog; that is, QuickLog2 also has 96-bit of security under the assumption that $q \le 2^{30}$.

Corollary 5.2. *Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Let F be the Even-Mansour construction $\mathsf{EM}[\pi]$, and let G be $\mathsf{XMAC}[\pi,\tau]$. Consider an adversary $\mathcal{A}$ making q signing queries and at most p ideal-permutation queries, and its messages are of at most s blocks. Then*

$$\mathbf{Adv}^{\text{fa2}}_{\text{QuickLog2}[F,G]}(\mathcal{A}) \le \frac{6pq + 4q^2 + 8sq}{2^n} + \frac{1}{2^\tau} \ .$$

Proof. From Theorem 5.1, we can construct an adversaries $\mathcal{B}$ and $\mathcal{D}$ such that

$$\mathbf{Adv}^{\text{fa2}}_{\text{QuickLog2}[F,G]}(\mathcal{A}) \le q \cdot \mathbf{Adv}^{\text{prf}}_F(\mathcal{B}) + q \cdot \mathbf{Adv}^{\text{prf2}}_G(\mathcal{D}) + \frac{1}{2^\tau} \ .$$

Adversary $\mathcal{B}$ makes at most $p + 2(q-1) + s$ ideal-cipher queries, and two PRF queries. Adversary $\mathcal{D}$ makes at most $p + s$ ideal-cipher queries, plus at most two PRF queries whose total block length is at most s. From Lemma 2.1,

$$\mathbf{Adv}^{\text{prf}}_F(\mathcal{B}) \le \frac{2(p+2q+s)}{2^n} \ .$$

Moreover, from Proposition 3.5,

$$\mathbf{Adv}^{\text{prf2}}_G(\mathcal{D}) \le \frac{4(p+s)+2s}{2^n} = \frac{4p+6s}{2^n} \ .$$

Summing up,

$$\mathbf{Adv}^{\text{fa2}}_{\text{QuickLog2}[F,G]}(\mathcal{A}) \le \frac{6pq + 4q^2 + 8sq}{2^n} + \frac{1}{2^\tau}$$

as claimed. $\qquad\square$

6 Implementation

We implement both QuickLog and QuickLog2 on the Linux kernel, version 3.10.0-1160, on top of Linux Audit, the standard system log collection framework of Linux. Below, we will briefly review the architecture of the Audit system.

THE AUDIT SYSTEM. The Audit system consists of two main components: kauditd (in the kernel) and auditd (in the user space). It applies a (rule-based) audit filter on each system call to check if a log is needed. The log message, if created, will be put into a buffer. Later, kauditd will *asynchronously* dequeue messages from the buffer and send them to auditd to create log entries. The race attack in [39] takes advantage of this asynchronous processing to modify a log message before it is committed to the log entries. To thwart this attack, following the approach in [39], we will sign each log message before it enters the buffer.

IMPLEMENTING QUICKLOG. In QuickLog, the initial 16-byte state S_0 is generated via the get_random_bytes function, a cryptographically secure source of randomness. At system startup, QuickLog will initialize S_0 and then derive the subsequent key-state pair (K_1, S_1). To do that, we modify the code of the audit_init function, which initializes the kernel components of Linux Audit.

We modify the code of the audit_log_end function to sign log messages right before they enter the buffer. In addition, we extend the spinlock that protects the existing critical section of the audit_log_end function to cover the signing operation.

As described in Fig. 7, QuickLog is based on a PRF and a one-time MAC; the former is instantiated from the Even-Mansour construction $\mathsf{EM}[\pi]$ and the latter from $\mathsf{XMAC}[\pi,\tau]$. Here the permutation π is AES-128 with the all-zero key, and the tag length τ is 64 (the same length as KennyLoggings). The tag length is somewhat short, compared to standard MAC constructions, but it is justified in Section 4.

The implementation of XMAC makes use of the Intel's SIMD vector extension and the AES-NI instruction set to maximize the speed.[5] Unfortunately, such use inside the kernel requires one to save the values of all floating-point unit registers via kernel_fpu_begin, and later restore them via kernel_fpu_end. This incurs a penalty of 100–120 ns for

---

[5] Since our experimental machine does not support 256-bit and 512-bit VAES, we use 128-bit AES-NI in the implementation. It is expected that using 256-bit or 512-bit VAES in newer CPUs will further improve the performance of QuickLog.

each signing operation. The verification operation is however implemented in the user space and is therefore not affected.

On the other hand, note that one is given lots of message-tag pairs to verify. By exploiting the pipelining of AES-NI, we improve the *amortized* speed of the verification operation.

IMPLEMENTING QUICKLOG2. Unlike QuickLog, for QuickLog2 we do not have to write individual tags T_i to the user space. We instead maintain a single aggregate tag T in the memory, and update $T \leftarrow T \oplus T_i$ when T_i is available. Since we only need to store a single aggregate tag, we use the full tag length $\tau = 128$ for QuickLog2.

7 Experiments

In this section, we measure and compare the performance of QuickLog, QuickLog2, and the optimized version of KennyLoggings (that has key precomputation). The unmodified Linux kernel (version 3.10.0-1160) is used as the baseline.

EXPERIMENT SETUP. We run experiments on a server with two Intel Xeon Gold 6240 processors. Each processor has 18 cores and 2.60 GHz base frequency. The server runs CentOS 7.8 with Linux 3.10.0-1160 kernel and has 192GB of DDR4 RAM. Following [39], we configure Linux Audit to log all forensics-related system calls via the same ruleset in [35, 38], and use a buffer whose capacity is 2^{20} entries. We however add an additional rule to filter proctitle logs, as the information they provide is redundant [43]:

- A proctitle log is supposed to record the full command line that triggers the corresponding log event. But if the corresponding process is invoked from, say a bash shell, then the corresponding proctitle log only records "bash".
- In the case that proctitle logs can capture the command-line information, this can also be found in the entries of `execve` syscall.

In fact, proctitle logs are never intended to have any forensic significance. A Linux kernel announcement [42] explicitly warns that:

> "Proctitle is controllable by userspace, and thus should not be trusted. It is meant as an aid to assist in debugging."

In the NAS benchmarks [5], the size of proctitle logs concentrates around 77B. Ridding them bumps the average log size from 199B to 310B, passing the break-even point 256B between QuickLog and KennyLoggings.

7.1 Microbenchmarks

HOW WE BENCHMARK. We first evaluate the application-independent performance of the Sign and Verify operations of QuickLog and QuickLog2, and compare that to KennyLoggings. To eliminate the effect of the spinlock on the Sign operation, we implement a kernel module and manually execute the Sign operations in that module. We measure the performance of each operation for several message sizes, from 64B to 384B, which covers the typical log size in our applications. For each message size, we run 10 iterations. For each iteration, we run the operation for 200,000 times, and measure the median latency. We report the median of those 10 median timings; their standard deviation is within 5% of the median.

For QuickLog, the reported running time of the Sign operation is the total time to sign a message, update the key and state, and append the tag to the log. The running time of the Verify operation however only includes the signing and updating time. For the Sign/Verify operations of QuickLog2, the running time consists of the signing time and the time to update the key, state, and aggregate tag. For the Sign operation of KennyLoggings, we report the total time to sign a message, erase the current key, and append the tag to the log; key precomputation cost is ignored. For the Verify operation of KennyLoggings, the running time includes both the signing and the key generation cost, since keys now have to be computed on the fly.

RESULTS. The experiment results are in Table 1 and Table 2. For the Sign operation, due to an FPU context-switching penalty, QuickLog is slightly slower than KennyLoggings on small data (256B and below). Once data are large enough so that the penalty is no longer the dominant cost, QuickLog starts to outperform KennyLoggings, but the performance gap is small for the typical log sizes. QuickLog2 is however much faster than both QuickLog and KennyLoggings for all message sizes, because it avoids appending tags to log messages.

	64B	128B	256B	320B	384B
KennyLoggings	276	307	362	391	417
QuickLog	325	348	366	386	403
QuickLog2	169	187	205	225	242

Table 1: Latencies (in ns) for the Sign operation.

	64B	128B	256B	320B	384B
KennyLoggings	417	462	529	576	601
QuickLog	44	53	73	91	98

Table 2: Latencies (in ns) for the Verify operation. QuickLog2 has exactly the same timing as QuickLog.

The Verify operation of QuickLog is implemented in the user space, and does not bear the context-switching penalty. Moreover, thanks to the pipelining of AES-NI, one can improve the amortized cost of QuickLog's verification by exploiting the fact that there are lots of data to verify. For example, it takes a naive implementation 95 ns to verify a 256-byte message, but our optimized code only needs 73 ns. QuickLog2

has about the same verification cost as QuickLog, because it only adds an extra xor operation. On the other hand, the verification of KennyLoggings includes the (somewhat expensive) key generation. As a result, for verification, KennyLoggings is 6–8 times slower than QuickLog/QuickLog2 for all message sizes.

7.2 System-call Benchmarks

HOW WE BENCHMARK. Next, we measure the effect of QuickLog, QuickLog2, and KennyLoggings on the execution time of individual system calls when logging is enabled. We first run the system calls on a single thread to measure the added latency. We then run those on multiple (4, 8, and 16) threads to study the effect on the spinlock contention as well as the overall additional work.

We choose the following system calls in our experiments: `open` (342-byte logs), `read` (280-byte logs), `write` (284-byte logs), and `stat` (304-byte logs). They are also used in [39] to evaluate KennyLoggings. Following [39], in the experiments, each thread will run for 10 iterations, In each iteration, we invoke the system call 100,000 times, and compute the median latency. We measure the median of those 10 median timings; their standard deviation is within 5% of the median. Based on these median timings, we report the relative overhead of QuickLog, QuickLog2, and KennyLoggings compared to the unmodified Linux kernel.

RESULTS. The benchmark results are shown in Fig. 13. When there is just a single thread, KennyLoggings has similar performance as QuickLog, because their signing costs are comparable for the log sizes in the benchmarks. As more cores are used, the performance gap between QuickLog and KennyLoggings widens. To understand this phenomenon, recall that KennyLoggings hides the (somewhat expensive) cost of key generation from the critical path by using an additional kernel thread. The cost of key generation is negligible if there are only a few threads, but it will be manifest in busy multi-threading environment. In contrast, QuickLog has no hidden computation, and is therefore more efficient in multi-threading environments. On the other hand, as expected, QuickLog2 significantly reduces the overhead, compared to both QuickLog and KennyLoggings.

7.3 Application Benchmarks

HOW WE BENCHMARK. To understand the system-wide impact of QuickLog, QuickLog2, and KennyLoggings in realistic situations, we evaluate them using a number of application benchmarks, including I/O-intensive benchmarks and CPU-intensive benchmarks.

The first test suite consists of the following I/O-intensive benchmarks: NGINX [27], apache2 [23], redis [31], and HAProxy [24]; the first three benchmarks are also used in [39]
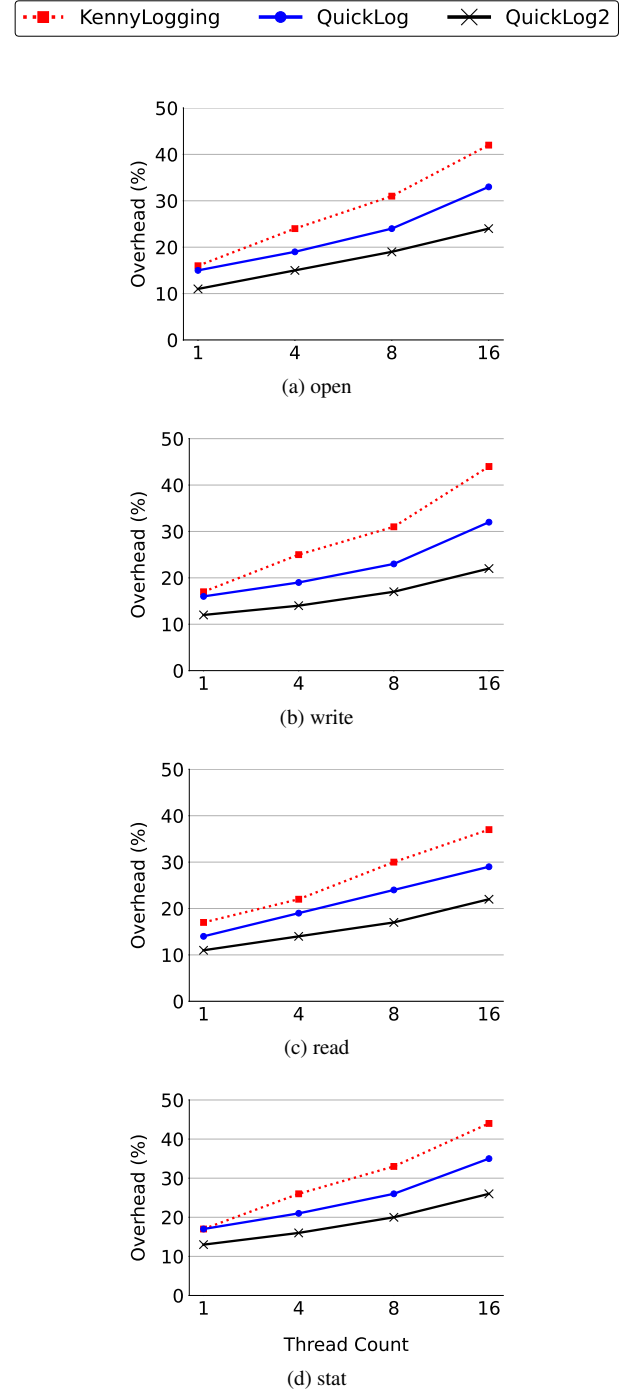
Figure 13: The relative overhead of QuickLog, QuickLog2, and KennyLoggings on system-call latency, compared to the unmodified Linux kernel.

to evaluate KennyLoggings. Each experiment consists of 10 iterations; we compute the median running time of those and make sure that the standard deviation is within 5% of the median. In each iteration, we use apache bench (for NGINX,

	Events/second	KennyLoggings	QuickLog	QuickLog2
NGINX	10,623	10.7	9.4	5.7
apache2	11,087	9.5	7.3	4.4
HAProxy	10,127	9.6	7.7	5.1
redis	3,223	8.8	8.1	5.9

Table 3: The relative overhead (%) of QuickLog, QuickLog2, and KennyLoggings compared to the unmodified Linux kernel in I/O-intensive benchmarks, together with the number of log events per second that they sign.

| | Vanilla Linux | KennyLoggings | | QuickLog | | QuickLog2 | |
	Time (s)	Time (s)	Overhead (%)	Time (s)	Overhead (%)	Time (s)	Overhead (%)
CG	0.19	0.22	12.50	0.21	9.37	0.2	3.65
FT	0.39	0.44	12.66	0.41	6.46	0.40	3.36
MG	0.23	0.27	14.22	0.26	9.90	0.24	5.17
LU	1.47	1.63	11.54	1.54	5.12	1.50	2.66
BT	1.85	2.02	9.52	1.95	5.52	1.91	3.46
SP	1.54	1.79	14.78	1.65	5.31	1.61	3.26
IS	0.20	0.23	15.50	0.22	8.51	0.21	4.00

Table 4: Median execution time of QuickLog, QuickLog2, and KennyLoggings with their relative overhead compared to the unmodified Linux kernel in the NAS parallel benchmarks.

apache2, and HAProxy) and redis-benchmark (for redis) to send 30,000 serial requests locally within the same machine. Based on the median timings, we report the relative overhead of QuickLog, QuickLog2, and KennyLoggings over the unmodified Linux kernel.

The second test suite consists of CPU-intensive programs from the NAS parallel benchmarks [5]. The High Performance Computing (HPC) workloads in the NAS parallel benchmarks allow us to evaluate the three secure logging systems in a realistic multi-threading environment. We run the CG, FT, MG, LU, BT, SP, and IS programs from the NAS parallel benchmarks on a single node. The problem size for all the benchmarks is Class A. BT and SP use 36 processes because these two programs require a perfect square number of processes. All other benchmarks use 32 processes because they require the number of processes to be a power of two. We perform each experiment 20 times, and report the median execution time as well as the corresponding relative overhead over the unmodified Linux kernel. For all cases, the standard deviation is within 5% of the reported median.

RESULTS FOR I/O-INTENSIVE BENCHMARKS. The results of the I/O-intensive benchmarks are shown in Table 3; the performance of KennyLoggings is consistent with the reported results in [39]. In all cases, QuickLog just slightly outperforms KennyLoggings because (1) their signing costs are comparable for the range of log sizes,[6] and (2) the key generation cost of KennyLoggings is negligible if an application

---

[6]The majority (83.2%) of log messages are from 281B to 339B; the remaining messages are rather short, from 60B to 165B.

uses only one or two threads, as reported in [39]. On the other hand, the signing time of QuickLog2 is at least 1.6 times faster than QuickLog for the log sizes in the benchmarks. As a result, QuickLog2's overhead is roughly 1.6 times smaller than that of QuickLog, for most cases.

RESULTS FOR CPU-INTENSIVE BENCHMARKS. The results for the NAS benchmarks are shown in Table 4. The overhead of KennyLoggings is more substantial in the busy multi-threading environment, as the hidden cost of key computation is now manifest. The average log message size is 310B in these programs; and most message sizes are between 296B and 388B. The signing latencies of QuickLog and KennyLoggings are comparable for these message sizes as shown in Table 1. Yet for most benchmarks, QuickLog's overhead is roughly half of that of KennyLoggings. Again, recall that the signing cost of QuickLog2 is at least 1.6 times faster than QuickLog for these log sizes. As a result, QuickLog2's overhead is at least 1.6 times smaller than that of QuickLog for all cases.

8 Conclusion

In this work, we build a secure logging system QuickLog2 that improves the state of the art, KennyLoggings [39], in several fronts: adoptability, performance, and security. Our implementation is open source and available at https://github.com/TsongW/QuickLog.git.

To achieve the goals above, we introduce a new cryptographic

primitive, one-time MAC, and realize it efficiently via the XMAC construction. While XMAC resembles the XOR-MAC construction [8], XMAC is simpler and faster because (i) it only needs to authenticate a single message per key, and (ii) it is built on top of AES with a fixed key, avoiding the cost of key setup and disrupting the AES-NI pipeline. We also realize that the xor trick in [29] can be used to aggregate MAC tags in our setting.

Our paper brings us another step forward in protecting system logs of Linux machines. Studying how to mount the race attack of [39] on Windows-based logging systems and building a Windows version of QuickLog2 is an important task for future work.

Acknowledgements

References

[1] Michelle Abraham and Christopher Kissel. Worldwide Security and Information Event Management Market Shares, 2020: SaaS-Focused Rise. Technical report, Splunk, 2020.

[2] Adil Ahmad, Sangho Lee, and Marcus Peinado. HARD-LOG: Practical Tamper-Proof System Auditing Using a Novel Audit Device. In *2022 IEEE Symposium on Security and Privacy*, 2022.

[3] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In Steven D. Galbraith and Mridul Nandi, editors, *INDOCRYPT 2012*, volume 7668 of *LNCS*, pages 489–508. Springer, Heidelberg, December 2012.

[4] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[5] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS Parallel Benchmarks. *Int. J. High Perform. Comput. Appl.*, 5(3):63–73, September 1991.

[6] Elaine Barker. NIST SP 800-175B. Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms. 2016.

[7] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 259–274. Springer, Heidelberg, May 2000.

[8] Mihir Bellare, Roch Guérin, and Phillip Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 15–28. Springer, Heidelberg, August 1995.

[9] Mihir Bellare and Viet Tung Hoang. Identity-based format-preserving encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1515–1532. ACM Press, October / November 2017.

[10] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society Press, May 2013.

[11] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[12] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 341–358. Springer, Heidelberg, August 1994.

[13] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006.

[14] Mihir Bellare and Bennet S. Yee. Forward-security in private-key cryptography. In Marc Joye, editor, *CT-RSA 2003*, volume 2612 of *LNCS*, pages 1–18. Springer, Heidelberg, April 2003.

[15] Tara Siegel Bernard, Tiffany Hsu, Nicole Perlroth, and Ron Lieber. Equifax Says Cyberattack May Have Affected 143 Million in the U.S. *The New York Times*, Sept 07, 2017.

[16] Eli Biham. How to decrypt or even substitute DES-encrypted messages in 2^{28} steps. *Inf. Process. Lett.*, pages 117–124, 2002.

[17] Erik-Oliver Blass and Stephan Marwedel. Secure logging with syslog-ng: Tamper evidence and confidentiality. In *Free and Open source Software Developers' European Meeting (FOSDEM 2020)*, 2020.

[18] Shan Chen and John P. Steinberger. Tight security bounds for key-alternating ciphers. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 327–350. Springer, Heidelberg, May 2014.

[19] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In Fabian Monrose, editor, *USENIX Security 2009*, pages 317–334. USENIX Association, August 2009.

[20] Orr Dunkelman, Nathan Keller, and Adi Shamir. Minimalism in cryptography: The Even-Mansour scheme revisited. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 336–354. Springer, Heidelberg, April 2012.

[21] Morris J. Dworkin. NIST SP 800-38D. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. 2007.

[22] Shimon Even and Yishay Mansour. A construction of a cipher from a single pseudorandom permutation. In Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, editors, *ASIACRYPT'91*, volume 739 of *LNCS*, pages 210–224. Springer, Heidelberg, November 1993.

[23] Apache Software Foundation. Apache2. `https://httpd.apache.org/`. Accessed on 2021-12-7.

[24] LLC HAProxy Technologies. HAProxy 1.5.18. `http://www.haproxy.org/`. Accessed on 2021-12-7.

[25] Viet Tung Hoang and Stefano Tessaro. Key-alternating ciphers and key-length extension: Exact bounds and multi-user security. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2016.

[26] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott D. Stoller, and V. N. Venkatakrishnan. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 487–504. USENIX Association, August 2017.

[27] Nginx Inc. NGINX 1.20.1. `https://www.nginx.com/`. Accessed on 2021-12-7.

[28] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. SGX-log: Securing system logs with SGX. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *ASIACCS 17*, pages 19–30. ACM Press, April 2017.

[29] Jonathan Katz and Andrew Y. Lindell. Aggregate message authentication codes. In Tal Malkin, editor, *CT-RSA 2008*, volume 4964 of *LNCS*, pages 155–169. Springer, Heidelberg, April 2008.

[30] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F. Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. MCI : Modeling-based causality inference in audit logging for attack investigation. In *NDSS 2018*. The Internet Society, February 2018.

[31] Redis Labs. redis 6.2.6. `https://redis.io/`. Accessed on 2021-12-7.

[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 973–990. USENIX Association, August 2018.

[33] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS 2018*. The Internet Society, February 2018.

[34] Di Ma and Gene Tsudik. A new approach to secure logging. *ACM Transactions on Storage (TOS)*, 5(1):1–21, 2009.

[35] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 1111–1128. USENIX Association, August 2017.

[36] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS 2016*. The Internet Society, February 2016.

[37] Giorgia Azzurra Marson and Bertram Poettering. Practical secure logging: Seekable sequential key generators. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 111–128. Springer, Heidelberg, September 2013.

[38] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In

Network and Distributed System Security Symposium (NDSS 2020), 2020.

[39] Riccardo Paccagnella, Kevin Liao, Dave Tian, and Adam Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *CCS 2020*, pages 1551–1574, 2020.

[40] Jacques Patarin. The "coefficients H" technique (invited talk). In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *SAC 2008*, volume 5381 of *LNCS*, pages 328–345. Springer, Heidelberg, August 2009.

[41] Tobias Pulls and Roel Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. In Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors, *ESORICS 2015, Part II*, volume 9327 of *LNCS*, pages 622–641. Springer, Heidelberg, September 2015.

[42] William Roberts. [PATCH v5 3/3] audit: Audit proc/<pid>/cmdline aka proctile. The Linux Kernel mailing list, February 2014. https://lkml.org/lkml/2014/2/6/353.

[43] Richard Roth. Reproducibility by Ontological representation of Provenance. Master's thesis, TU Wien, 2018.

[44] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In Aviel D. Rubin, editor, *USENIX Security 98*. USENIX Association, January 1998.

[45] VMware Carbon Black. Global Incident Response Threat Report - The Ominous Rise of "Island Hopping" & Counter Incident Response Continues. Technical report, 2019.

[46] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981.

[47] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. Dead store elimination (still) considered harmful. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 1025–1040. USENIX Association, August 2017.

A   Proof of Theorem 3.6

Without loss of generality, assume that if $\mathcal{A}$ outputs a fogrery (i, M', T') then M' is different from the i-th signing message. We construct $\mathcal{B}$ as follows.

Game $G_u(\mathcal{A})$	**procedure** SIGN(M)
$K_1, \ldots, K_q \leftarrow\!\!{}_\$ \mathcal{K}()$	$v \leftarrow v + 1; \; M_v \leftarrow M$
$v \leftarrow 0; \; (i, M', T') \leftarrow\!\!{}_\$ \mathcal{A}^{\text{SIGN}}$	$T_v \leftarrow F(K_v, M_v)$
If $i > u$ then $T^* \leftarrow\!\!{}_\$ \text{Rng}$	If $v > u$ then $T_v \leftarrow\!\!{}_\$ \text{Rng}$
Else $T^* \leftarrow F(K_i, M')$	Return T_v
Return $(M' \neq M_i \text{ and } T' = T^*)$	

Figure 14: Games for the proof of Proposition 3.6.

- It picks $U \leftarrow\!\!{}_\$ \{1, \ldots, q\}$, and simulates game $\mathbf{G}_F^{\text{mu-mac1}}(\mathcal{A})$. For the v-th query SIGN(M), if $i < U$ then $\mathcal{B}$ picks a key $K_v \leftarrow\!\!{}_\$ \mathcal{K}$ and returns to $\mathcal{A}$ the tag $T_v \leftarrow F(K_v, M)$. If $v = U$ then $\mathcal{B}$ will return $T_v \leftarrow \text{FN}(M)$ to $\mathcal{A}$. Otherwise, if $v > U$ then $\mathcal{B}$ picks $T_v \leftarrow\!\!{}_\$ \text{Rng}$ and returns T_v to $\mathcal{A}$.

- When $\mathcal{A}$ outputs (i, M', T'), if $i < U$ then $\mathcal{B}$ computes $T^* \leftarrow F(K_i, M')$. If $i = U$ then $T^* \leftarrow \text{FN}(M')$. Otherwise, if $i > U$ then $T^* \leftarrow\!\!{}_\$ \text{Rng}$.

- Finally $\mathcal{B}$ returns $(T' = T^*)$.

To analyze the advantage of $\mathcal{B}$, for each $u \in \{0, 1, \ldots, q\}$ let G_u be defined as in Fig. 14. Then

$$
\begin{aligned}
\mathbf{Adv}_F^{\text{prf2}}(\mathcal{B}) &= \mathbf{E}_{U \leftarrow\!\!{}_\$ \{1, \ldots, q\}} \Big[\Pr[G_U(\mathcal{A})] - \Pr[G_{U-1}(\mathcal{A})] \Big] \\
&= \frac{1}{q} \sum_{v=1}^{q} \Pr[G_u(\mathcal{A})] - \Pr[G_{u-1}(\mathcal{A})] \\
&= \frac{1}{q} \Big(\Pr[G_q(\mathcal{A})] - \Pr[G_0(\mathcal{A})] \Big) \ .
\end{aligned}
$$

Note that in game G_0, the tag T^* is sampled uniformly from Rng, independent of whatever $\mathcal{A}$ receives. Thus $\Pr[G_0(\mathcal{A})] = 1/|\text{Rng}|$. On the other hand, game $G_q(\mathcal{A})$ coincides with game $\mathbf{G}_F^{\text{mu-mac1}}(\mathcal{A})$. Hence

$$
\begin{aligned}
\mathbf{Adv}_F^{\text{prf2}}(\mathcal{B}) &= \frac{1}{q} \Big(\Pr[\mathbf{G}_F^{\text{mu-mac1}}(\mathcal{A})] - \frac{1}{|\text{Rng}|} \Big) \\
&= \frac{1}{q} \Big(\mathbf{Adv}_F^{\text{mu-mac1}}(\mathcal{A}) - \frac{1}{|\text{Rng}|} \Big) \ .
\end{aligned}
$$

In other words,

$$
\mathbf{Adv}_F^{\text{mu-mac1}}(\mathcal{A}) = q \cdot \mathbf{Adv}_F^{\text{prf2}}(\mathcal{B}) + \frac{1}{|\text{Rng}|} \ .
$$

B   Proof of Proposition 3.4

Without loss of generality, assume that the forgery message of $\mathcal{A}$ is different from the signing message. We construct $\mathcal{B}$ as follows. It runs $(M, \sigma) \leftarrow\!\!{}_\$ \mathcal{A}$ and queries $T \leftarrow \text{FN}(M)$. It then runs $(M', T') \leftarrow\!\!{}_\$ \mathcal{A}(T, \sigma)$ and queries $T^* \leftarrow \text{FN}(M')$. If $T^* = T'$ then $\mathcal{B}$ then outputs 1, indicating that it is in the real world. Otherwise it outputs 0, indicating that it is in the ideal world. Note that if F is built on top of an ideal permutation π

19

then since $\mathcal{A}$ stops querying π and π^{-1} after producing the forgery (M', T'), adversary $\mathcal{B}$ also stops querying them after the second PRF query.

We now analyze the advantage of $\mathcal{B}$. Let b be the challenge bit of game $\mathbf{G}_F^{\text{prf2}}$. Then on the one hand,

$$\Pr[\mathbf{G}_F^{\text{prf2}}(\mathcal{B}) \Rightarrow 1 \mid b = 1] = \Pr[\mathbf{G}_F^{\text{mac1}}(\mathcal{A})] = \mathbf{Adv}_F^{\text{mac1}}(\mathcal{A}) \ . \tag{3}$$

On the other hand, if $b = 0$ then $T^* \leftarrow\!\!\$\, \text{Rng}$, independent of what $\mathcal{A}$ observes (and in particular, independent of T'). Thus

$$\Pr[\mathbf{G}_F^{\text{prf2}}(\mathcal{B}) \Rightarrow 1 \mid b = 0] = \Pr[T' = T^*] = \frac{1}{|\text{Rng}|} \ . \tag{4}$$

Subtracting Eq. (4) from Eq. (3) yields the claimed bound.

C Proof of Proposition 3.5

Because we consider a computationally unbounded adversary, without loss of generality, assume that the adversary is deterministic, and never makes *redundant* queries. That is, it will not repeat a prior (PRF or ideal-permutation) query, and if it queries $\pi(x)$ to get an answer y, then later it will not query $\pi^{-1}(y)$, and vice versa. Assume further that the adversary makes exactly two PRF queries. Our proof is based on the H-coefficient technique.

<u>SETUP.</u> Let M and M' be the messages in the PRF queries that are reordered such that $|M| \geq |M'|$. When we run $\text{XMAC}[\pi, \tau](K, M)$, we will iteratively call $\pi(X_1 \oplus K), \ldots, \pi(X_m \oplus K)$, for distinct n-bit strings $X_1, \ldots, X_m$. Likewise, when we run $\text{XMAC}[\pi, \tau](K, M')$, we will iteratively call $\pi(X_1' \oplus K), \ldots, \pi(X_s' \oplus K)$ for distinct n-bit strings $X_1', \ldots, X_s'$, with $s \leq m$ and $s + m \leq \ell$. Parse M (with zero-padding if necessary) into $(n - r)$-bit blocks: $B_1 \cdots B_m \leftarrow M$. Likewise, parse $B_0' \cdots B_s' \leftarrow M'$.

We claim that we can pick a string $V \in \{X_1, \ldots, X_m\}$ such that $V \notin \{X_1', \ldots, X_s'\}$. In particular, let $i \leq s$ be the first index that $X_i \neq X_i'$, and let $V = X_i$. If this index i does not exist then let $V = X_m$. We will show that the choice above of V ensures that $V \notin \{X_1', \ldots, X_s'\}$. We consider the following cases.

Case 1: There is some index $i \leq s$ such that $X_i \neq X_i'$, and $V = X_i$. Note that $X_i \neq X_k'$ for every $k < i$ because the r-bit prefix of X_k' encodes the number k, whereas the number encoded in the r-bit prefix of X_i is at least $i > k$. If $i = s$ then we are done. Suppose that $i < s$ (and thus $i < m$). Then $X_i \neq X_t'$ for every $t \in \{i+1, \ldots, s\}$, because the r-bit prefix of X_i encodes the number i, whereas the encoded number in the r-bit prefix of X_t' is at least $t > i$. As a result, $X_i \notin \{X_1', \ldots, X_s'\}$.

Case 2: $X_j = X_j'$ for every $j \leq s$, and $s < m$. Then $X_m \neq X_k'$ for every $k \leq s$, because the encoded number in the r-bit prefix of X_m is at least m, whereas the r-bit prefix of $X_k' = X_k$ encodes the number $k \leq s < m$. As a result, $X_m \notin \{X_1', \ldots, X_s'\}$.

Case 3: $X_j = X_j'$ for every $j \leq s$, and $s = m$. Let u be the number of bytes to pad the (possibly fragmentary) B_m. Then

$$X_m = [m + u]_r \parallel B_m 0^{8u} \ .$$

Since $X_m' = X_m$, it means that $X_m' = [m + u]_r \parallel B_m' 0^{8u}$, and thus $B_m' = B_m$. For any $k < m$, we have $X_k = [k]_r \parallel B_k$ and $X_k' = [k]_r \parallel B_k'$, and thus $B_k' = B_k$. Hence $M' = M$ but this violates the assumption that $\mathcal{B}$ never repeats a prior query.

<u>DEFINING TRANSCRIPTS.</u> After the adversary finishes querying, in the real world, we will grant it the signing key K. In the ideal world, we will instead give it a fresh random key $K \leftarrow\!\!\$\, \{0, 1\}^n$. We also give the adversary the full n-bit outputs (instead of the τ-bit truncation) of the PRF oracle. Additional, we grant the adversary free queries $\pi(Z \oplus K)$ for every $Z \in \{X_1, \ldots, X_m, X_1', \ldots, X_s'\} \setminus \{V, X_1'\}$ at the end. This revelation can only help the adversary. Thus the transcript consists of the revealed key and the following information:

- **Ideal-permutation queries (including granted ones):** for a query $\pi(x)$ with answer y, we create a corresponding entry (prim, x, y) in the transcript. Likewise, for a query $\pi^{-1}(y)$ with answer x, we create a corresponding entry $\text{prim}(x, y)$. Note that we do not store the direction (forward or backward) of these queries, as only inputs and outputs are useful for $\mathcal{A}$. Note that there are at most $p + \ell$ entries (prim, x, y).

- **PRF queries:** we create entries (eval, M, T) and (eval, M', T'), where T and T' are the (full n-bit) answers for queries $\text{FN}(M)$ and $\text{FN}(M')$ respectively.

At the first glance, it is tempting to consider a transcript bad if there is an entry (prim, x, y) out of p queries and some $U \in \{X_1, \ldots, X_m, X_1', \ldots, X_s'\}$ such that $U \oplus K = x$. Doing so will however lead to an inferior bound $p\ell/2^n$. To obtain the strong bound $(4p + 2\ell)/2^n$, we instead say that a transcript is *bad* if one of the following happens:

1. Bad_0: There is some entry (prim, x, y) such that $x \in \{V \oplus K, X_s' \oplus K\}$. Intuitively, if the interaction of $\mathcal{B}$ with the real system results in a good transcript, the one-time pads $\pi(V \oplus K)$ and $\pi(X_1' \oplus K)$ will make the answers of the PRF queries look like uniformly random strings in the adversary's view.

2. Bad_1: There is some entry (prim, x, y) such that $y \in \{R, R'\}$, where the strings R and R' are computed from the transcript as follows. Let

$$R' = T' \oplus K \oplus \pi(X_1' \oplus K) \oplus \cdots \oplus \pi(X_{s-1}' \oplus K) \ .$$

In the real system, the string R' corresponds to $\pi(X_s' \oplus K)$. Next, let $Y_1, \ldots, Y_t$ be the elements of $\{X_1, \ldots, X_m\} \setminus \{V, X_s'\}$, and let

$$R \leftarrow T \oplus K \oplus \pi(Y_1 \oplus K) \oplus \cdots \oplus \pi(Y_t \oplus K) \oplus S \ ,$$

where $S = R'$ if $X'_s \in \{X_1, \ldots, X_m\}$, and $S = 0^n$ otherwise. In the real system, the string R corresponds to $\pi(V \oplus K)$. Intuitively, conditioning that Bad_0 does not happen, Bad_1 cannot happen in the real system, but it may happen in the ideal system.

If a transcript is not bad and is valid for the ideal system then we say that it is *good*.

<u>PROBABILITY OF BAD TRANSCRIPTS.</u> We now bound $\Pr[\mathsf{Bad}_0 \cup \mathsf{Bad}_1]$ in the ideal system. From the union bound,

$$\Pr[\mathsf{Bad}_0 \cup \mathsf{Bad}_1] \leq \Pr[\mathsf{Bad}_0] + \Pr[\mathsf{Bad}_1] \ .$$

Recall that for the granted queries $\pi(Z \oplus K)$, we have $Z \notin \{V, X'_s\}$. Thus to bound $\Pr[\mathsf{Bad}_0]$, we only need to consider non-granted prim entries. In the ideal system, since the revealed key $K \leftarrow\!\!\$ \{0,1\}^n$ is independent of the adversary's interaction with the system, the chance that there is one entry (prim, x, y) among the p non-granted ones such that $x \in \{V \oplus K, X'_s \oplus K\}$ is at most $2p/2^n$, and thus

$$\Pr[\mathsf{Bad}_0] \leq \frac{2p}{2^n} \ .$$

To bound $\Pr[\mathsf{Bad}_1]$, note that the revealed key $K \leftarrow\!\!\$ \{0,1\}^n$ is independent of the remainder of the transcript and thus the chance that there is an entry (prim, x, y) such that $y = R'$ is at most $(p + \ell)/2^n$. We then consider the following cases.

Case 1: $\mathcal{B}$ queries M' before querying M to the FN oracle. Since the adversary stops querying the ideal permutation after querying $\mathrm{FN}(M')$, the output $T \leftarrow\!\!\$ \{0,1\}^n$ is independent of the remainder of the transcript. As a result, the probability that there is an entry (prim, x, y) such that $y = R$ is at most $(p + \ell)/2^n$.

Case 2: $\mathcal{B}$ queries M before querying M' to the FN oracle, and $X'_s \notin \{X_1, \ldots, X_m\}$. Then

$$R = T \oplus K \oplus \pi(Y_1 \oplus K) \oplus \cdots \oplus \pi(Y_t \oplus K) \ .$$

As $K \leftarrow\!\!\$ \{0,1\}^n$ is independent of the remainder of the transcript the chance that there is an entry (prim, x, y) such that $y = R$ is at most $(p + \ell)/2^n$.

Case 3: $\mathcal{B}$ queries M before querying M' to the FN oracle, and $X'_s \in \{X_1, \ldots, X_m\}$. Then

$$\begin{aligned} R &= T \oplus K \oplus \pi(Y_1 \oplus K) \oplus \cdots \oplus \pi(Y_t \oplus K) \oplus R' \\ &= T \oplus T' \oplus \bigoplus_{i=1}^{t} \pi(Y_i \oplus K) \oplus \bigoplus_{j=1}^{s-1} \pi(X'_j \oplus K) \ . \end{aligned}$$

Since the adversary stops querying the ideal permutation after querying $\mathrm{FN}(M')$, the output $T' \leftarrow\!\!\$ \{0,1\}^n$ is independent of the remainder of the transcript. As a result, the chance that there is an entry (prim, x, y) such that $y = R$ is at most $(p + \ell)/2^n$.

Hence in any case,

$$\Pr[\mathsf{Bad}_1] \leq \frac{2(p + \ell)}{2^n} \ .$$

Summing up,

$$\Pr[\mathsf{Bad}_0 \cup \mathsf{Bad}_1] \leq \Pr[\mathsf{Bad}_0] + \Pr[\mathsf{Bad}_1] \leq \frac{4p + 2\ell}{2^n} \ . \quad (5)$$

<u>TRANSCRIPT RATIO.</u> Fix a good transcript θ. Suppose that there are exactly Q distinct entries (prim, x, y) in θ. Then

$$\mathsf{p}(\mathbf{S}_0) = \frac{1}{2^n} \cdot \left(\prod_{i=0}^{Q-1} \frac{1}{2^n - i} \right) \cdot \frac{1}{2^{2n}} \ ,$$

where (i) the first term corresponds to the chance that the revealed key matches the transcript, (ii) the second term corresponds to the chance that the answers of the ideal-permutation queries match the transcript, and (iii) the last term is the the chance that the two answers of the PRF queries match the transcript. In contrast, since θ is good,

$$\mathsf{p}(\mathbf{S}_1) = \frac{1}{2^n} \cdot \left(\prod_{i=0}^{Q-1} \frac{1}{2^n - i} \right) \cdot \frac{1}{(2^n - Q)(2^n - Q - 1)} \ .$$

Hence

$$1 - \frac{\mathsf{p}(\mathbf{S}_1)}{\mathsf{p}(\mathbf{S}_0)} \leq 0 \ . \quad (6)$$

<u>WRAPPING UP.</u> Combining Eq. (5) and Eq. (6), from Lemma 2.2,

$$\mathbf{Adv}_{\mathsf{XMAC}[\pi, \tau]}^{\mathsf{prf2}}(\mathcal{B}) \leq \frac{4p + 2\ell}{2^n}$$

as claimed.