Unintended Features of APIs: Cryptanalysis of Incremental HMAC

Gal Benmocha, Eli Biham, and Stav Perle

Computer Science Department
Technion – Israel Institute of Technology
{gal.benmocha,biham,stavp}@cs.technion.ac.il

Abstract. Many cryptographic APIs provide extra functionality that was not intended by the designers. In this paper we discuss such an unintended functionality in the API of HMAC, and study the security implications of it's use by applications.
HMAC authenticates a single message at a time with a single authentication tag. However, most HMAC implementations do not complain when extra data is added to the stream after that tag is computed, nor they undo the side effects of the tag computation. Think of it as an API of a new authentication primitive, that provides tags to prefixes, rather than just to the full message. We call such primitives Incremental MACs (IncMACs). IncMACs may be used by applications to efficiently authenticate long messages, broken into fragments, which need their own individual authentication tag for performing an early abort or to retransmit only bad fragments, while each tag (strongly) authenticates the message prefix so far, and the last tag fully authenticates the full message.
It appears that some applications (e.g., Siemens S7 protocol) use the standard HMAC API to provide an incremental MAC, allowing to identify transmission errors as soon as the first error occurs, while also directly authenticating the full message. We discuss two common implementations, used by cryptographic libraries and programs, whose APIs do not forbid using them incrementally, continuing with extra data after computing the tag. The most common one, which Siemens uses, uses a *naive* implementation (as natively coded from the RFCs). The other is the implementation of the OpenSSL library.
We discuss these implementations, and show that they are not as secure as HMAC. Moreover, some of them may even be highly insecure when used incrementally, where in the particular case of OpenSSL it is possible to instantly find collisions and multi-collisions, which are also colliding under any key. We also discuss the fine details of the definition of IncMACs, and propose secure versions of such a primitive.

1 Introduction

It is well known that attackers are keen to use oversights of designers of security systems. In particular, they like to use features that designers did not intend to make accessible. This state is a security threat since in many cases when

naive implementers incorrectly use features in a wrong way, attackers may be able to build their attacks on top of these naive mistakes. Such attacks may be specially attractive when the attackers can use oversights in a standard cryptographic API, where application developers call an API function in a situation that the designers did not foresee, and did not check for errors but which may be attractive for the calling application.

In this paper we show that most HMAC implementations have oversights that may be used by application developers. In particular, most HMAC implementations allow to call further API functions after a digest is computed, though the designers of HMAC considered that the digest calculation is the last activity in the HMAC computation. We also show that usage of this oversight may be a security catastrophe, and discuss several such cases. The most common HMAC implementation, when used incrementally, such as by Siemens in their S7 protocol of their industrial control systems, is not as secure as HMAC. Another one, used in OpenSSL, is highly insecure when used incrementally. It has an instant attack for finding key-independent collisions (collisions that hold for any key). We also discuss forgery, key-recovery and other attacks.

1.1 Authentication of Fragmented Messages

In networks communication, messages are typically fragmented, and each fragment is sent separately. If authentication of a full message is needed, the authentication tag is computed by a Message Authentication Code (MAC) on the full message, and appended to the end of the message. The problem is that only after the entire message is received the receiver can verify that the message has not been modified. Moreover, the receiver has no clue which of the fragments cause the authentication error, thus the whole message should be retransmitted. If the receiver had the ability to verify authenticity of a fragment immediately after receiving it, he could have performed an early abort, saving reception of many further fragments and much resources needed to keep and process them, or request to retransmit only the non-matching fragments. Therefore, most network protocols authenticate each fragment individually (e.g., IPSec [14]).

Some protocol designers prefer incremental authentication of the fragments (authentication of each fragment in context of the prefix of the message received so far). Since there is no standard for incremental authentication of fragmented messages, developers that need such functionality design their own mode or primitive based on existing implementations. In many cases, it performs an incorrect usage of existing MAC primitives (such as HMAC [5]) by calling their API in an unintended way. It results with an unexpected behavior of the MAC code. An example that uses the HMAC code applies HMAC on the first fragment (using init, update, and then finalize). It then calls update with the next fragment (without an extra init, while the context still depends on the prior fragment) and then another finalize to fetch the next digest. As the designers of HMAC (like the designers of hash functions) did not anticipate that a second fragment is to be added after finalize, various implementations give different (unexpected) results in this case. Unless well planed in advance by the programmer, using such

a primitive based on misuse of the API of HMAC is typically a bad solution, as we show later.

1.2 IncMACs (Incremental MACs)

In this paper we discuss incremental authentication and the security and efficiency of its implementations. For this purpose, we define IncMACs (Incremental MACs). They are similar to MACs, but with incremental authentication tags. I.e., for a message M divided to t fragments $M = (M_1, ..., M_t)$, there is an authentication tag for authenticating each prefix $M_1||...||M_i$ (the fragment M_i, $i \in \{1, \ldots, t\}$ together with all prior fragments). The security requirements on these many tags should be similar to those of the single tag of MACs, i.e., each tag should authenticate the full prefix. In some cases, e.g., when transferring records of a database, it is also advantageous to authenticate the division of the message to fragments. We call such a primitive a Fragment-Protecting IncMAC (FP-IncMAC).

We emphasis that it is crucial that each tag authenticates the whole prefix, rather than only the last fragment. One may claim that once each packet is authenticated, the list of tags authenticates the full message. This is true when the protocol that uses the MAC includes a session number and a sequence number in each fragment. But once the MAC is used incrementally as a single primitive, these extra additions are not there. Instead, unless we require each tag to authenticate the whole prefix, it may be the case that an attacker would be able to combine tags from different messages together in a way that will not be identified – and this would be potentially possible just because there is no requirement that strongly connects a tag to the whole prefix. It is therefore that the each tag must always authenticates the full prefix, and thus there is no reason to remember prior tags once a successor is received. Usage of IncMAC primitives provides a strong sense of authentication of the message, transmitted as many fragments. Unlike in common authentication of fragmented messages where each fragment is authenticated independently, IncMACs ensure that any authentication tag authenticates the message prefix. Therefore, no attacks based on replacing individual fragments are possible, independently of any (bad) choice of session IDs.

1.3 The Common (Native) API of Hash Functions and HMAC

Consider how a native implementation of a Merkle-Damgård hash function [9] is typically implemented (and described in the RFCs), starting with MD4 [20], through MD5 [21] to SHA-1 [17] and SHA-2 [18]. Basically, the hash function iterates calls to a compression function on each of the blocks of the input, where it outputs an intermediate value as its output, which is then used together with the next block as the input to the next call to the compression function. The API includes three functions: init, update and finalize that use a context record with their internal data. The context typically includes the number of bits hashed so far, a chaining value, and a block-sized array that is used to keep partial blocks

between calls to update. The init function initializes the context record with zero bits so far and the standard IV. The update function accepts the context record and a byte string and calls the compression function on each block of the input. If a partial block remains at the end, it is kept in the context record, and is processed along with the next input in the next call to update. The finalize function adds the padding to the partial block in the context (sometimes requires an additional block) and compresses the padded block(s). The output of the last compression function is the output of the hash function. A pseudocode of this structure is given in Listing 1.1. A typical implementation of this structure is given in Listing 1.2.[1]

Since finalize is intended to be called last, implementations can arbitrarily choose whether they modify the context state or not during it's calculation, i.e., whether finalize adds the padding directly into the partial block kept in the context, and updates the final intermediate value into the chaining value of the context, or work on a local copy without affecting the context. As a matter of fact, most implementations chose the former case, as done in the RFCs [20], i.e., finalize writes over the context. We call this strategy the *Naive* hash implementation. This strategy is very reasonable as once finalize is called no further calls to update or finalize should be made. But if such calls are made anyway, there is a difference in the behaviour of both strategies.

The API of HMAC also includes the same three functions: init, update and finalize. The init function initializes a context record, which includes two hash contexts that serve for the inner stream and the outer stream, and mixes the key into them. The update function accepts the context record and a byte string, and calls to the hash update of the inner stream. The finalize function make two calls to the hash finalize. The first call operates on the inner stream, and the second on the outer stream. In between, update is called on the outer with the digest of the inner. The digest value of the outer stream is returned as the HMAC output tag.

1.4 HMAC-based IncMAC Variants

We present three implementations of HMAC whose API does not raise an error when used as IncMACs, i.e., it is possible to call update and finalize after an earlier finalize. The implementations differs form each other only in the finalize function.

We call the first the *Naive* implementation, as it is just the common simple code, which is also common in the reference implementations and RFCs. This implementation is used by Siemens as IncMAC in their S7 protocol (e.g., while downloading a control program to a PLC).[2] Listing 1.3 shows an example code of such a Naive HMAC implementation. In this implementation, the init function initializes a context record that includes two hash contexts that serve for the

[1] For simplicity, this implementation assumes that the inputs of the update function are always in multiples of full bytes.

[2] In their P3 protocol, e.g., between TIA V15 and PLC S7-1500 with firmware v1.8 [8].

inner stream and the outer stream. In this implementation, when finalize is called, two calls to the underlying hash finalize are made, both modify the content state. The first call operates on the inner stream. It pads the data of the inner stream and computes the hash value of the inner stream. Then, the HMAC updates the outer stream with the digest value of the inner stream. The second call to the hash finalize function operates on the outer stream. Similarly to the inner stream, it pads the data and computes the hash value. The digest value of the outer stream is returned as the HMAC output.

The second implementation protects the outer stream of HMAC, so we call it the *NIPO* (Naive Inner Patched Outer) implementation. The implementation of the OpenSSL library [1] is an example for a NIPO implementation. This implementation protects the outer stream by saving a local copy of the outer content, and works on the local copy during the finalize function. Therefore, each call to finalize uses the correct intermediate value (that follows the key) in the outer stream. This extra protection that seems to make the implementation closer to a real HMAC on the full message so far (and thus seems more secure than the Naive HMAC implementation) is actually much less secure. Listing 1.4 shows a code of the finalize function of the NIPO HMAC implementation.

A third implementation that protects both hash streams of HMAC is used by Python [2]. We call it the *PIPO* (Patched Inner Patched Outer) implementation. This implementation MACs the fragment prefix but not its division to fragments. Unlike the previous two, it computes standard HMAC on the prefixes. In this implementation, the inner and the outer streams are copied during finalize, and further processing is made on the local copy. Listing 1.5 shows a code of the finalize function of the PIPO HMAC implementation,

1.5 Related Work

The paper [12] shows that many security vulnerabilities were caused by software developers making mistakes, and argues that security professionals are responsible to create developer-friendly APIs. In addition, it focuses on the usability of cryptographic APIs, and proposes several principles for constructing usable and secure cryptographic APIs. The paper [11] demonstrates that SSL certificate validation is broken in many security applications and libraries due to badly designed APIs of SSL implementations.

Bellare, Goldreich, and Goldwasser defined the concept of incremental cryptography [6,7] as algorithms that allow to efficiently modify their outputs according to modifications in the input message. For example, given a message that is digitally signed. If this input message is slightly modified, an incremental algorithm provides an efficient way to get the new digital signature (without the full computation time of the digital signature of the modified message). Modification of a message includes replacing one block by another, insert a new block or delete an existing block. This definition is different than IncMACs because its purpose is to efficiently update the output (in time proportional to the change made in the underlying message). IncMACs' goal is to supply a secure and efficient algorithms to authenticate incremental prefixes of the same message.

An idea similar to IncMACs was proposed in [3]. It was noted that existing MAC primitives does not allow incremental calls, and some of the difficulties in allowing this kind of usage were mentioned. However, no concrete solution was suggested for this problem. A concrete proposal, but to a related problem, was suggested by Gennaro [10]. While we are interested in incremental symmetric authentication of fragmented messages, he is interested in incremental (public-key) signatures, with the non-repudiation property. He calls the fragmented messages by the term streams. He suggests protocols that provide digital signatures on streams more efficiently than signing each fragment individually, while still being able to provide a usable signature for every received prefix, even if communication hangs at the middle of a message.

1.6 Our Results

We concentrate on three implementations of HMAC-based IncMAC variants. The underlying issue behind all our analysis is that the security proof of HMAC does not hold once HMAC is used incrementally, as it's assumptions hold only on the first fragment. Once a second fragment is processed, the assumptions become invalid, and cannot be cured. We show that the Naive implementation as used as IncMAC by Siemens in their S7 protocol is vulnerable to collision, key recovery, message extension and forgery attacks whose complexity is significantly lower than prior attacks and lower than expected from such primitives. The NIPO implementation (the implementation of the OpenSSL library [1]), which seems to be better protected, is actually even less secure. Collisions can easily be found in a negligible time, which collide under all keys. It is also vulnerable to forgery attacks with negligible time and data complexities.

Table 1 summarizes our results compared to previously published attacks. For each attack, the data complexity is the number of required messages, where each message may consist of a small number of fragments (up to three in our case). The suffixes stand for the type of the data, where KM stands for known messages, CM for chosen messages and ACM for adaptive chosen messages. Notice that in some cases zero messages are required. These attacks are key independent and the same pairs of messages are collisions under any key. The time complexity is the number of calls that the attacker calls to the compression function plus the number of table lookups that the attacker performs to insert or fetch messages.

1.7 Notations

Throughout this paper we use the notations listed in Table 2.

1.8 Structure of the Paper

This paper is organized as follows: Section 2 presents key-independent collision attacks against the NIPO HMAC implementation, and Section 3 describes

Attack	Primitive	Source or Section	Complexity* Data	Compressions	Lookups	Success Prob.
Collision	HMAC (non-inc.)	-	$2^{n/2}$ KM	-	$2^{n/2}$	50%
(Single	NIPO HMAC	2.2	**0**	**0**	**0**	100%
Key)	Ideal MAC	-	$2^{n/2}$ KM	-	$2^{n/2}$	50%
Collision	HMAC (non-inc.)	-	No such attack algorithm			
(Key	NIPO HMAC	2.2	**0**	**0**	**0**	100%
Independent)	Ideal MAC	-	$2^{n \cdot 2^{\lvert K \rvert}}$	-	$2^{n \cdot 2^{\lvert K \rvert}}$	100%
Multi-Collision	HMAC (non-inc.)	-	$2^{n/2} \log c$ ACM	-	$2^{n/2} \log c$	50%
(Single	NIPO HMAC	2.3, 2.4	**0**	**0**	**0**	100%
Key)	Ideal MAC	-	$(2^n)^{c-1/c}$ KM	-	$(2^n)^{c-1/c}$	50%
Multi-Collision	HMAC (non-inc.)	-	No such attack algorithm			
(Key	NIPO HMAC	2.3, 2.4	**0**	**0**	**0**	100%
Independent)	Ideal MAC	-	No such attack algorithm			
Key-	HMAC (non-inc.)	Ex. Sea.	1 KM	$\min(4 \cdot 2^{\lvert K \rvert}, 2 \cdot 2^{2n})$	-	100%
Recovery	HMAC (non-inc.)	[5]	$2^{n/2}$ KM	$2 \cdot 2^n$	$2^{n/2}$	50%
	Naive HMAC	4.1	1 KM	$4 \cdot 2^n$	-	100%
	Ideal MAC	-	$\lceil \lvert K \rvert / n \rceil$	$2^{\lvert K \rvert}$	-	100%
Forgery	HMAC (non-inc.)	[5]	$2^{n/2}$ ACM	-	$2^{n/2}$	50%
	HMAC (non-inc.)	[15]	$2^{n/2}$ ACM	-	$2^{n/2}$	50%
	Naive HMAC	4.4	$2^{n/2}$ CM	$2^{n/2}$	$2^{n/2}$	50%
	NIPO HMAC	3.1	**2 CM**	**0**	**0**	100%
	NIPO HMAC	3.2	$2^{n/2}$ CM	-	$2^{n/2}$	50%
	NIPO HMAC	3.3	$2^{n/2}$ KM	-	$2^{n/2}$	50%
	Ideal MAC	-	$\lceil \lvert K \rvert / n \rceil$	$2^{\lvert K \rvert}$	-	100%
Message	Naive HMAC	4.2	1 KM	2^n	-	100%
Extension	Naive HMAC	4.3	$2^{n/2}$ KM	$2^{n/2}$	$2^{n/2}$	50%
	Ideal MAC	-	$\lceil \lvert K \rvert / n \rceil$	$2^{\lvert K \rvert}$	-	100%

* '-' marks 0 complexity (compressions or table lookups).

Table 1. Complexity of Attacks Against HMAC and HMAC-based IncMACs

Notation	Description
b	is the length of a message block in bits (typically 512 or 1024).
n	is the length of the output of the hash function or MAC in bits.
d	is the number of padding bits assigned for length information (typically 64 or 128 bits).
$\lvert x \rvert$	is the length of the bit string x.
$x \lvert\lvert y$	is the concatenation of two bit strings x and y.
0^s	is the concatenation of s zeros.
K	is a b-bit MAC secret key. Shorter keys are padded to b bits by 0's ($K \leftarrow K \lvert\lvert 0^{b-\lvert K \rvert}$).
$H(x)$	is the application of a hash function H on a bit string x.
$H^*(h, x, l)$	is the application of a hash function H on a bit string x, with the initial value being h (instead of the standard IV), and with the length in the padding being l (instead of $\lvert x \rvert$).
$H^*(h, x, \cdot)$	is $H^*(h, x, l)$ when the length l is easily extractable from the context.
$pad(x, l)$	is the string $1 \lvert\lvert 0^c \lvert\lvert l$, where $c = b - 1 - (\lvert x \rvert + 1 + d - 1 \mod b)$, and where l is represented as a d-bit big-endian integer.
v_i	The inner intermediate chaining value after the ith fragment.
u_i	The outer intermediate chaining value after the ith fragment.

Table 2. Notations

forgery attacks against this implementation. Section 4 discusses the most common implementation of HMAC — the Naive HMAC implementation, while Section 5 does the same for the PIPO HMAC implementation. Section 6 presents Secure IncMAC constructions and discusses their security. Section 7 summarizes this paper.

2 Key-Independent Collision Attacks on the OpenSSL Implementation (NIPO)

The implementation of HMAC in OpenSSL follows the NIPO implementation. The NIPO implementation protects the outer stream by working on a local copy of the content during the finalize function. Therefore, the outer stream always contains only a single tag value from the inner stream, which ensures that any collision in any chaining value of the inner stream results in a collision in the outer stream at the same location, i.e., a collision of the final authentication tag. In this section we present our basic techniques which shows how mixed calls to update / finalize affect the inner stream of NIPO and lead to collisions. Such collisions are then also collisions in the outer stream.

In the NIPO HMAC implementation, when we ask for the authentication tags of a message M fragmented to t fragments $M = (M_1, ..., M_t)$, we receive the incremental outputs that are used to authenticate the prefixes of M which are calculated as follows:

$$
\begin{aligned}
&v_1 = H(K \oplus ipad || M_1) \\
&v_i = H^*(v_{i-1}, M_i, l_i^{in}), \qquad i \geq 2 \text{ where } l_i^{in} = |K| + \textstyle\sum_{j=1}^{i} |M_j| \\
&u_i = H(K \oplus opad || v_i), \qquad i \geq 1 \\
&\text{IncMAC}_K(M_1, .., M_i) = u_i.
\end{aligned}
$$

In this implementation the context of the outer stream remains the same in any call to finalize. Figure 1 demonstrates the inner and outer streams in this implementation. The broken line represent this fixed context. Notice that the outer stream always contains only a single tag value from the inner stream after the key.

2.1 The Padded Inner Stream of NIPO

Consider a long message broken into a number of fragments at the locations of finalize calls. I.e., the first fragment of the message is the concatenated inputs to the calls to update before the first finalize. Any other fragment is the concatenated inputs to all update calls between two consecutive calls to finalize.

As a simple example, lets consider a message consisting of two fragments that are hashed in the following sequence. After init, the first fragment serves as the input to the first update, after which a finalize is called. The digest of the inner stream after this call to finalize is the correct keyed hash value of the first fragment, since this sequence is a legal sequence of calling the API. Consider

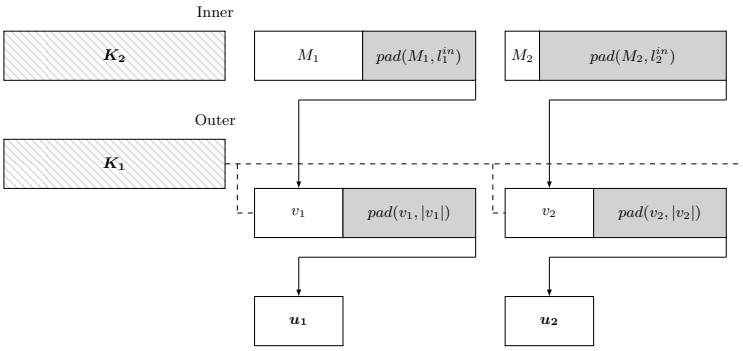

Fig. 1. The Inner and Outer Streams in The HMAC NIPO Implementation

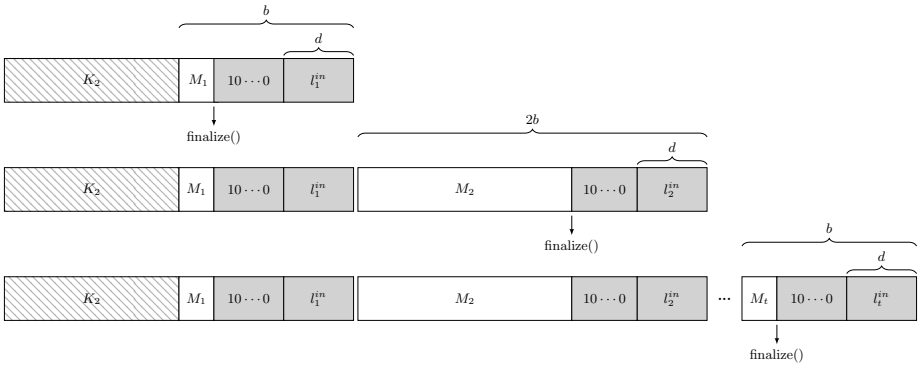

Fig. 2. The Padded Inner Stream of the HMAC NIPO Implementation

now that another update is called on the context with a second fragment, after which another finalize is called. Since the first finalize modified the context of the inner stream, padding is added in the padded stream between the two message fragments, where this padding also affect the final result. We define the padded stream of the inner stream of NIPO to be $K \oplus ipad$ (denoted by K_2) followed by the message fragments with the padding added by calling finalize. I.e, the padded stream is K_2 followed by the message with a padding after each of the message fragments. Figure 2 shows this padded inner stream that is passed as inputs to the compression function, and how the padded inner stream evolves when new fragments are added. Each row consists of K_2 followed by the first fragment on the left side, then padding, then the next fragment and its padding, and so on, repeatedly till the end of the message. The key is marked with stripes, the parts with white background mark the fragments of the message, and parts with gray background mark the padding added by the calls to finalize.

Notice that in such fragmented messages:

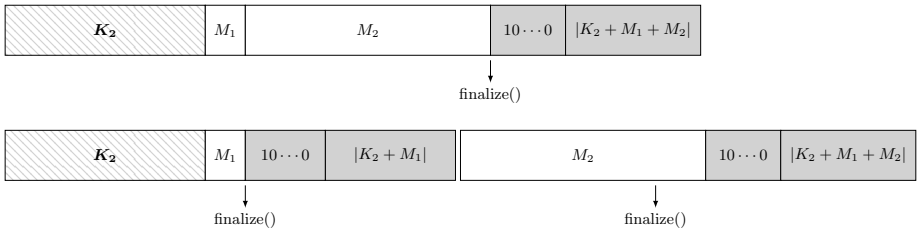

Fig. 3. Two Different Fragmentations of a Message Which Result with Different Authentication Tags in the NIPO HMAC Implementation

1. The padded inner stream contains a padding after every fragment. As a result, unlike in pure HMAC, the digest is calculated on a stream that mixes the message and many intermediate paddings rather than only a message and a single padding at the end.
2. Each padding contains the length of the key and message so far (i.e., the sum of the lengths of the key and the fragments, not including the lengths of the paddings). As a result, this length is not the actual length of the stream. Even worse, the number of blocks of the padded stream is not a function of the length any more.

These properties are not welcome. But they are not security threats on their own. The question posed is whether we can use these properties for creating valid attacks. Or in other words, can we find collisions between digests reached from calling finalize with different message fragments? Or maybe we can even devise some other attacks that give unexpected properties that MAC functions should not have? We answer these questions in the next subsections.

2.1.1 Every Message Has Many Possible Authentication Tags We may expect that each message should have a unique authentication tag, that may serve to uniquely identify the content. However, as we show here, the tag depends on the fragmentation: Every message can be fragmented after every bit. Therefore, a message of length n, has $n-1$ potential fragmentation points, with 2^{n-1} possible fragmentations. Each of these fragmentations results with a different padded inner stream, and thus a different (last) authentication tag.

As a simple example, lets consider a message $M = M_1||M_2$. The message M, when not fragmented, is padded with a single '1' bit, followed by (zero or more) zeros and the length of the data (including the key) at the end of the message to create its padded stream. When it is fragmented to two fragments M_1 and M_2, there are two padding, one after M_1 and another after M_2. Figure 3 demonstrates these fragmentations and their padding (the parts with gray background marks the end of fragments). Clearly, the final digests of the two cases are different.

2.1.2 The Merkle-Damgård Proof Does Not Hold

The Merkle-Damgård construction is proven to be collision-resistant if underlying compression function is collision-resistant. This proof[3] serves as the cornerstone of the trust in this construction. We show here that this proof is invalid in the IncMAC case.

The security proof of the Merkle-Damgård construction relies on the following two assumptions that do not hold in our case (M_{pad} is the padded stream of a message M):

1. M is a prefix of M_{pad}.
2. if $|M^1| = |M^2|$ then $|M^1_{pad}| = |M^2_{pad}|$.

In our case, Assumption 1 does not hold because fragmenting a message M results in a padding being added to the padded stream after each fragment. Therefore, the message is not a prefix of the padded stream.

In addition, Assumption 2 does not hold: Same message lengths (as written at the end of the padding) do not ensure that the padded messages have the same lengths. The Merkle-Damgård proof uses the length to ensure that padded messages with different numbers of blocks have different last blocks. This property does not hold in our case because the same message can be fragmented differently with different intermediate paddings, and thus different block-content and different number of total blocks, while still leaving the final padding (in the last block) unchanged.

2.2 Collisions (Key-Independent)

For a collision attack we find two different messages that result in the same padded inner stream under the same key. We actually describe an even stronger attack that finds collisions that collide under any key, and with most underlying hash primitives (conditioned on having the same block length and same padding). The trick of our collision attack is to include the intermediate padding of the first fragment of one message as a part of a fragment of the other message, and vice versa, thus ensuring that both padded streams are equal.

For example, let x, y be two bit-strings of the same length s (modulo b), and let z be some arbitrary string of length r. The following two messages lead to the same padded stream:

1. M^1 is a message fragmented to two fragments: the first is x, and the second is $y||pad(y, |K| + |x| + |y|)||z$.
2. M^2 is also a message fragmented to two fragments: the first is $x||pad(x, |K| + |x|)||y$, and the second is z.

Figure 4 illustrates this collision. The key is marked with stripes. Parts with white background mark fragments of the message itself, and parts with gray background mark the paddings added by the calls to finalize. The messages are different, and the paddings of the first fragment of each message are included as

[3] An early version of this proof is given in [16].

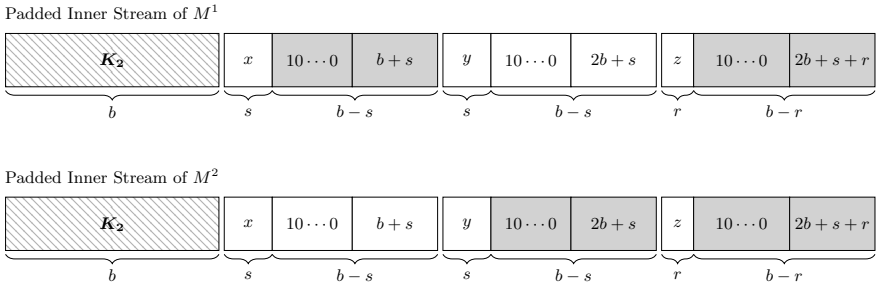

Fig. 4. The Padded Inner Stream of Colliding Messages in the NIPO Implementation

part of the other message. As can easily be seen, these messages result in the same padded inner stream for any key K_2 and x, y, and z for which $|x| = |y|$. Thus, the outputs of $H(K_2||\ldots)$ on these fragmented messages collide. Because the outer stream processes only the colliding value from the inner stream (i.e., only $H(K_2||\ldots)$), any collision in the inner stream results in a collision in the outer stream. Therefore, the authentication tag of these fragmented messages also collide. It can be viewed as shifting of the location of the padding in the padded stream between two possible locations.

Notice that there is no need for the full message to be known for this technique to work. In particular, there is no need to know the key K in order to calculate $pad(y, |K|+|x|+|y|)$ and $pad(x, |K|+|x|)$, since only the key length matters. The complexity of finding such collisions is negligible, and they are key independent.

We can extend this example to longer colliding messages with more fragments. In a simple case, we append the same fragments to the end of both messages to get new colliding messages.

2.3 Linear Multi-Collisions (Key-Independent)

We can extend this technique to create multi-collisions in several ways. We call the first of those *linear multi-collisions*. It shifts the location of the padding in the padded stream between more than two locations. Technically, the message contains several locations in the message that fit to become a valid padding of the message prefix till that location.

2.4 Exponential Multi-Collisions (Key-Independent)

Linear multi-collisions are not the most efficient to make k-collisions for large k's. Instead of preparing k locations for setting the paddings and breaking the message to fragments at a single such location, it is possible to make the choice many times in a single message. Moreover, the choice can be made independently for any pair of locations. Therefore, the number of choices become exponential with the length of the message.

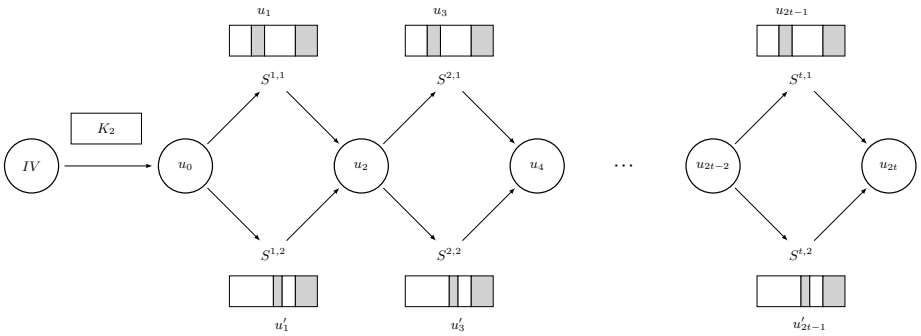

Fig. 5. Exponential Multi-Collisions

Fragments 1–2	Fragments 3–4	Fragments 5–6
$1 - 110^{446}0^{52}0100000000011$	$1 - 110^{446}0^{52}0110000000111$	$1 - 110^{446}0^{52}1000000001011$
$1 - 110^{446}0^{52}0100000000011$	$1 - 110^{446}0^{52}0110000000111$	$110^{446}0^{52}0110000001011 - 1$
$1 - 110^{446}0^{52}0100000000011$	$110^{446}0^{52}0100000000111 - 1$	$1 - 110^{446}0^{52}1000000001011$
$1 - 110^{446}0^{52}0100000000011$	$110^{446}0^{52}0100000000111 - 1$	$110^{446}0^{52}0110000001011 - 1$
$110^{446}0^{52}0010000000011 - 1$	$1 - 110^{446}0^{52}0110000000111$	$1 - 110^{446}0^{52}1000000001011$
$110^{446}0^{52}0010000000011 - 1$	$1 - 110^{446}0^{52}0110000000111$	$110^{446}0^{52}0110000001011 - 1$
$110^{446}0^{52}0010000000011 - 1$	$110^{446}0^{52}0100000000111 - 1$	$1 - 110^{446}0^{52}1000000001011$
$110^{446}0^{52}0010000000011 - 1$	$110^{446}0^{52}0100000000111 - 1$	$110^{446}0^{52}0110000001011 - 1$

Each row contains one message, each message has six fragments. Content is in binary.

Table 3. Key-Independent 8-Collisions of Inc-HMAC-SHA-1 and Inc-HMAC-SHA-256

Exponential multi-collisions are formed in a similar way to the multi-collisions of [13]. Given t pairs of fragmented sequences $(S^{i,1}, S^{i,2})$ that lead to a collision with any initial value (as in Subsection 2.2), we create 2^t different messages with the same digest by concatenating any combination of t sequences, where the ith sequence is arbitrarily chosen from $S^{i,1}$ or $S^{i,2}$. As $S^{1,1}$ and $S^{1,2}$ collide, then the chaining values after this sequence are equal. Therefore, also $S^{2,1}$ and $S^{2,2}$ that follow them collide and so on, leading to multi-collision of all the 2^t messages. Figure 5 illustrates exponential multi-collisions. As in the case of collisions, since the length is included in the paddings of the fragments of the sequences, then the length is also included in the partner fragments as part of the message.

With this construction, it is possible to create 2^t-collisions with $2t$-block messages. These multi-collisions are key-independent, and hold for any key. Moreover, the attacker has no need to know the key, or ask the victim to authenticate with the unknown key, in order to find the multi-collisions. All he needs, for example, in case of 8-collision, is to copy the key-independent multi-collisions from Table 3. We stress that these collisions hold to all MD-based Inc-HMAC with 512-bit blocks, as the details of the compression functions are not affecting the collisions.

3 Forgery Attacks Against NIPO HMAC Implementation

In this section we describe attacks against the NIPO and HMAC implementation.

3.1 A Simple Forgery Attack

A forgery attack can easily be performed given a pair of colliding messages, e.g., those mentioned in Subsection 2.2. Given such a collision, the attacker asks for the authentication tags of the first member of this message pair. Since the second tags of these messages are equal, the attacker only needs to request the tag of the first fragment to forge the authentication tags of the second message. The data and the time complexity of this attack are negligible (computing tags of two messages with a total of three fragments). The full details of the attack are as follows:

1. Choose two bit strings x and y of the same length (modulo b), and any bit string z.
2. Ask for the authentication tags of the message $M = (M_1, M_2)$ fragmented to two fragments, where $M_1 = x$ and $M_2 = y||pad(y, |K| + |x| + |y|)||z$. Let the authentication tags be u_1, u_2.
3. Ask for the authentication tag of the single-fragment message
$$M'_1 = x||pad(x, |K| + |x|)||y.$$
Let the authentication tag be u'_1.
4. Let M' be a two-fragment message $M' = (M'_1, M'_2)$ where $M'_2 = z$. The authentication tags of M' are the already known u'_1 and $u'_2 = u_2$.

3.2 Chosen Plaintext Forgery Attack

This attack is similar to the forgery attack against HMAC, and based on finding collisions in the underlying hash function. In this attack, the attacker forges authentication tags of massages that had not been authenticated. The data and the time complexity of this attack are $2^{n/2}$. The full details of the attack are as follows:

1. Ask for the authentication tags of $2^{n/2}$ messages M^i fragmented to three fragments $M^i = (M^i_1, M_2, M^i_3)$ each, where the first fragment is of fixed length, and the second fragment is identical in all messages. Let the three authentication tags of a message M^i be u^i_1, u^i_2, u^i_3
2. If there exist i and j such that $u^i_1 = u^j_1$ and $u^i_2 = u^j_2$, conclude that $v^i_1 = v^j_1$.
3. The authentication tags of the three-fragment message $M' = (M'_1, M'_2, M'_3)$, where $M'_1 = M^i_1$, $M'_2 = M_2$ and $M'_3 = M^j_3$ are u^j_1, u^j_2, u^j_3.

3.3 Known Plaintext Forgery Attack

The attack of Subsection 3.2 is a chosen plaintext attack because it validates that the collision occurs in the inner stream. This attack can be converted into a known plaintext attack, without this validation at the cost of a reduced success rate. The data and the time complexity of this attack are $2^{n/2}$.

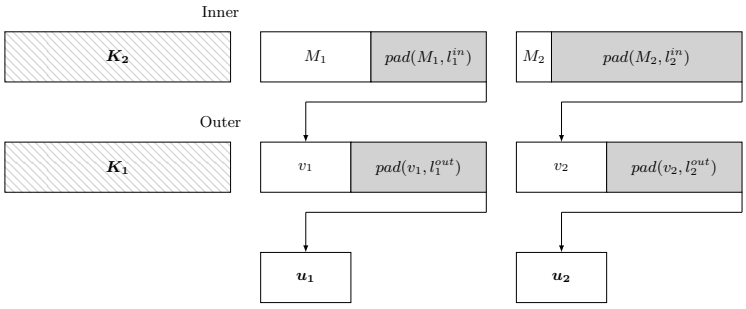

Fig. 6. The Inner and Outer Streams in the HMAC Naive Implementation

4 Cryptanalysis of the Naive HMAC Implementation

In this section we discuss the most common HMAC implementation, to which we call the Naive HMAC implementation. This Naive implementation is the most common in libraries and applications, since it is based on the versions listed in the RFCs. This implementation is used by **Siemens** controllers and control software to incrementally authenticate fragmented messages. It optimizes the memory usage in finalize without any specific means to allow further calls to update afterwards. It uses the Naive hash implementation when it calls the hash function, without any trial to protect against incremental calls.

In the Naive HMAC implementation, the authentication tags of a message M fragmented to t fragments $M = (M_1, ..., M_t)$ are calculated as follows:

$$
\begin{aligned}
v_1 &= H(K \oplus ipad || M_1) \\
v_i &= H^*(v_{i-1}, M_i, l_i^{in}), \qquad i \geq 2 \text{ where } l_i^{in} = |K| + \textstyle\sum_{j=1}^{i} |M_j| \\
u_1 &= H(K \oplus opad || v_1) \\
u_i &= H^*(u_{i-1}, v_i, l_i^{out}), \qquad i \geq 2 \text{ where } l_i^{out} = |K| + \textstyle\sum_{j=1}^{i} |v_j| \\
&\text{IncMAC}_K(M_1, .., M_i) = u_i.
\end{aligned}
$$

Recall that v_i and u_i are the chaining values of the inner and outer streams after the i'th fragment, respectively. The weaknesses described in Section 2.1 cause the inner stream and the outer stream to overwrite their states during every call to finalize. Figure 6 shows the padded streams of a Naive HMAC implementation. The first row shows the padded inner stream, and the second shows the padded outer stream. The arrows represent the chaining values and they point to where they are used. Notice that the outer padded stream contains many fragments, one for each message fragment.

In the following subsections, we present various attacks against the Naive HMAC implementation when used with multiple fragments. Unlike in the case of NIPO, in Naive implementations a collision in the inner stream does not force a collision in the outer stream, because the intermediate digests of the inner stream are mixed as fragments in the outer stream. So more complex algorithms are required to attack the Naive implementation.

4.1 Key Recovery Attack

There are two known key recovery attacks against (standard) HMAC. Exhaustive search allows the attacker to find the key itself,[4] given a single authentication tag of a message, in $\min(4 \cdot 2^{|K|}, 2 \cdot 2^{2n})$ time complexity. The other attack [5] is based on finding collisions in the underlying hash function. In this attack, the attacker finds the intermediate values after compressing $K \oplus ipad$ and $K \oplus opad$ denoted by c^{in} and c^{out}, whose knowledge is equivalent to recovering the key. The data complexity of this attack is $2^{n/2}$ MACed messages, and the time complexity is 2^n.

In our case, where each message is fragmented and each fragment has a tag, we can improve over this attack. When the attacker asks for the authentication tags of a message M fragmented to t fragments $M = (M_1, ..., M_t)$, he receives the incremental outputs computed on the prefixes of M. In the Naive HMAC implementation the inner intermediate chaining values v_i are calculated as follows:

$$v_1 = H(K \oplus ipad || M_1)$$
$$v_i = H^*(v_{i-1}, M_i, l_i^{in}),\ i \geq 2 \ \text{ where } l_i^{in} = |K| + \textstyle\sum_{j=1}^{i} |M_j|.$$

Based on the inner intermediate chaining values, the outer intermediate chaining values u_i, that serve as the incremental outputs tags, are calculated as follows:

$$u_1 = H(K \oplus opad || v_1)$$
$$u_i = H^*(u_{i-1}, v_i, l_i^{out}),\ i \geq 2 \ \text{ where } l_i^{out} = |K| + \textstyle\sum_{j=1}^{i} |v_j|.$$

These outer intermediate chaining values u_i ($i \geq 1$) are assumed to be known to the attacker once the message has been authenticated. Clearly, this knowledge may be very helpful to the attacker.

In this attack, the attacker asks for the authentication tags of a single message, fragmented to three fragments, and finds c^{in} and c^{out}, the intermediate values after compressing $K \oplus ipad$ and $K \oplus opad$. These intermediate values are equivalent to the key for the purpose of forging messages. The data complexity of this attack is negligible and the time complexity is 2^n. The full details of the attack are as follows:

1. Ask for the three authentication tags of a message M fragmented to three fragments $M = (M_1, M_2, M_3)$. Let u_1, u_2, u_3 be the tags.
2. Find v_2: For every n-bit string $\alpha \in \{0,1\}^n$:
   (a) Calculate the value of $H^*(u_1, \alpha, \cdot)$.
   (b) If $H^*(u_1, \alpha, \cdot) = u_2$:
      i. Calculate the value of $H^*(u_2, H^*(\alpha, M_3, \cdot), \cdot)$.
      ii. If $H^*(u_2, H^*(\alpha, M_3, \cdot), \cdot) = u_3$, conclude that $v_2 = \alpha$.
3. Find c^{in}: For every n-bit string $\alpha \in \{0,1\}^n$:
   (a) Calculate the value of $H^*(H^*(\alpha, M_1, \cdot), M_2, \cdot)$.
   (b) If $H^*(H^*(\alpha, M_1, \cdot), M_2, \cdot) = v_2$, conclude that $c^{in} = \alpha$.
4. Find c^{out}: For every n-bit string $\alpha \in \{0,1\}^n$:
   (a) Calculate the value of $H^*(\alpha, H^*(c^{in}, M_1, \cdot), \cdot)$.
   (b) If $H^*(\alpha, H^*(c^{in}, M_1, \cdot), \cdot) = u_1$, conclude that $c^{out} = \alpha$.

[4] Or an equivalent key with the same chaining value.

4.2 A Simple Message Extension Attack

If the attacker had known some intermediate chaining variable v_i ($i \geq 2$) in the inner stream of a message $M = (M_1, ..., M_i)$, he could have easily forged an authentication tag of the message extended by any single fragment M'_{i+1} to $M' = (M_1, ..., M_i, M'_{i+1})$, by calculating the values:

$$v'_{i+1} = H^*(v_i, M'_{i+1}, l^{in'}_{i+1}) \text{ where } l^{in'}_{i+1} = l^{in}_i + |M'_{i+1}|$$
$$u'_{i+1} = H^*(u_i, v'_{i+1}, l^{out'}_{i+1}) \text{ where } l^{out'}_{i+1} = l^{out}_i + |v'_{i+1}|.$$

Then, the attacker could extend this prefix by more fragments. It is therefore that only the knowledge of v_i stands between the attacker and his ability to forge messages. Let's show how the attacker can perform this missing step.

In this attack, the attacker asks for the authentication tags of a single message, fragmented to three fragments, and finds the second intermediate chaining value v_2 in the inner stream. The data complexity of this attack is negligible and the time complexity of is 2^n. The full details of the attack are as follows:

1. Ask for the three authentication tags of a message M fragmented to three fragments $M = (M_1, M_2, M_3)$. Let u_1, u_2, u_3 be the tags.
2. Find v_2: For every n-bit string $\alpha \in \{0,1\}^n$:
   (a) Calculate the value of $H^*(u_1, \alpha, \cdot)$.
   (b) If $H^*(u_1, \alpha, \cdot) = u_2$:
      i. Calculate the value of $H^*(u_2, H^*(\alpha, M_3, \cdot), \cdot)$.
      ii. If $H^*(u_2, H^*(\alpha, M_3, \cdot), \cdot) = u_3$, conclude that $v_2 = \alpha$ and that $v_3 = H^*(\alpha, M_3, \cdot)$
3. Let M' be a message fragmented to four fragments $M' = (M_1, M_2, M_3, M'_4)$ (where M'_4 can be any fragment). The authentication tags of M' are u_1, u_2, u_3, u'_4. The tag u'_4 of M'_4 can be forged by computing: $v'_4 = H^*(v_3, M'_4, \cdot)$ and $u'_4 = H^*(u_3, v'_4, \cdot)$.

4.3 A More Efficient Message Extension Attack

In this attack the attacker asks for authentication tags of $2^{n/2}$ messages fragmented to three fragments each, where the first fragment is identical in all messages, and the second fragment is different in all messages. The first intermediate chaining values in the outer stream u_1 of all of these messages are identical. The rest is then performed by applying the Birthday Paradox, which allows the attacker to find the second intermediate chaining value in the inner stream of some message. The data and the time complexity of this attack are $2^{n/2}$. The full details of the attack are as follows:

1. Ask for an authentication tags of $2^{n/2}$ messages M^i fragmented to three fragments each $M^i = (M_1, M^i_2, M^i_3)$, where the first fragment is identical in all messages, and the second fragment is different in all messages. Let the three authentication tags of a message M^i be u_1, u^i_2, u^i_3.
2. Do forever (in practice about $2^{n/2}$ times):

(a) Choose random n-bit string $\alpha \in \{0,1\}^n$.
(b) Calculate the value of $H^*(u_1, \alpha, \cdot)$.
(c) If there exists i such that $H^*(u_1, \alpha, \cdot) = u_2^i$:
   i. Calculate the value of $H^*(u_2^i, H^*(\alpha, M_3^i, \cdot), \cdot)$.
   ii. If $H^*(u_2^i, H^*(\alpha, M_3^i, \cdot), \cdot) = u_3^i$, conclude that $v_2^i = \alpha$ and that $v_3^i = H^*(\alpha, M_3^i, \cdot)$.
   iii. Go to Step 3.

3. Let M' be a message fragmented to four fragments $M' = (M_1, M_2^i, M_3^i, M_4')$ (where M_4' can be any fragment). The authentication tags of M' are u_1, u_2^i, u_3^i, u_4'. The tag u_4' of M_4' can be forged by computing: $v_4' = H^*(v_3^i, M_4', \cdot)$ and $u_4' = H^*(u_3^i, v_4', \cdot)$.

4.4 Forgery Attack

Two similar versions of adaptive chosen forgery attack against HMAC, which is based on finding collisions in the underlying hash function, were presented in [5,15]. In this attack, the attacker forges authentication tags of messages that had not been authenticated. The data and the time complexity of this attack are $2^{n/2}$.

We show another authentication forgery attack, which is based on the technique of Subsection 2.2. This technique creates pairs of messages fragmented to two fragments each, where every pair of messages has the same value of the second inner intermediate chaining value. In this attack the attacker creates $2^{n/2}$ such pairs, and asks for the authentication tags of the first member of each of these message-pairs, resulting with $2^{n/2}$ outputs. Similarly to the attack of Section 4.3, the attacker finds the second intermediate chaining value of the inner stream of some first member message. Recall that the second inner intermediate chaining value is identical in both members of the same pair. Now if the attacker knows the authentication tag of the first fragment of the second member, and thus he can forge an authentication tag of the entire message. The data and the time complexity of this attack are $2^{n/2}$. The full details of the attack are as follows:

1. Choose two bit strings x and y of the same length (modulo b).
2. For $1 \leq i \leq 2^{n/2}$:
   (a) Choose two bit strings z^i and M_3^i.
   (b) Create a message $M^i = (M_1^i, M_2^i, M_3^i)$ fragmented to three fragments, where $M_1^i = x$ and $M_2^i = y||pad(y, |K| + |x| + |y|)||z^i$.
3. Ask for the authentication tags of the messages M^i. Let the three authentication tags of the message M^i be u_1, u_2^i, u_3^i.
4. Ask for the authentication tag of the single-fragment message $x||pad(x, |K| + |x|)||y$. Let the authentication tag be u_1'.
5. Do forever (in practice about $2^{n/2}$ times):
   (a) Choose a random n-bit string $\alpha \in \{0,1\}^n$.
   (b) Calculate the value of $H^*(u_1, \alpha, \cdot)$.
   (c) If there exists i such that $H^*(u_1, \alpha, \cdot) = u_2^i$:

i. Calculate the value of $H^*(u_2^i, H^*(\alpha, M_3^i, \cdot), \cdot)$.
ii. If $H^*(u_2^i, H^*(\alpha, M_3^i, \cdot), \cdot) = u_3^i$, conclude that $v_2^i = \alpha$.

6. Let M' be a message fragmented to two fragments $M' = (M'_1, M'_2)$ where $M'_1 = x||pad(x, |K|+|x|)||y$ and $M'_2 = z^i$. The authentication tags of M' are the already known u'_1, and $u'_2 = H^*(u'_1, v_2^i, \cdot)$.

5 The PIPO HMAC Implementation

The PIPO (Patched Inner Patched Outer) HMAC implementation is identical to the naive and NIPO ones, except for the finalize function. In this implementation, the inner and the outer streams are copied during finalize, and further processing is made on the local copy. Listing 1.5 shows a code of the finalize function of the PIPO HMAC implementation.

In the PIPO HMAC implementation, when we ask for the authentication tags of a message M fragmented to t fragments $M = (M_1, ..., M_t)$, we receive the incremental outputs that are used to authenticate the prefixes of M. These tags are calculated as follows:

$$\begin{aligned} v_i &= H(K \oplus ipad||M_1||...||M_i) \\ u_i &= H(K \oplus opad||v_i) \\ IncMAC_K(M_1, .., M_i) &= u_i. \end{aligned}$$

This implementation is used in the Python programing language. In the Python implementation of hash functions, calling finalize does not add the padding to the inner stream — the state of the hash is copied and the padding is only added to the copy. In addition, in the finalize function of HMAC, the outer stream is copied. This is equivalent to copying and using the copies of both inner and outer streams. Therefore, the Python implementation of HMAC is equivalent to the PIPO HMAC implementation.

Note that in this implementation, mixed calls to update / finalize produce correct HMAC values for every prefix of the input. Therefore, the security of PIPO as an IncMAC seems equivalent to the security of HMAC on the same prefixes (the security of HMAC was proven in [4, 5]). Indeed any attack on HMAC is also an attack on PIPO, and any chosen plaintext (or adaptive) attack on PIPO is easily convertible to an attack on HMAC (by replacing the request for authentication tags for several fragments of a single message by requests for the HMAC tags for each fragment prefix). The complexity in this latter reduction may increase by a small factor (the maximal number of fragments in a message). However, some chosen plaintext attacks on HMAC may be converted to known plaintext attacks. This is caused by the fact that in PIPO a multiple-fragment message corresponds to several messages in the HMAC case, which have common prefixes. So there exist known plaintext attacks against PIPO that have no corresponding known plaintext attacks against HMAC.[5]

[5] Applications using the context copy method of HMAC implementations are mostly also vulnerable to such known plaintext attacks.

6 Secure IncMACs

Incremental MACs are of a special interest to those that wish to transmit a message that contains many fragments, and wish to be able to authenticate the message both at the packet level and at the application level. The network driver may authenticate each packet as a fragment, linking it directly to it's predecessor fragments, without the extra need for handling a session number and a packet counter typically used to protect against replay and other attacks. Each of these fragments has it's own digest, that also authenticates all the message prefix. At the same time, the application may receive only the last digest, which is useful for it to authenticate the full message, without the need to compute a second MAC instance on the same message, and send a redundant MAC value over the communication channel.

For example, transmission of a long file may be protected by an IncMAC, where the drivers at both side compute and check the digests of the fragments. The drivers then send only the last digest to the application, which can save it along the message on the disk, enabling to verify the file without an extra computation of another MAC instance. Alternatively, a database of many records (or tables) is transmitted, where each record is sent in a separate fragment. The drivers check authenticity of each fragment, while the application receives the fragments and the last digest. The application is able to verify by the single digest that the full message is correct. When using an FP-IncMAC, the application may even know that the division to records is correct, without needing to receive a separate digest to each record.

Notice that in the latter case it may also be desirable to authenticate the division to fragments, rather than just the full message. This additional feature is easy to support, as many IncMAC implementations do it any way, we call them FP-IncMACs. But for those that do not (e.g., PIPO) we propose how to add this feature.

6.1 PIPO and LAPIPO

As mentioned earlier, PIPO is a secure IncMAC. The only drawback is that some potential attacks might use the copying of the state during finalize to gain some factor in the complexity. This issue that also occurs when HMAC is used by the applications with a state copy operation. If an FP-IncMAC is wished for, based on a PIPO, we propose the LAPIPO FP-IncMAC construction. The extension technique is generic and can be used over any secure IncMAC. Also notice that it eliminates the potential issues mentioned for PIPO.

The LAPIPO (Length Added PIPO) construction uses the PIPO implementation with the addition of the fragment length after each fragment. This length is typically added by a call to the update function at the beginning of LAPIPO's finalize, rather than by modifying the PIPO API. It can be done by a direct call to update by the application, or by a dedicated module that provides an LAPIPO API and calls the PIPO API with this extra addition. Listing 1.6 contains an example code of LAPIPO. The difference from PIPO is that the fragment length is

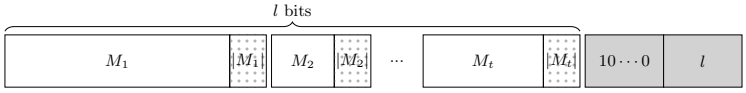

Fig. 7. The Padded Inner Stream of LAPIPO

added at the end of every fragment, and the length written in the PIPO paddings is the total length of the message prefix and the size of the added lengths (in order to allow to implement LAPIPO using a standard PIPO). We can consider it as a length-added (postfix free) message in which after each fragment the length of the fragment is added. This length-added message is then MACed by PIPO. Figure 7 illustrates the inner stream of LAPIPO (the values marked with dots are the fragments lengths added at the end of every fragment). The addition of the fragment length distinguishes between the same message with different fragmentations, while in the PIPO implementation, an authentication tag is independent of the message fragmentation.

7 Summary

This paper discusses unintended features of HMAC APIs. It shows that commonly used HMAC APIs allow calling the update function after the finalize function, which causes unintended features that expose these APIs to a number of vulnerabilities. Moreover, some applications use these unintended features for incremental authentication. It focuses on two implementations of HMAC: NIPO and Naive HMAC, and describe several attacks against them, such as key independent collision and multi-collision attacks, that are one of the strongest attacks possible, as the same collisions hold for any key, and even for other HMACs with other underlying hash functions. Other attacks include forgery, message extension and key recovery attacks.

An extreme example is the OpenSSL library, which implements HMAC using the NIPO implementation. Though the OpenSSL documentation states that hash functions should not be used incrementally, it does not state it regarding HMAC. It is unclear to us why OpenSSL uses the NIPO implementation, i.e., why the outer stream's state is copied in OpenSSL's implemenation to a local variable, while the inner stream's state is not. But this exact decision makes this implementation vulnerable to a number of highly efficient attacks, including key-independent collision and multi-collision attacks, with complexity 0, whose same collisions collide under any key and even when using other underlying hash function.[6]

The paper also discusses two other HMAC implementations, called PIPO and LAPIPO. The former is used as the implementation of Python. Both are secure.

[6] Note that this decision does not affect SSL/TLS, as far as we know, as the SSL/TLS protocol [19] does not use HMAC incrementally.

We recommend using the PIPO HMAC implementation when an IncMAC is required, or the LAPIPO construction when an FP-IncMAC is required. Their security is the best possible in the known message scenario, and is equivalent to the security of HMAC in all other scenarios.

We informed Siemens about their non-standard use of HMAC and of our attacks in 2019, and as far as we understand from their response, they made changes to their authentications in their latest product versions. In parallel, we also informed openSSL, who decided to wait with the decision whether to update. We hope that they will also update their software.

Acknowledgements

This research was partially supported by the Technion Hiroshi Fujiwara cyber security research center and the Israel national cyber directorate.

References

1. Openssl. `https://www.openssl.org`.
2. Python.org. `https://www.python.org`.
3. Rob Austein. [cryptech tech] incremental digest outputs. `https://lists.cryptech.is/archives/tech/2014-November/001008.html`, November 2014.
4. Mihir Bellare. New proofs for nmac and hmac: Security without collision resistance. *Journal of Cryptology*, 28(4):844–878, Oct 2015.
5. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 1–15, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
6. Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo G. Desmedt, editor, *Advances in Cryptology — CRYPTO '94*, pages 216–233, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
7. Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography and application to virus protection. In *Proc. of the 27th Ann. ACM Symp. on the Theory of Computing*, pages 45–56. ACM Press, 1995.
8. Eli Biham, Sara Bitan, Aviad Carmel, Alon Dankner, Uriel Malin, and Avishai Wool. Rogue7: Rogue engineering-station attacks on s7 simatic plcs. In *Blackhat USA 2019*, 2019.
9. Ivan Bjerre Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 416–427, New York, NY, 1990. Springer New York.
10. Rosario Gennaro and Pankaj Rohatgi. How to sign digital streams. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 180–197, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
11. Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 38–49, 2012.

```
initialize{
    initialize empty b-bit buffer;
    streamSize = 0;
    intermediateValue = IV;
}

update(data){
  prepend the content stored in the buffer to the data and
    divide it into b-bit blocks;
  if the last block is not complete, store it in the buffer;
  for each block B:
    intermediateValue = compress(intermediateValue, B);
  streamSize += dataSize;
}

finalize{
  append to the content stored in the buffer a single '1' bit;
  append c '0' bits; \\ c is the smallest non negative number
  such that bufferSize + 1 + c + d is a multiple of b;
  append streamSize as a d-bit big-endian integer;
  divide it into b-bit blocks;
  for each block B:
    intermediateValue = compress(intermediateValue, B);
  return intermediateValue;
}
```

Listing 1.1. Pseudocode of a Naive Merkle-Damgård Hash Function Implementation

12. Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.
13. Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, pages 306–316, 2004.
14. S. Kent. Rfc 4301 - security architecture for the internet protocol. `https://tools.ietf.org/html/rfc4301`, December 2005.
15. Jongsung Kim, Alex Biryukov, Bart Preneel, and Seokhie Hong. On the security of HMAC and NMAC based on haval, md4, md5, SHA-0 and SHA-1 (extended abstract). In *Security and Cryptography for Networks, 5th International Conference, SCN 2006, Maiori, Italy, September 6-8, 2006, Proceedings*, pages 242–256, 2006.
16. Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems*. UMI Research Press, 1979.
17. National Bureau of Standards and Technologies. Secure Hash Standard. *Federal Information Processing Standards, Publication FIPS-180-1*, 1995.
18. National Bureau of Standards and Technologies. Secure Hash Standard. *Federal Information Processing Standards, Publication FIPS-180-4*, 2001.
19. E. Rescorla. Rfc 8446 - the transport layer security (tls) protocol version 1.3. `https://tools.ietf.org/html/rfc8446`, August 2018.
20. R. Rivest. Rfc 3120 - the md4 message-digest algorithm. `https://tools.ietf.org/html/rfc1320`, April 1992.
21. Ronald L. Rivest. The MD5 message-digest algorithm. *RFC*, 1321:1–21, 1992.

```
ctx:
     buffer[b/8]
     bufferSize: int
     streamSize: int
     intermediateValue[n/8]
initialize{
     buffer = {};
     bufferSize = streamSize = 0;
     intermediateValue = IV;
}
update(data){
  index = 0;
  while(index < data.length){
    if(bufferSize + data.length-index < b/8){
      buffer.append(data[index,...,data.length-1]);
      bufferSize += data.length-index;
      index = data.length;
    } else {
      buffer.append(data[index,...,b/8-bufferSize-1]);
      intermediateValue = compress(intermediateValue, buffer);
      index += b/8-bufferSize;
      buffer = {};
      bufferSize = 0;
    }
  }
  streamSize += data.length*8;
}
finalize{
  buffer.append(0x80);
  if(bufferSize+1 > (b-d)/8){
    for(int i=0; i < b/8-bufferSize-1; i++){
      buffer.append(0);
    }
    intermediateValue = compress(intermediateValue, buffer);
    buffer = {};
    bufferSize = 0;
  }
  for(int i=0; i < (b-d)/8-bufferSize-1; i++) {
    buffer.append(0);
  }
  length = toBigEndian(streamSize, d);
  buffer.append(length);
  intermediateValue = compress(intermediateValue, buffer);
  buffer = {};
  bufferSize = 0;
  return intermediateValue;
}
```

Listing 1.2. Naive Merkle-Damgård Hash Function Implementation

```
ctx:
     innerStream: ctx of hash
     outerStrean: ctx of hash

initialize(key){
     while(key.len < b/8){
         key.append(0);
     }
     innerStream.initialize();
     outerStream.initialize();
     innerStream.update(key⊕ipad);
     outerStream.update(key⊕opad);
}

update(data){
     innerStream.update(data);
}

finalize{
     inner = innerStream.finalize();
     outerStream.update(inner);
     outer = outerStream.finalize();
     return outer;
}
```

Listing 1.3. Naive HMAC Implementation

```
finalize{
    outerStreamCopy = outerStream.copy();

    inner = innerStream.finalize();
    outerStreamCopy.update(inner);
    outer = outerStreamCopy.finalize();
    return outer;
}
```

Listing 1.4. The Finalize Function of the NIPO HMAC Implementation

```
finalize{
    innerStreamCopy = innerStream.copy();
    outerStreamCopy = outerStream.copy();

    inner = innerStreamCopy.finalize();
    outerStreamCopy.update(inner);
    outer = outerStreamCopy.finalize();
    return outer;
}
```

Listing 1.5. The Finalize Function of the PIPO HMAC Implementation

```
ctx:
    PIPO_ctx: ctx of PIPO
    previousLength: int

initialize(key){
    PIPO_ctx.initialize(key);
    previousLength = 0;
}

update(data){
    PIPO_ctx.update(data);
}

finalize(){
    fragmentLength = PIPO_ctx.innerStream.streamSize -
                     previousLength;
    PIPO_ctx.update(toBigEndian(fragmentLength, d));
    previousLength = PIPO_ctx.innerStream.streamSize;
    PIPO_ctx.finalize();
}
```

Listing 1.6. LAPIPO Implementation