Symbolic and Computational Mechanized Verification of the ARINC823 Avionic Protocols

Bruno Blanchet

Symbolic and Computational Mechanized Verification of the ARINC823 Avionic Protocols

Bruno Blanchet

RESEARCH REPORT
N° 9072
May 2017
Project-Team Prosecco

ISSN 0249-6399   ISRN INRIA/RR--9072--FR+ENG

Symbolic and Computational Mechanized Verification of the ARINC823 Avionic Protocols

Bruno Blanchet

Project-Team Prosecco

Research Report n° 9072 — May 2017 — 40 pages

Abstract: We present the first formal analysis of two avionic protocols that aim to secure air-ground communications, the ARINC823 public-key and shared-key protocols. We verify these protocols both in the symbolic model of cryptography, using ProVerif, and in the computational model, using CryptoVerif. While we confirm many security properties of these protocols, we also find several weaknesses, attacks, and imprecisions in the standard. We propose fixes for these problems. This case study required the specification of new cryptographic primitives in CryptoVerif. It also illustrates the complementarity between symbolic and computational verification.

Key-words: security protocols, verification, symbolic model, computational model

RESEARCH CENTRE
PARIS

2 rue Simone Iff
CS 42112
75589 Paris Cedex 12

Vérification mécanisée, symbolique et calculatoire, des protocoles avioniques ARINC823

Résumé : Nous présentons la première analyse formelle de deux protocoles avioniques qui visent à sécuriser les communications air-sol, les protocoles ARINC823 à clé publique et à clé partagée. Nous vérifions ces protocoles, à la fois dans le modèle symbolique de la cryptographie, en utilisant ProVerif, et dans le modèle calculatoire, en utilisant CryptoVerif. Si nous confirmons beaucoup de propriétés de sécurité de ces protocoles, nous trouvons aussi plusieurs faiblesses, attaques, et imprécisions dans le standard. Nous proposons des corrections pour ces problèmes. Cette étude de cas a nécessité la spécification de nouvelles primitives cryptographiques dans CryptoVerif. Elle illustre aussi la complémentarité entre la vérification symbolique et la vérification calculatoire.

Mots-clés : protocoles cryptographiques, vérification, modèle symbolique, modèle calculatoire

1 Introduction

Securing electronic communications between aircrafts and ground entities (control towers, airlines) becomes more and more important. Indeed, such electronic communications convey more and more important information and aircrafts are prominent targets for attacks, such as terrorist attacks. So, even if most air-ground communications are currently sent in the clear, the adoption of secured communications seems unavoidable.

Several standards have been developed [2,34,35] or are in development (third edition of [34]) in order to secure these communications. In this paper, we study the ARINC823 standard [2]. This standard aims to secure ACARS (Aircraft Communications Addressing and Reporting System) messages. These messages are short text messages, which include air traffic control messages, such as clearance messages, flight plans, weather information, as well as maintenance messages, so that needed maintenance operations can be planed at the next airport.

In addition to security (authentication and secrecy of the messages), other goals are apparent in the design of the standard:

- Bandwidth: the available bandwidth is limited and the ACARS messages are relatively short, so the protocol is designed to minimize the additional bandwidth required for security.
- Resistance to failures: the messages are conveyed even in case of failure of the security system.
- Flexibility: the protocol is designed to provide different levels of security: authentication and secrecy, authentication only, and no security, on a message-per-message basis.

These additional goals justify the design of specific protocols. The ARINC823 standard provides two such protocols: a public-key protocol and a shared-key protocol. (The latter assumes that the aircraft and the ground entity share a long-term secret key.) To our knowledge, the public-key protocol has been implemented and is currently used by one major airline. We do not know of a current usage of the shared-key protocol, but it might be used in the future.

Other avionic protocols share the constraints of low bandwidth and resistance to failures, but some of them are much simpler. For instance, the third edition of [34] just aims at authentication and does not require flexibility, so its analysis does not need to consider as many cases as the one of the ARINC823 standard.

In order to obtain guarantees that the ARINC823 protocols are secure, we use mechanized protocol verification tools. These tools typically rely on one of the following two models of cryptography:

- The *symbolic model*, so-called Dolev-Yao model [31, 39], is an abstract model in which cryptographic primitives are considered as perfect black-boxes, messages are terms on these primitives, and the adversary is restricted to apply only these primitives. This model facilitates formal verification, and many automatic protocol verifiers (AVISPA [3], F7 [12], FDR [37], ProVerif [21], Scyther [28], Tamarin [45], . . .) exist in this model.
- The *computational model* [48], typically used for manual cryptographic proofs, is more concrete. In this model, messages are bitstrings, cryptographic primitives are functions on these bitstrings, and the adversary is any probabilistic Turing machine. Formal verification is more difficult in this model, and most proofs are manual. Some mechanized verifiers exist (CryptoVerif [17, 18], CertiCrypt [7, 8] and its successor EasyCrypt [4, 6], Computational F7 [15] and its successors F* and RF* [5]).

We consider both models in our study, and rely on ProVerif (`http://proverif.inria.fr`) for the symbolic analysis and on CryptoVerif (`http://cryptoverif.inria.fr`) for the computational analysis. Our analysis does not consider side-channels (such as timing and power consumption), and works on a model of the protocol, not on an implementation. Our goal is to verify the ARINC823 standard, not a particular implementation (to which we did not have access). Having a secure standard is an essential first step, even if additional weaknesses may appear at the implementation level.

1.1 Contributions

We formally model both the public-key (Section 2) and the shared-key (Section 3) ARINC823 protocols, in ProVerif and CryptoVerif. Our models are available at `http://prosecco.inria.fr/personal/bblanche/arinc823/` We study many security properties of these protocols: freshness of messages, absence of replays, authentication, secrecy, forward secrecy, resistance to key compromise impersonation attacks. While we confirm important properties (the public-key protocol guarantees secrecy and authentication of the application data messages), we also find several weaknesses and imprecisions in the standard, in particular related to the resistance to failures and to the protection against replays. In the shared-key protocol, a subtle, computational replay attack may lead to the breach of secrecy. We propose fixes for these weaknesses, and prove their security. Moreover, the protocols use undesirable cryptographic constructions; in particular, they share keys between several primitives. The public-key protocol does not satisfy forward secrecy and is subject to key compromise impersonation attacks. While these security properties may not be explicit requirements of the standard, the protocol can be strengthened to guarantee them and to avoid key sharing between primitives. We propose such a strengthened protocol and verify it.

We reported our results to Honeywell, the industry editor for and a key technical contributor to the ARINC823 standard. Honeywell gave us helpful feedback on our analysis of the public-key protocol, which they implement. We summarize this feedback in the rest of this paper.

This case study required the specification of new cryptographic primitives in CryptoVerif: a joint assumption between gap Diffie-Hellman and signatures (Section 2.3 *2)*), and a counter-mode encryption scheme (Section 2.3 *5)*). As we shall see, this case study also illustrates the complementarity between verification in the symbolic and in the computational models and the usage of manual arguments in order to complement some ProVerif proofs.

1.2 Related Work

Other approaches to protocol verification have already been discussed above. Many other cases studies have been conducted using ProVerif (see [16] for references), a few using CryptoVerif [1, 14, 20, 22, 24, 46]. This study is one of the rare that combine both tools [13, 36]: [36] studies the Signal messaging protocol and [13] studies TLS 1.3 Draft-18. Those two studies generate ProVerif scripts from JavaScript code for symbolic verification, and additionally use manually written CryptoVerif scripts to obtain computational guarantees. This study is probably the one that exploits most the complementarity between these tools. We do not know of any previous formal verification of the ARINC823 protocols.

2 AMS Protocol using Public/Private Keys

In this section, we study the AMS (ACARS Message Security) protocol using public/private keys as defined in [2, Attachment 7].

Aircraft U *Ground entity* V

Ground-initiated trigger

AMS_Init_IND message

$$m_0 = (H_0, \mathsf{Init\_IND}, \mathsf{U}, \mathsf{V}, [\mathit{cert}_{\mathsf{V}}], t_{\mathsf{V}})$$

$$s_{\mathsf{V}} = \mathsf{SIG}(d_{\mathsf{V}}, (\mathsf{V}, \mathsf{U}, m_0))$$

$$\longleftarrow$$

Secure session initiation

AMS_Init_REQ message

$$m_1 = (H_1, \mathsf{Init\_REQ}, \mathit{Policy}, \mathsf{U}, \mathsf{V}, \mathit{Algos}, t_{\mathsf{U}})$$

$$s_{\mathsf{U}} = \mathsf{SIG}(d_{\mathsf{U}}, (\mathsf{U}, \mathsf{V}, m_1))$$

$$\longrightarrow$$

AMS_Init_RSP message

$$m_2 = (H_2, \mathsf{Init\_RSP}{+}, \mathit{AlgSel}, [\mathit{cert}_{\mathsf{V}}], t_{\mathsf{U}}, \mathit{Rand}_{\mathsf{V}})$$

$$\mathsf{MAC}(\mathit{KMAC}_{\mathsf{U},\mathsf{V}}, (\mathsf{V}, \mathsf{U}, m_2, \mathit{Algos}, t_{\mathsf{U}}, s_{\mathsf{U}}))$$

$$\longleftarrow$$

Application data exchange

$$m = (H, \mathit{Cmd}, \mathsf{ENC}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, (0, C_{\mathsf{DN}}), \mathit{Payload}), C_{\mathsf{DN}})$$

$$\mathsf{MAC}(\mathit{KMAC}_{\mathsf{U},\mathsf{V}}, (\mathsf{U}, \mathsf{V}, m))$$

$$\longrightarrow$$

$$m = (H, \mathit{Cmd}, \mathsf{ENC}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, (1, C_{\mathsf{UP}}), \mathit{Payload}), C_{\mathsf{UP}})$$

$$\mathsf{MAC}(\mathit{KMAC}_{\mathsf{U},\mathsf{V}}, (\mathsf{V}, \mathsf{U}, m))$$

$$\longleftarrow$$

Figure 1: Simplified AMS protocol using public/private keys

2.1 Protocol Description

Let us first provide a simplified description of the protocol; our models cover the full specification. The protocol establishes a secure session between an aircraft U and a ground entity V, as shown in Figure 1. The aircraft initially has an elliptic curve key pair (secret key d_{U} and public key Q_{U}) and similarly the ground entity has an elliptic curve key pair (secret key d_{V} and public key Q_{V}). The ground entity may request that the aircraft initiates a secure session. In this case, it sends a ground-initiated trigger, AMS_Init_IND, which consists of a header field H_0, a constant command $\mathsf{Init\_IND}$ for initiation indication, the identities of the aircraft U and ground entity V, optionally the certificate $\mathit{cert}_{\mathsf{V}}$ for the public key Q_{V} of V (the certificate must be present if the aircraft does not already have it), and a timestamp t_{V}. This message is signed with the secret key d_{V}. Upon receipt of this message, the aircraft verifies its own identity, the certificate if present, the signature, and verifies that the timestamp t_{V} is recent enough:

$$t - 60 \text{ seconds} \leq t_{\mathsf{V}} \leq t + 120 \text{ seconds}.$$

where t is the current time at the aircraft. Then, it initiates a secure session. The aircraft may also directly initiate a secure session; the ground-initiated trigger is then omitted.

To initiate a secure session, the aircraft sends an AMS_Init_REQ message, which consists of a header field H_1, a constant command $\mathsf{Init\_REQ}$ for initiation request, an identifier *Policy* for a security policy that determines when messages are encrypted and/or MACed, the identities of the aircraft U and ground entity V, the algorithms *Algos* that the aircraft supports (encryption, MAC, compression), and a timestamp t_{U}. This message is signed with the secret key d_{U}. Upon receipt of this message, the ground entity V verifies its own identity, gets the aircraft certificate for the public key Q_{U} from its local storage or from a PKI certificate distribution service, verifies the signature, verifies that $(\mathit{Algos}, t_{\mathsf{U}}, s_{\mathsf{U}})$ is unique for the aircraft initiating a new session, and verifies that the timestamp t_{U} is recent enough:

$$t - 60 \text{ seconds} \leq t_{\mathsf{U}} \leq t + 120 \text{ seconds}. \tag{1}$$

where t is the current time at the ground entity. It selects supported algorithms among those proposed by the aircraft, and stores them in *AlgSel*, computes a shared Diffie-Hellman secret $Z_{\mathsf{U},\mathsf{V}} = d_{\mathsf{V}} Q_{\mathsf{U}}$, chooses a 32-bit random value $\mathit{Rand}_{\mathsf{V}}$, and computes the following values:

$$\begin{aligned} X_{\mathsf{U},\mathsf{V}} &= \mathsf{HASH}((\mathit{Algos}, t_{\mathsf{U}}, s_{\mathsf{U}}, \mathit{Rand}_{\mathsf{V}})) \\ \mathit{KMAC}_{\mathsf{U},\mathsf{V}} &= \mathsf{KDF}(Z_{\mathsf{U},\mathsf{V}}; m; \mathsf{concatSI}(1, \mathit{MAC\_algo}, X_{\mathsf{U},\mathsf{V}}, \mathsf{U}, \mathsf{V}, m)) \\ \mathit{KENC}_{\mathsf{U},\mathsf{V}} &= \mathsf{KDF}(Z_{\mathsf{U},\mathsf{V}}; n; \mathsf{concatSI}(2, \mathit{ENC\_algo}, X_{\mathsf{U},\mathsf{V}}, \mathsf{U}, \mathsf{V}, n)) \end{aligned}$$

where the hash function HASH is SHA256, $\mathit{MAC\_algo}$ is the selected MAC algorithm and m is the length of its keys, $\mathit{ENC\_algo}$ is the selected encryption algorithm and n is the length of its keys, the function $\mathsf{concatSI}$ concatenates its arguments, the key derivation function KDF is defined by ANSI X9.63 but uses SHA256 instead of SHA1 as underlying hash function, and $\mathsf{KDF}(Z; m; X)$ returns an m-bit key.

Then the ground entity sends the AMS_Init_RSP message, which consists of a header field H_2, a constant command $\mathsf{Init\_RSP}+$ for a positive initiation response (a $\mathsf{Init\_RSP}-$ message may be sent when the ground entity rejects the secure session), the selected algorithms *AlgSel*, optionally the certificate $\mathit{cert}_{\mathsf{V}}$ for the public key Q_{V} of V (the certificate must be present if the aircraft does not already have it), the timestamp t_{U}, and the random value $\mathit{Rand}_{\mathsf{V}}$. This message is MACed together with the fields $\mathit{Algos}, t_{\mathsf{U}}, s_{\mathsf{U}}$ received in the AMS_Init_REQ message, using the key $\mathit{KMAC}_{\mathsf{U},\mathsf{V}}$.

Upon receipt of this message, the aircraft verifies that the received timestamp t_{U} is equal to the one it sent in the AMS_Init_REQ message, verifies the certificate if present, computes the shared Diffie-Hellman secret $Z_{\mathsf{U},\mathsf{V}} = d_{\mathsf{U}} Q_{\mathsf{V}}$, computes $X_{\mathsf{U},\mathsf{V}}$, $\mathit{KMAC}_{\mathsf{U},\mathsf{V}}$, and $\mathit{KENC}_{\mathsf{U},\mathsf{V}}$ as above, and verifies the MAC.

When all these steps succeed, the aircraft and ground entity can exchange application data messages in an encrypt-then-MAC tunnel, using $\mathit{KENC}_{\mathsf{U},\mathsf{V}}$ as encryption key and $\mathit{KMAC}_{\mathsf{U},\mathsf{V}}$ as MAC key. They use two distinct counters C_{DN} and C_{UP}, one in each direction, to order the messages and to prevent replays. These counters are incremented when a message is sent or received in the considered direction.

The specified MAC algorithm is HMAC-SHA256, truncated to 32, 64, or 128 bits. The specified encryption algorithm is AES128-CFB128. The specified compression algorithms are Dynamic Markov Compression and DEFLATE. These algorithms are chosen with bandwidth in mind: the MACs are rather short, the encryption algorithm is such that the length of the ciphertext is equal to the length of the plaintext (at the bit level), and compression may be used. The protocol allows specifying additional algorithms in the future, which are negotiated via the fields *Algos* and *AlgSel*. The MAC and encryption algorithms are chosen at the beginning of the protocol. The compression algorithm may vary for each message, among the supported algorithms. The chosen compression algorithm is included in the header field H of each message. When compression is active, the payload is compressed before encryption.

The protocol offers 4 protection modes for the messages:

- BOTH: the message is encrypted then MACed.
- AUTH: the message is just MACed.

- NONE: no encryption nor MAC.
- SIGN: the message is signed.

Protection mode SIGN is used only in the AMS_Init_IND and AMS_Init_REQ messages. The AMS_Init_RSP message always uses protection mode AUTH. The application data messages may use the modes BOTH (as represented in Figure 1), AUTH, or NONE. The protection mode of each message is included in the header field H. It is chosen based on the security policy of identifier *Policy* and the type of message sent. Finally, the aircraft may or may not support encryption. In the latter case, messages are never encrypted, even in protection mode BOTH.

2.2 Modeling Notes

Our models of this protocol have the following features:

1) Sessions. The models consider a single honest aircraft U and a single honest ground entity V. Other aircrafts and ground entities are included in the adversary. The aircraft U is willing to run the protocol with any ground entity, and the ground entity V is willing to run the protocol with any aircraft. We prove security properties for the sessions between the honest aircraft U and ground entity V.

The public-key protocol and the shared-key protocol do not share secrets, so we can consider them separately without missing attacks: the shared-key protocol is supposed to be included in the adversary when we analyze the public-key protocol.

The models consider an unbounded number of executions of the protocol.

2) Algorithms. The models consider only the algorithms currently specified for compression, encryption, and MAC. We do not model any algorithms reserved for future use. In the future, it might be interesting to verify that the algorithm negotiation remains secure even if algorithms are added.

3) Certificates. The certificates are modeled in a fairly abstract way: the certificate that Q_{U} is the public key of U is modeled as a signature of the pair $(\mathsf{U}, Q_{\mathsf{U}})$ under the secret key sCA of a certificate authority.

4) Equations for encoding and compression. To model that encoding and compression are the identity when they are OFF, ProVerif uses equations $\mathsf{encode}(\mathsf{encode\_OFF}, x) = x$ and $\mathsf{compress}(\mathsf{comp\_OFF}, x) = x$. These equations considerably increase the runtime and memory consumption of ProVerif, to the point that ProVerif could not prove authentication for the public-key protocol with these equations. Indeed, with these equations, ProVerif needs to consider two cases for each encode operation: either the mode is $\mathsf{encode\_OFF}$, and the encode operation is the identity, or it is another mode, and encode is a non-identity function, considered as a free function symbol. The same holds for $\mathsf{compress}$ and for the reverse operations, so the number of cases to consider doubles for each encode, decode, $\mathsf{compress}$, and $\mathsf{decompress}$. Therefore, we omit these equations, to avoid such case distinctions. CryptoVerif proves desired properties for any injective encoding and compression functions, in particular those that satisfy these equations.

5) Timestamps. The specification [2, Table 7.2.1-1, Step N04, Item 3] says that a timestamp t_{U} is accepted at time t when (1) holds. We found this inequality surprising: if we suppose that the message may take between 0 and 60 seconds to go from one entity to the other, and the clocks can differ by at most 60 seconds, we obtain

$$t - 120 \text{ seconds} \leq t_{\mathsf{U}} \leq t + 60 \text{ seconds} \tag{2}$$

rather than the inequality (1). Honeywell confirmed that there is a typo in the standard and that (2) is the correct inequality. We rely on the corrected inequality in the rest of this document.

We use the following abstract model of timestamps. We define a channel timech on which we input the current time, and a predicate check_time such that check_time(t_{U}, t) is true if and only if (2). In the models, since ProVerif and CryptoVerif cannot model time precisely, the channel timech is in fact a public channel, so the adversary can send anything on it, and the definition of the predicate check_time is left unspecified, so the proofs done by ProVerif and CryptoVerif are valid for any definition of check_time.

6) Replay protection. We do not model the check mentioned in [2, Table 7.2.1-1, Step N07, Item 4]:

> [The ground entity] verifies that the received AMS_Appendix field [$Algos, t_{\mathsf{U}}, s_{\mathsf{U}}$ in Figure 1] is unique for the aircraft initiating a new session.
>
> Note 1—This check is necessary to ensure that the received AMS_Init_REQ message is associated with a new initiation request and not an ACARS retransmission of an AMS_Init_REQ message for which a secure session is already established.

Above, the AMS_Appendix field is considered non-unique when a session with the aircraft is established with that appendix [2, Table 7.2.1-2, Step N07, Case 2].

Even though replays of AMS_Init_REQ messages may not be problematic, we clarify that, in order to avoid all such replays, it is not enough to verify that the received AMS_Appendix field does not occur in the AMS_Init_REQ messages of currently established sessions. With this check, in case a session terminates at a time t such that $t - 120$ seconds $\leq t_{\mathsf{U}}$, the adversary can replay the initial AMS_Init_REQ message just after the session terminates, and this message will be accepted. To avoid all replays, the check should be defined as follows.

For each aircraft U, the ground entity should remember the AMS_Appendix field of the AMS_Init_REQ messages received from U with the most recent timestamp t_{U}, as long as t_{U} is such that $t - 120$ seconds $\leq t_{\mathsf{U}}$ where t is the ground entity's local time. (Replays of AMS_Init_REQ messages older than 120 seconds are discarded by checking the timestamp.) A new AMS_Init_REQ message received from U should be accepted if and only if its timestamp t_{U} satisfies (2), and it is strictly more recent than the remembered timestamp t_{U} or it is equal to the remembered t_{U} and AMS_Appendix field is not among the remembered AMS_Appendix fields. (The AMS_Appendix field includes the signature s_{U} which is probabilistic, so each session uses a different signature, and hence a different AMS_Appendix field. A message with an AMS_Appendix field that has already been seen is therefore a replay.) This check guarantees that the ground entity never accepts two AMS_Init_REQ messages with the same $\mathsf{U}, s_{\mathsf{U}}, t_{\mathsf{U}}$.

Moreover, in the case of a ground-initiated trigger, the uniqueness of the AMS_Appendix field is never checked. As a result, the adversary can replay the AMS_Init_IND message as long as its timestamp t_{V} satisfies the inequality $t - 120$ seconds $\leq t_{\mathsf{V}} \leq t + 60$ seconds, where t is the aircraft entity's local time. In particular, if the adversary replays the AMS_Init_IND message while a secure session is established between the aircraft and the ground entity, this message will be accepted by the aircraft; according to [2, Table 7.2.2-1, Step G08, Item 2], this replay will cause the just started session to end and a new secure session to be initiated. If the entities handle a secure session restart by sending again the messages of the previous session, the attack may cause messages to be delayed by up to 120 seconds (perhaps using several replays of the first AMS_Init_IND message). It may also have more serious consequences if the entities forget the messages they cannot send in the initial session due to the unexpected termination. To avoid this attack, we recommend protecting against replays of the AMS_Init_IND message by using a check similar to the one mentioned above for the AMS_Init_REQ message. Honeywell agrees with the anti-replay checks we recommend.

7) Message counter. ProVerif and CryptoVerif cannot fully model the incrementation of the message counter and the tests performed on this counter to accept or reject the message. In

ProVerif, the value of the counter is input on a public channel and the tests are omitted. The CryptoVerif model is more precise: it models that the sender never reuses the same counter in the same session, and that the receiver rejects a message when it has already seen a message with the same counter in the same session. It still omits that

$$\textit{highest received count} - 5 \leq \textit{current received count} \leq \textit{highest received count} + 5$$

as mentioned in the message count windowing criteria [2, Table 7.3.1-1, Step D08]. (These windowing criteria allow a counter smaller than the highest received counter, to allow receiving out-of-order messages, which may happen when the network is not entirely reliable.) The proved security properties hold a fortiori with this additional check.

8) Resistance to failures. [2, page 133, note 1] reads:

> If triggering secure session initiation is unsuccessful after an initial attempt and one re-attempt, then the ground entity assumes AMS is unavailable for the aircraft entity and sends queued messages in the clear.

Similarly, [2, page 130, note 1] reads:

> If secure session initiation is unsuccessful after an initial attempt and a configurable number of re-attempts, then the aircraft entity executes a configurable unavailable action, as described in Section 3.5.4.

and [2, Section 3.5.4] specifies

> Possible actions may include, but are not limited to, the following:
>
> - Attempt to send the message to an alternate AMS-equipped entity specified by the airline.
> - Send the message in the clear bypassing AMS.
> - Do not send the message.

Honeywell points out that, when an AMS-equipped aircraft entity is not able to establish a secure session with a ground entity, stakeholders (e.g., avionics suppliers, datalink service providers, airlines) that contributed to the development of the standard agreed that the ARINC 823 specification should include flexibility such that implementers may provide such customer-configurable actions.

This behavior may cause obvious security breaches when messages are sent in the clear: if an adversary manages to intercept or jam messages so that the secure session initiation fails even though AMS is actually available, then it will get the sent messages in the clear, thus breaking secrecy.

The intention of communicating the messages even in case of failure of the security system is understandable, but it conflicts with the goal of securing the communication: security is guaranteed only when a secure session is established. We recommend taking additional actions in case secure session initiation fails or messages are sent or received outside secure sessions, such as informing the pilot and reporting the problem to a computer maintenance team on the ground. We recommend mentioning that in the standard, even though Honeywell points out that user notifications and problem reporting are considered local implementation issues that are not within scope of the AMS interoperability specification. Our models include only the behavior inside secure sessions, so in our models, no message is sent in the clear when secure session initiation fails. With the migitations mentioned above, our security results still hold for the full protocol when no problem is reported to the pilot or to the ground maintenance.

9) Policy check. The models include the verification that messages were transmitted with the correct protection mode, as specified by the security policy [2, Section 3.6.2, pages 29–31].

We model these checks in CryptoVerif using a function get_policy such that get_policy(*policy*, *msg*) is the protection mode (NONE, AUTH, or BOTH) of the message *msg* in the policy *policy*. The function get_policy is left unspecified, so that the result holds for any such function. In ProVerif, we cannot leave functions unspecified, we can only leave predicates unspecified (by declaring that they "**block**"), so we use a predicate policy_ok(*policy*, *prot*, *msg*) to mean that *prot* is the protection mode of *msg* in the policy *policy*.

These checks are essential for security. In particular, an unprotected message should *never* be accepted when the policy specifies a protection AUTH or BOTH, otherwise the authenticity of the message can be broken. These checks are correctly mentioned as mandatory in the body of the specification [2, Section 3.6.2, page 29]:

> When a secure session is established between communicating peer aircraft and ground entities, the receiving entity verifies all received messages to ensure correct application of the security policy.

However, they are omitted in the description of the protocol in [2, Attachment 7]. We recommend adding them there. Honeywell agrees that it may be helpful to include a reference to Section 3.6.2 in Attachment 7, but points out that the commentary in Attachment 7, Section 7.1 clearly states that the AMS protocol process steps are included as guidance and the provisions in the main body of the document must be followed precisely.

Moreover, since messages may be sent in the clear when a secure session cannot be established (Section 2.2 *8)*), the receiving entity has to accept messages independently of the security policy outside secure sessions. This behavior comes with security risks: if no secure session is established, for instance because the adversary jammed secure session initiation messages, then the adversary can send its messages to the receiving entity, which will accept them, so authenticity will be broken. As we already noted in Section 2.2 *8)*, security is guaranteed only when a secure session is established, and our models include only the behavior inside secure sessions. In this case, the receiving entity always verifies the correct application of the security policy.

10) MAC length. The specification is not explicit on what the aircraft does with the MAC length that is part of the *AlgSel* field of the AMS_Init_RSP message that it receives from the ground entity. If the protocol runs normally, the ground entity sends back the MAC length chosen by the aircraft. However, [2, Table 7.2.1-1] does not say what the aircraft should do when the MAC length it receives differs from the one it sent.

We stress that the aircraft should not set the MAC length based on *AlgSel* (in contrast to what it does for the MAC algorithm). Indeed, suppose that the aircraft sets its MAC length to the length received in *AlgSel*. The aircraft has initially sent a MAC length l in the field *Algos* of the AMS_Init_REQ message. If the adversary cannot forge MACs of length l but can forge MACs of another length l', it can set the MAC length in *AlgSel* to l' and forge the MAC of the AMS_Init_RSP message for the length l' (either by modifying the AMS_Init_RSP message sent by the ground entity or by building its own). The aircraft would then accept this message. The adversary may then continue forging MACs of length l' for subsequent messages. This is a downgrading attack in which the achieved security level corresponds to the weakest MAC length (the shortest), even though the aircraft may have initially selected a stronger MAC length.

The following two behaviors avoid this problem:

- The aircraft ignores the MAC length in *AlgSel*, and always uses the MAC length it sent. Our models assume this behavior.

- The aircraft verifies that the MAC length in *AlgSel* is the same as the MAC length it sent. Otherwise, it discards the AMS_Init_RSP message.

Honeywell disagrees that the specification is not explicit on this point. They confirm that the MAC length is dictated by the aircraft and cannot be changed by the ground, and mention that the treatment of the MAC length is explained in [2, Table 7.2.1-1, Step N02, Item 1, Bullet 4]:

> $\mathit{AlgID}_{\mathsf{U}}$, aircraft entity crypto-algorithm identifier, which indicates the *configured* MAC length, and the message authentication and encryption algorithms supported by the aircraft entity

and in [2, Section 5.2.2]:

> The AMS Algorithm Select Type (AMS_AlgSelType) is a 16-bit parameter type used to *confirm* the selected MAC length and [...]

We still recommend making explicit the reaction of the aircraft when the MAC length it receives differs from the one it sent, to avoid any misinterpretation of the standard.

2.3 Security Assumptions on Primitives

ProVerif uses the symbolic model, in which messages are terms, so the model cannot be accurate at the bit level. Bitstring concatenation is modeled as tuples. Cryptographic primitives are supposed to be perfect. Diffie-Hellman key agreements are modeled by the equation $d_{\mathsf{U}}Q_{\mathsf{V}} = d_{\mathsf{V}}Q_{\mathsf{U}}$; all other algebraic relations are ignored.

In contrast, CryptoVerif relies on the computational model of cryptography. The CryptoVerif model is exact at the bit level. It uses the following assumptions:

1) Certificates. The signatures used in certificates are unforgeable under chosen-message attacks (UF-CMA) [33]. These signatures may use a scheme different from the one used in the protocol to sign messages under d_{U} or d_{V}.

2) Signatures and Diffie-Hellman. The long-term key pairs of the aircraft and the ground entity are used for ECDSA signatures and for Diffie-Hellman key exchanges. Therefore, we need a joint security assumption on the signatures and the Diffie-Hellman key exchange, which is non-standard. We assume that ECDSA signatures are unforgeable under chosen-message attacks (UF-CMA) [23] and that the gap Diffie-Hellman (GDH) assumption [43] holds with common keys.

For our anti-replay protection, we need strong unforgeability: if the signatures are just unforgeable, an adversary can replay the AMS_Init_REQ message by using a different signature for the same message, and thus pass our anti-replay protection. However, standard ECDSA signatures are malleable: if (r, s) is a valid signature, then so is $(r, (-s) \mod n)$ where n is the curve order. This malleability can be avoided by requiring that s is less than $n/2$ [47]. For the protocol with replay protection, we assume this additional restriction, so that ECDSA signatures are strongly unforgeable under chosen-message attacks (SUF-CMA) [23] and that the gap Diffie-Hellman (GDH) assumption [43] holds with common keys. We do not make this stronger assumption for the original protocol, since the standard does not mention such a restriction.

These assumptions are defined in CryptoVerif by allowing calls to signature oracles in the gap Diffie-Hellman assumption, and calls to Diffie-Hellman oracles in the (S)UF-CMA signature assumption (see Appendix A).

Additionally, we need a collision resistance property for signatures: two signatures (generated with distinct fresh randomness) have a negligible probability of being equal. This property holds for ECDSA signatures, because if two signatures with randomness r_1 and r_2 respectively are equal, then $r_1.G$ and $r_2.G$ have the same first coordinate modulo n, where G is the considered group generator and n is its order. This property has a negligible probability of happening for independent random r_1 and r_2. This property is important for the security of the protocol: using deterministic signatures would remove any randomness from the AMS_Init_REQ message, so that two AMS_Init_REQ messages with the same timestamp would be equal, leading to the same replay attacks as those we find for the shared-key protocol (Section 3.4).

3) Key derivation. We assume that the concatenation of the key derivation functions $\mathsf{KDF256}$ and $\mathsf{KDF128}$ defined by

$$\mathsf{KDF256}(k, (X, U, V)) = \mathsf{KDF}(k; 256; \mathsf{concatSI}(1, \mathsf{HMAC\text{-}SHA256}, X, U, V, 256))$$
$$\mathsf{KDF128}(k, (X, U, V)) = \mathsf{KDF}(k; 128; \mathsf{concatSI}(2, \mathsf{AES128\text{-}CFB128}, X, U, V, 128))$$

is a random oracle [11]. We justify this assumption in Appendix B, assuming that the compression function of SHA256 is a random oracle. Using a single random oracle and splitting its result into two keys yields more efficient verification in CryptoVerif than using two independent random oracles.

4) MAC. We assume that the truncation of HMAC-SHA256 (to 32, 64, and 128 bits, as used in the protocol) is an SUF-CMA MAC. This property holds assuming that HMAC-SHA256 is a pseudo-random function (PRF). Indeed, the truncation of a PRF is also a PRF, and a PRF is an SUF-CMA MAC [10, Section 2.4]. By [9], HMAC-SHA256 is a PRF if the compression function of SHA256 is a PRF, using its first argument as key, and also using its second argument as key under a related key attack.

5) Encryption. The initialization vector (IV) of the encryption scheme is computed as $IV = \mathsf{E}(KENC_{\mathsf{U,V}}, 0; IVdata)$ where $\mathsf{E}(k, IV, m)$ is the AES128-CFB128 encryption of m under the key k with IV IV and $IVdata$ is basically a counter [2, Table 7.3.1-1, Step D03, Item 1].

By definition of the CFB encryption mode [40], we have

$$IV = \mathsf{AES128}(KENC_{\mathsf{U,V}}, 0) \oplus IVdata \tag{3}$$

where $\oplus$ is exclusive or. However, [40, Appendix C] says:

> For the CBC and CFB modes, the IVs must be unpredictable. In particular, for any given plaintext, it must not be possible to predict the IV that will be associated to the plaintext in advance of the generation of the IV.
>
> There are two recommended methods for generating unpredictable IVs. The first method is to apply the forward cipher function, under the same key that is used for the encryption of the plaintext, to a nonce. The nonce must be a data block that is unique to each execution of the encryption operation. [...] The second method is to generate a random data block using a FIPS-approved random number generator.

The generation of the IV in the AMS protocol does not follow these recommended methods for generating unpredictable IVs. The first recommended method would yield

$$IV = \mathsf{AES128}(KENC_{\mathsf{U,V}}, IVdata) \tag{4}$$

Assuming that AES128 is a pseudo-random permutation (PRP) [38], formula (4) is indistinguishable from generating a fresh random IV for each new value of $IVdata$, so the IVs are unpredictable as desired.

In contrast, formula (3) allows one to predict the whole sequence of IVs knowing the first IV. Fortunately, the IVs remains secret in the AMS protocol, so we can still prove the protocol secure, at the cost of a less standard proof.

We assume that AES128 is a PRP. We let

$$\begin{aligned}\mathsf{ENC}(k, \mathit{IVdata}, m) &= \mathsf{E}(k, \mathsf{E}(k, 0; \mathit{IVdata}); m)\\ \mathsf{DEC}(k, \mathit{IVdata}, c) &= \mathsf{D}(k, \mathsf{E}(k, 0; \mathit{IVdata}); c)\end{aligned}$$

and we show that $\mathsf{DEC}(k, \mathit{IVdata}, \mathsf{ENC}(k, \mathit{IVdata}, m)) = m$ and that the probability of distinguishing

$$\begin{aligned}&\mathcal{O}_{\text{gen}}() := k \overset{R}{\leftarrow} \mathsf{enc\_key}; \mathbf{return}(); \mathbf{foreach}\ i \leq n\ \mathbf{do}\\ &\quad \mathcal{O}_{\text{enc}}(\mathit{IVdata}, m) := \mathbf{return}(\mathsf{ENC}(k, \mathit{IVdata}, m))\end{aligned} \tag{5}$$

from

$$\begin{aligned}&\mathcal{O}_{\text{gen}}() := k \overset{R}{\leftarrow} \mathsf{enc\_key}; \mathbf{return}(); \mathbf{foreach}\ i \leq n\ \mathbf{do}\\ &\quad \mathcal{O}_{\text{enc}}(\mathit{IVdata}, m) :=\\ &\qquad \mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(\mathit{IVdata}[j]) \wedge \mathit{IVdata} = \mathit{IVdata}[j]\ \mathbf{then}\\ &\qquad\quad \mathbf{event}\ \mathsf{repeatedIV}\\ &\qquad \mathbf{else}\\ &\qquad\quad \mathbf{return}(\mathsf{ENC}(k, \mathit{IVdata}, Z(m)))\end{aligned} \tag{6}$$

is at most the probability of event repeatedIV plus a negligible probability, where $Z(m)$ is the bitstring of the same length as m and consisting only of zeroes. The game (5) defines an oracle $\mathcal{O}_{\text{gen}}$ that generates an encryption key k, chosen randomly with uniform probability in type enc_key. Then, this oracle returns control to the adversary and n copies of oracle $\mathcal{O}_{\text{enc}}$ are defined, which return the encryption of message m using the IV data *IVdata*. The game (6) is similar but it executes event repeatedIV when *IVdata* has been previously used, and encrypts $Z(m)$ instead of m. This property shows that, when the *IVdata* is never reused, the adversary cannot distinguish the encryption of m from the encryption of $Z(m)$. Because CryptoVerif lacks loops, we prove this property only on an example with 3 messages ($n = 3$), containing respectively 1, 2, and 3 blocks. (Details on this proof can be found in Appendix C.) It is easy to see manually that this result generalizes to any number of messages with any number of blocks. We use that general result as assumption in the proof of the protocol.

6) Hash function. We assume that SHA256 is collision-resistant. This property holds in particular when the compression function of SHA256 is collision-resistant [29].

2.4 Security Analysis

Having modeled the protocol, we can use ProVerif and CryptoVerif to verify its security properties:

1) Freshness of AMS_Init_REQ and AMS_Init_IND messages. ProVerif proves this freshness, by showing the following correspondences (7) and (8):

$$\begin{aligned}&\mathbf{event}(\mathsf{recv\_Init\_REQ}(\mathsf{V}, \mathsf{U}, x, s_{\mathsf{U}}, t_{\mathsf{U}}, t_{\mathsf{V}})) \Longrightarrow\\ &\quad \mathsf{check\_time}(t_{\mathsf{U}}, t_{\mathsf{V}}) \wedge \mathbf{event}(\mathsf{send\_Init\_REQ}(\mathsf{U}, \mathsf{V}, x, s_{\mathsf{U}}, t_{\mathsf{U}})).\end{aligned} \tag{7}$$

The event $\mathsf{recv\_Init\_REQ}(\mathsf{V}, \mathsf{U}, x, s_{\mathsf{U}}, t_{\mathsf{U}}, t_{\mathsf{V}})$ is executed when the ground entity V accepts an AMS_Init_REQ message x, s_{U} from the aircraft U at time t_{V}, and this message has timestamp t_{U}. (x is the message without signature, s_{U} is its signature.) Similarly, the event $\mathsf{send\_Init\_REQ}(\mathsf{U}, \mathsf{V}, x, s_{\mathsf{U}}, t_{\mathsf{U}})$ is executed when U sends an AMS_Init_REQ message x, s_{U} to V with timestamp

t_{U}. Hence, this correspondence means that, if V accepts an AMS_Init_REQ message at time t_{V} from U, and this message has signature s_{U} and timestamp t_{U}, then the message has been sent by U to V, with the same signature s_{U}, at time t_{U} and $\mathsf{check\_time}(t_{\mathsf{U}}, t_{\mathsf{V}})$, that is, the inequality (2) holds.

The following correspondence proves a similar property for the AMS_Init_IND message:

$$\begin{aligned}&\mathbf{event}(\mathsf{recv\_Init\_IND}(\mathsf{U}, \mathsf{V}, x, s_{\mathsf{V}}, t_{\mathsf{V}}, t_{\mathsf{U}})) \Longrightarrow \\ &\quad \mathsf{check\_time}(t_{\mathsf{V}}, t_{\mathsf{U}}) \wedge \mathbf{event}(\mathsf{send\_Init\_IND}(\mathsf{V}, \mathsf{U}, x, s_{\mathsf{V}}, t_{\mathsf{V}})).\end{aligned} \tag{8}$$

The authenticity of the timestamp comes from the signature, and the freshness is checked by the receiver of the message.

CryptoVerif proves slightly weaker properties, due to the malleability of ECDSA signatures. The signatures may be different on the ground and aircraft sides, so CryptoVerif proves the correspondences[1]:

$$\begin{aligned}&\mathbf{event}(\mathsf{recv\_Init\_REQ}(\mathsf{V}, \mathsf{U}, x, s_{\mathsf{U}}, t_{\mathsf{U}}, t_{\mathsf{V}})) \Longrightarrow \\ &\quad \mathsf{check\_time}(t_{\mathsf{U}}, t_{\mathsf{V}}) \wedge \mathbf{event}(\mathsf{send\_Init\_REQ}(\mathsf{U}, \mathsf{V}, x, s'_{\mathsf{U}}, t_{\mathsf{U}})).\end{aligned} \tag{9}$$

$$\begin{aligned}&\mathbf{event}(\mathsf{recv\_Init\_IND}(\mathsf{U}, \mathsf{V}, x, s_{\mathsf{V}}, t_{\mathsf{V}}, t_{\mathsf{U}})) \Longrightarrow \\ &\quad \mathsf{check\_time}(t_{\mathsf{V}}, t_{\mathsf{U}}) \wedge \mathbf{event}(\mathsf{send\_Init\_IND}(\mathsf{V}, \mathsf{U}, x, s'_{\mathsf{V}}, t_{\mathsf{V}})).\end{aligned} \tag{10}$$

For the protocol with replay protection, we assume SUF-CMA signatures. Hence CryptoVerif proves (7) and (8).

Obviously, for a proper functioning of the protocol, the clocks of the aircraft and of the ground entity must remain synchronized. An adversary could perform attacks if it managed to desynchronize these clocks; such attacks are outside the scope of our model. (Using nonces instead of timestamps could avoid such attacks.)

2) No replay for AMS_Init_REQ and AMS_Init_IND messages. To prove this property, we need to consider our improved replay protection, since the original protocol does not guarantee the absence of replays. This property is modeled by injective variants of (7) and (8):

$$\begin{aligned}&\mathbf{inj\text{-}event}(\mathsf{recv\_Init\_REQ}(\mathsf{V}, \mathsf{U}, x, s_{\mathsf{U}}, t_{\mathsf{U}}, t_{\mathsf{V}})) \Longrightarrow \\ &\quad \mathsf{check\_time}(t_{\mathsf{U}}, t_{\mathsf{V}}) \wedge \mathbf{inj\text{-}event}(\mathsf{send\_Init\_REQ}(\mathsf{U}, \mathsf{V}, x, s_{\mathsf{U}}, t_{\mathsf{U}})).\end{aligned} \tag{11}$$

$$\begin{aligned}&\mathbf{inj\text{-}event}(\mathsf{recv\_Init\_IND}(\mathsf{U}, \mathsf{V}, x, s_{\mathsf{V}}, t_{\mathsf{V}}, t_{\mathsf{U}})) \Longrightarrow \\ &\quad \mathsf{check\_time}(t_{\mathsf{V}}, t_{\mathsf{U}}) \wedge \mathbf{inj\text{-}event}(\mathsf{send\_Init\_IND}(\mathsf{V}, \mathsf{U}, x, s_{\mathsf{V}}, t_{\mathsf{V}})).\end{aligned} \tag{12}$$

These correspondences guarantee that each reception of the AMS_Init_REQ message on V corresponds to a *distinct* emission of that message on U, and similarly for AMS_Init_IND. Injectivity is enforced by the keyword **inj-event**.

In ProVerif, we prove these correspondences by combining (7) and (8) with a manual argument, which relies on the following observation.

Lemma 1. *If the correspondence* $\mathbf{event}(e(x, M)) \Longrightarrow \phi \wedge \mathbf{event}(e'(x, M'))$ *holds and event* $e(x, M)$ *cannot be executed several times with the same value of* x*, then the injective correspondence* $\mathbf{inj\text{-}event}(e(x, M)) \Longrightarrow \phi \wedge \mathbf{inj\text{-}event}(e'(x, M'))$ *also holds.*

Proof. If a trace contains n executions of $e(x, M)$, they are $e(x, M)\sigma_1, \ldots, e(x, M)\sigma_n$ for substitutions $\sigma_1, \ldots, \sigma_n$ with pairwise distinct values of $x\sigma_1, \ldots, x\sigma_n$. Then, by the correspondence

[1] For uniformity, we use the ProVerif syntax for all correspondences. The actual syntax in CryptoVerif differs in minor ways.

$\mathbf{event}(e(x,M)) \implies \phi \wedge \mathbf{event}(e'(x,M))$, for all $i \leq n$, we have that $\phi\sigma_i$ holds and $e'(x,M)\sigma_i$ is executed in this trace. Furthermore, these executions of $e'(x,M)\sigma_i$ are distinct for distinct i, because their arguments $x\sigma_i$ are distinct for distinct i. This property proves the injective correspondence $\mathbf{inj\text{-}event}(e(x,M)) \implies \phi \wedge \mathbf{inj\text{-}event}(e'(x,M'))$. □

This observation is not new [19, before Corollary 3, page 406]. It obviously remains true when x and M are replaced with several variables and terms respectively, and when the arguments of events are reordered.

Our improved replay protection guarantees that the ground entity never accepts two AMS_Init_REQ messages with the same $\mathsf{U}, s_\mathsf{U}, t_\mathsf{U}$. Hence distinct events $\mathsf{recv\_Init\_REQ}(\mathsf{V}, \mathsf{U}, x, s_\mathsf{U}, t_\mathsf{U}, t_\mathsf{V})$ have distinct triples $\mathsf{U}, s_\mathsf{U}, t_\mathsf{U}$, so by correspondence (7) and Lemma 1, (11) holds. The correspondence (12) is proved similarly. Indeed, ProVerif is unable to prove injectivity automatically when it is obtained using a replay cache: injectivity relies on the event $\mathsf{recv\_Init\_REQ}$ being executed only when a lookup in the replay cache fails. However, in its internal representation by Horn clauses, ProVerif overapproximates the behaviors and considers that this lookup may always fail, so the required information is lost.

In contrast, CryptoVerif treats lookup failures precisely, so in CryptoVerif, we model our improved replay check, which guarantees that the AMS_Init_REQ and AMS_Init_IND messages are never accepted with the same identity, timestamp, and signature. With this improved check, CryptoVerif shows (11) and (12).

3) Entity authentication. ProVerif and CryptoVerif prove the correspondence (7) or (9), which proves non-injective authentication of the aircraft to the ground entity. Furthermore, with our improved replay check, (11) proves injective authentication of the aircraft to the ground entity.

ProVerif and CryptoVerif also prove the correspondence:

$$\begin{aligned}&\mathbf{inj\text{-}event}(\mathsf{recv\_Init\_RSP}(\mathsf{U}, \mathsf{V}, x, s_\mathsf{U}, \mathit{Rand}_\mathsf{V}, t_\mathsf{U}, \mathit{KMAC}_{\mathsf{U},\mathsf{V}}, \mathit{KENC}_{\mathsf{U},\mathsf{V}})) \implies \\ &\quad \mathsf{check\_time}(t_\mathsf{U}, t_\mathsf{V}) \wedge \\ &\quad \mathbf{inj\text{-}event}(\mathsf{send\_Init\_RSP}(\mathsf{V}, \mathsf{U}, x, s_\mathsf{U}, \mathit{Rand}_\mathsf{V}, t_\mathsf{U}, t_\mathsf{V}, \mathit{KMAC}_{\mathsf{U},\mathsf{V}}, \mathit{KENC}_{\mathsf{U},\mathsf{V}})).\end{aligned} \tag{13}$$

This correspondence means that, if U accepts an AMS_Init_RSP message x from V, then the message has been sent by V to U, and U and V share the same signature s_U, random number Rand_V, time t_U, MAC key $\mathit{KMAC}_{\mathsf{U},\mathsf{V}}$, encryption key $\mathit{KENC}_{\mathsf{U},\mathsf{V}}$, and there is a time t_V at which V received the AMS_Init_REQ message such that $\mathsf{check\_time}(t_\mathsf{U}, t_\mathsf{V})$, that is, the inequality (2) holds. Furthermore, since the correspondence is injective (**inj-event**), each AMS_Init_RSP message received by U corresponds to a distinct AMS_Init_RSP message sent by V. So (13) proves authentication of the ground entity to the aircraft.

Hence, these correspondences prove mutual authentication of the aircraft and ground entity.

4) Message authentication. This property is formalized by the following correspondences:

$$\begin{aligned}&\mathbf{inj\text{-}event}(\mathsf{recv\_msg}(x, y, t_\mathsf{U}, s_\mathsf{U}, \mathit{Rand}_\mathsf{V}, \mathit{policy}, \mathit{prot}, \mathit{enc}, \mathit{count}, \mathit{msg})) \implies \\ &\quad \mathit{prot} = \mathsf{get\_policy}(\mathit{policy}, \mathit{msg}) \wedge \\ &\quad (\mathbf{inj\text{-}event}(\mathsf{send\_msg}(y, x, t_\mathsf{U}, s_\mathsf{U}, \mathit{Rand}_\mathsf{V}, \mathit{policy}, \mathit{prot}, \mathit{enc}, \mathit{count}, \mathit{msg})) \vee \mathit{prot} = \mathsf{NONE})\end{aligned} \tag{14}$$

where (x, y) is (U, V) or (V, U). These correspondences mean that, for (x, y) equal to (U, V) or (V, U), if x has received the payload message msg from entity y, protected under the protection mode prot, and the policy identifier is policy, then the protection mode associated to msg in the policy policy is prot, and either $\mathit{prot} = \mathsf{NONE}$ or the message msg has been sent by y to x with the same timestamp t_U, signature s_U, random value Rand_V, policy policy, protection mode prot, encryption algorithm *enc*, and counter *count* on the receiver and sender sides. Furthermore, when $\mathit{prot} \neq \mathsf{NONE}$, the message has been sent at least as many times as it has been received. In other words, authenticity and absence of replays of the message are guaranteed when its associated policy is not NONE.

In ProVerif, since we use the predicate policy_ok instead of the function get_policy (see Section 2.2 *9)*), we prove the following correspondences:

$$\begin{aligned}
&\mathbf{event}(\mathsf{recv\_msg}(x, y, t_{\mathsf{U}}, s_{\mathsf{U}}, \mathit{Rand}_{\mathsf{V}}, \mathit{policy}, \mathit{prot}, \mathit{enc}, \mathit{count}, \mathit{msg})) \Longrightarrow\\
&\quad \mathsf{policy\_ok}(\mathit{policy}, \mathit{prot}, \mathit{msg}) \wedge\\
&\quad \mathsf{policy\_ok}(\mathit{policy}, \mathit{prot}', \mathit{msg}) \wedge\\
&\quad (\mathbf{event}(\mathsf{send\_msg}(y, x, t_{\mathsf{U}}, s_{\mathsf{U}}, \mathit{Rand}_{\mathsf{V}}, \mathit{policy}, \mathit{prot}', \mathit{enc}, \mathit{count}, \mathit{msg})) \vee \mathit{prot} = \mathsf{NONE})
\end{aligned}$$

where (x, y) is (U, V) or (V, U), and we infer (14) by combining them with the following manual argument. Since $\mathsf{policy\_ok}(\mathit{policy}, \mathit{prot}, \mathit{msg})$ holds if and only if $\mathit{prot} = \mathsf{get\_policy}(\mathit{policy}, \mathit{msg})$, we have $\mathit{prot} = \mathit{prot}' = \mathsf{get\_policy}(\mathit{policy}, \mathit{msg})$, so

$$\begin{aligned}
&\mathbf{event}(\mathsf{recv\_msg}(x, y, t_{\mathsf{U}}, s_{\mathsf{U}}, \mathit{Rand}_{\mathsf{V}}, \mathit{policy}, \mathit{prot}, \mathit{enc}, \mathit{count}, \mathit{msg})) \Longrightarrow\\
&\quad \mathit{prot} = \mathsf{get\_policy}(\mathit{policy}, \mathit{msg}) \wedge\\
&\quad (\mathbf{event}(\mathsf{send\_msg}(y, x, t_{\mathsf{U}}, s_{\mathsf{U}}, \mathit{Rand}_{\mathsf{V}}, \mathit{policy}, \mathit{prot}, \mathit{enc}, \mathit{count}, \mathit{msg})) \vee \mathit{prot} = \mathsf{NONE}).
\end{aligned} \tag{15}$$

which is a non-injective variant of (14). We prove the absence of replays as follows. By the message count windowing criteria [2, Table 7.3.1-1, Step D08], the receiver accepts only once a message with a given count. Moreover, x, y identify the receiver and sender, respectively, and s_{U} and $\mathit{Rand}_{\mathsf{V}}$ identify uniquely the session of the aircraft and of the ground entity, respectively. (Since ECDSA signatures are probabilistic, a new signature s_{U} is created at each session.) So distinct executions of $\mathsf{recv\_msg}(x, y, t_{\mathsf{U}}, s_{\mathsf{U}}, \mathit{Rand}_{\mathsf{V}}, \mathit{policy}, \mathit{prot}, \mathit{enc}, \mathit{count}, \mathit{msg})$ (with $\mathit{prot} \neq \mathsf{NONE}$) have distinct values of the quintuple $x, y, s_{\mathsf{U}}, \mathit{Rand}_{\mathsf{V}}, \mathit{count}$. We have (15) and we prove injectivity as in Lemma 1, distinguishing the cases $\mathit{prot} \neq \mathsf{NONE}$ and $\mathit{prot} = \mathsf{NONE}$, so we obtain (14).

Since the receiver accepts a message with a given count only once, message counter roll-over must be avoided by ending the session and starting a new secure session when the counter reaches its maximal value of 255, as mentioned in [2, page 79].

CryptoVerif proves (14) directly.

5) Secrecy of keys. We study the secrecy of the keys $\mathit{KMAC}_{\mathsf{U,V}}$ and $\mathit{KENC}_{\mathsf{U,V}}$. In the symbolic model, ProVerif shows that the adversary cannot compute these keys (by showing that a constant message encrypted under these keys remains secret). In the computational model, we would like to show that these keys are indistinguishable from fresh independent random keys at the end of the session initiation. CryptoVerif proves this property for keys $\mathit{KENC}_{\mathsf{U,V}}$. This property does not hold for $\mathit{KMAC}_{\mathsf{U,V}}$, because this key is used in the second message of the session initiation. This point does not correspond to an attack against the protocol, but it makes it more difficult to prove security modularly: we cannot show that the session initiation provides a secure key, and then use this property in the application data exchange.

6) Secrecy of messages. To prove secrecy in ProVerif, we modify our model slightly so that the payload message is not initially known to the adversary. We define a private function (a function that cannot be applied by the adversary) msg that we use to build the payload message. The term $\mathsf{msg}(\mathsf{U}, \mathsf{V}, \mathit{policy}, x)$ represents a message sent from U to V under the policy *policy*.

When encryption is supported, ProVerif proves the property:

$$\begin{aligned}&\mathit{attacker}(\mathsf{msg}(\mathsf{U},\mathsf{V},\mathit{policy},x)) \Longrightarrow\\ &\quad \mathsf{policy\_ok}(\mathit{policy},\mathsf{AUTH},\mathsf{msg}(\mathsf{U},\mathsf{V},\mathit{policy},x)) \vee\\ &\quad \mathsf{policy\_ok}(\mathit{policy},\mathsf{NONE},\mathsf{msg}(\mathsf{U},\mathsf{V},\mathit{policy},x)).\end{aligned} \tag{16}$$

and the symmetric correspondence obtained by swapping U and V. In other words, if the attacker has the payload message sent from U to V or from V to U, then the policy specifies a protection AUTH or NONE for this message. So, if the policy specifies a protection BOTH, then the attacker cannot have the payload message, so this message is secret.

Obviously, when encryption is not supported, the messages cannot be secret. ProVerif confirms this point.

In CryptoVerif, we suppose that encryption is supported. CryptoVerif then proves the following secrecy property. At the beginning of the protocol, we choose a random bit b with uniform probability. For each payload message, the sender actually gets two candidate payloads $\mathit{payload}_0$ and $\mathit{payload}_1$ from the adversary. When the current session is between the honest entities U and V and the two payloads have protection mode BOTH (that is, $\mathsf{get\_policy}(\mathit{policy},\mathit{payload}_0) = \mathsf{get\_policy}(\mathit{policy},\mathit{payload}_1) = \mathsf{BOTH}$), we require that the corresponding encoded and compressed payloads have the same length. Otherwise, we require that $\mathit{payload}_0 = \mathit{payload}_1$. The sender then sends $\mathit{payload}_b$ following the protocol. We show that the random bit b is secret, that is, the adversary has a probability close to $1/2$ of guessing b. So the adversary only has a negligible probability of distinguishing whether $\mathit{payload}_0$ or $\mathit{payload}_1$ has been sent, which proves secrecy for messages with associated protection mode BOTH in sessions between U and V.

Although the content of the messages is secret, the length of the encoded and compressed payload is public, since encryption preserves the length. In some cases, this length may reveal important information about the underlying message. (See [44] for an attack that relies on the length of encrypted messages in a different context.) From the point of view of security, it would be stronger to hide the length of the messages as well, for instance by arranging so that all messages whose length is within a certain interval encrypt into a ciphertext of the same length, in the style of the countermeasure proposed in [44]. However, a loss of bandwidth is unavoidable in this case.

7) Unknown key share attacks. ProVerif and CryptoVerif prove the following correspondence:

$$\begin{aligned}&\mathbf{event}(\mathsf{send\_Init\_RSP}(V,U,x,s_{\mathsf{U}},\mathit{Rand}_{\mathsf{V}},t_{\mathsf{U}},t_{\mathsf{V}},\mathit{KMAC}_{\mathsf{U},\mathsf{V}},\mathit{KENC}_{\mathsf{U},\mathsf{V}})) \wedge\\ &\mathbf{event}(\mathsf{recv\_Init\_RSP}(U',V',x',s'_{\mathsf{U}},\mathit{Rand}'_{\mathsf{V}},t'_{\mathsf{U}},\mathit{KMAC}'_{\mathsf{U},\mathsf{V}},\mathit{KENC}_{\mathsf{U},\mathsf{V}})) \Longrightarrow U = U' \wedge V = V'.\end{aligned} \tag{17}$$

(An encoding is needed in ProVerif because it does not support conjunctions of events on the left-hand side of the arrow, but the proved correspondence is equivalent to the one above.) This correspondence shows that, if executions of U and V share the same encryption key, then U believes he talks to V and conversely. A similar correspondence holds for executions that share the same MAC key. These correspondences show the absence of unknown key share (UKS) attacks [26,30]. In these attacks, the aircraft U believes it shares a key with a (possibly dishonest) ground entity V' and the ground entity V believes it shares a key with a (possibly dishonest) aircraft U' but the two sessions share the same keys, so the adversary can make a message from U to V' appear to V as a message from U', and similarly swapping U and V. (In bilateral UKS attacks [26], $\mathsf{U} \neq \mathsf{U}'$ and $\mathsf{V} \neq \mathsf{V}'$; in unilateral UKS attacks [30], $\mathsf{U} \neq \mathsf{U}'$ and $\mathsf{V} = \mathsf{V}'$ or symmetrically.)

8) Forward secrecy of messages. Forward secrecy means that, even if a long-term secret key is compromised after the protocol is executed, the exchanged messages remain secret. ProVerif finds an attack against this property. Indeed, if an adversary manages to compromise a secret key d_{U} or d_{V} even after the session has been terminated, it will be able to compute the shared keys and decrypt the messages, because the entities always reuse the same secret keys d_{U} and d_{V}, without including ephemerals in the Diffie-Hellman key agreement. We did not try to prove forward secrecy in CryptoVerif, since we know that it does not hold from the symbolic analysis.

9) Key compromise impersonation. When the long-term key d_{U} of an aircraft is compromised, we naturally expect that the adversary can impersonate the aircraft to any ground entity. However, ProVerif finds out that the adversary can also impersonate any ground entity to the compromised aircraft: it finds an attack against the correspondence (13). This is a key compromise impersonation (KCI) attack. To perform this attack, the adversary runs the protocol exactly as a ground entity, except that it computes $Z_{\mathsf{U},\mathsf{V}}$ as $d_{\mathsf{U}}Q_{\mathsf{V}}$ using the compromised secret key d_{U} of the aircraft and the public key Q_{V} of the ground entity, instead of the normal computation of $Z_{\mathsf{U},\mathsf{V}} = d_{\mathsf{V}}Q_{\mathsf{U}}$. Since $d_{\mathsf{U}}Q_{\mathsf{V}} = d_{\mathsf{V}}Q_{\mathsf{U}}$, the two computations yield the same result. Furthermore, the computation of $Z_{\mathsf{U},\mathsf{V}}$ is the only point at which the secret key d_{V} of the ground entity is normally used, so the adversary succeeds in impersonating the ground entity.

When the long-term key d_{V} of a ground entity is compromised, ProVerif shows that the correspondence (7) still holds. Indeed, the protocol requires a signature of the aircraft, so the adversary cannot fully impersonate an aircraft to the ground entity, because it cannot forge a signature. However, ProVerif also finds out that the correspondence (15) does not hold when (x, y) is (U, V): the ground entity may receive messages that seem to come from the aircraft even though the aircraft never sent them. Indeed, after the first two messages of the protocol, the adversary can compute $Z_{\mathsf{U},\mathsf{V}} = d_{\mathsf{V}}Q_{\mathsf{U}}$, and from that compute the encryption and MAC keys, so it can decrypt and forge messages.

We did not try to prove the absence of key compromise impersonation attacks in CryptoVerif, since we know that it does not hold from the symbolic analysis.

10) Strengthened protocol. Moreover, attacks against these properties are not very serious as they only appear after the compromise of a long-term key, which is already a serious situation. However, they make this compromise even more serious.

These attacks can be avoided by using a signed Diffie-Hellman key exchange in which the long-term keys are used only for signing and Diffie-Hellman exponents are fresh in each session. Hence, we define a fixed protocol as follows:

$$\underline{\textit{Aircraft } \mathsf{U}} \qquad\qquad\qquad\qquad\qquad\qquad \underline{\textit{Ground entity } \mathsf{V}}$$

Secure session initiation

$$\begin{array}{c}
\text{AMS\_Init\_REQ message}\\
m_1 = (H_1, \mathsf{Init\_REQ}, \mathit{Policy}, \mathsf{U}, \mathsf{V}, R_{\mathsf{U}}, \mathit{Algos}, t_{\mathsf{U}})\\
s_{\mathsf{U}} = \mathsf{SIG}(d_{\mathsf{U}}, (\mathsf{U}, \mathsf{V}, m_1))\\
\xrightarrow{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}
\end{array}$$

$$\begin{array}{c}
\text{AMS\_Init\_RSP message}\\
m_2 = (H_2, \mathsf{Init\_RSP+}, \mathit{AlgSel}, R_{\mathsf{V}}, [\mathit{cert}_{\mathsf{V}}], t_{\mathsf{U}}, \mathit{Rand}_{\mathsf{V}})\\
\mathsf{SIG}(d_{\mathsf{V}}, (\mathsf{V}, \mathsf{U}, m_2, m_1, s_{\mathsf{U}}))\\
\xleftarrow{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}
\end{array}$$

where $e_{\mathsf{U}}, R_{\mathsf{U}}$ is a fresh ephemeral private-public key pair for the aircraft and $e_{\mathsf{V}}, R_{\mathsf{V}}$ is a fresh ephemeral private-public key pair for the ground entity, and we compute

$$\begin{aligned}
Z_{\mathsf{U},\mathsf{V}} &= e_{\mathsf{V}}R_{\mathsf{U}} = e_{\mathsf{U}}R_{\mathsf{V}}\\
X_{\mathsf{U},\mathsf{V}} &= \mathsf{HASH}((\mathit{AlgSel}, \mathit{Algos}, t_{\mathsf{U}}, s_{\mathsf{U}}, \mathit{Rand}_{\mathsf{V}}))\,.
\end{aligned}$$

The inclusion of m_1 in the signature of the AMS_Init_RSP message guarantees that U and V have common parameters (in particular the ephemeral R_{U} and the algorithms *Algos*) even when U's long-term key is compromised. The addition of *AlgSel* in the computation of $X_{\mathsf{U},\mathsf{V}}$ guarantees that U and V share the same algorithms even when V's long-term key is compromised. We could not fully verify the fixed protocol in ProVerif because it took more than 8 Gb of memory. Indeed, the protocols verified in this paper are fairly intensive for ProVerif, because there are many cases. The fixed public key protocol is even more, probably because of the presence of additional randomness (the ephemerals), which leads to bigger terms. However, we proved using CryptoVerif that it satisfies all properties we consider in the computational model, which is stronger than a symbolic proof in ProVerif. The assumptions are the same as for the original protocol, except that we do not need a joint security assumption between Diffie-Hellman key agreements and signatures: we use SUF-CMA signatures and the computational Diffie-Hellman assumption (for most properties) or the stronger gap Diffie-Hellman assumption (for resistance to KCI attacks when the ground entity is compromised and for forward secrecy; these properties do not hold under the CDH assumption). Obviously, the stronger security properties come at a price: the key exchange of the fixed protocol is more costly than the original one, since U and V have to choose an ephemeral secret key and compute and send the corresponding ephemeral public key, and a MAC is replaced with a signature.

Another possible way to strengthen the protocol would be to use a triple Diffie-Hellman exchange, with ephemerals and without signatures, as in Signal [32], for instance. We preferred using a signed Diffie-Hellman exchange to be closer to the original protocol: the signatures authenticate the AMS_Init_IND and AMS_Init_REQ messages, as in the original protocol; that would not be true with a triple Diffie-Hellman exchange.

Honeywell made several comments on our proposal:

- The AMS security protocol is based on the technical provisions and security protocol specified for the Aeronautical Telecommunications Network (ATN). Although the ATN does not currently include provisions for confidentiality (encryption), the AMS encryption implementation is consistent with a white paper [25] that was presented to ICAO to demonstrate extensibility of the ATN security solution to support encryption in the future. The approach presented in [25] does not consider use of ephemerals.

- The current AMS security protocol is optimized to take into account the bandwidth, computational, and memory constraints of avionic systems.

- Our proposed use of ephemerals would require further assessment in the following areas: the ability of legacy avionics to generate quality ephemeral key pairs; the impact of increased overhead to exchange ephemeral public components during secure session initiation; and the impact on avionics computing performance and memory as a result of more complex computation of the secret key agreement value ($Z_{\mathsf{U},\mathsf{V}}$).

- With the current AMS security solution, session-specific encryption and authentication keys are never stored but may be recreated using archived copies of the session initiation messages plus the static public/private key pair of the ground entity. This may be necessary to decrypt archived encrypted messages in the event of an accident investigation. The addition of ephemeral keys would require the ground entity to also archive its public/private ephemerals (consistent with message retention policy) associated with each secure session.

11) Proof indications. In CryptoVerif, the proofs of the AMS public-key protocol require the user to give the main proof steps: apply the assumption on the signatures in certificates, use the random oracle assumption on the concatenation of $\mathsf{KDF256}$ and $\mathsf{KDF128}$, remove assignments on

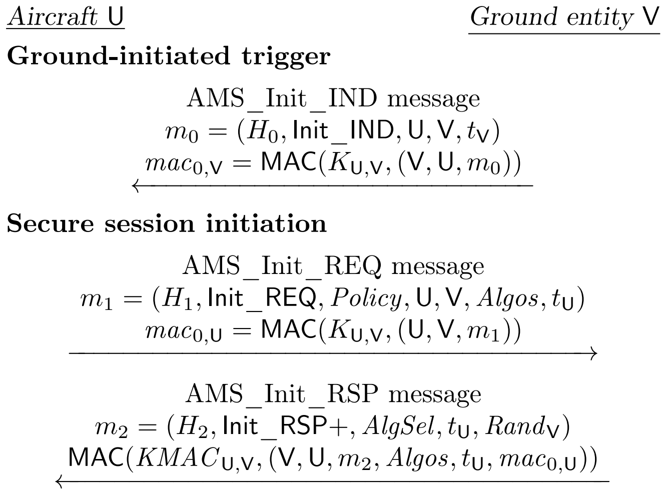
Aircraft U
Ground entity V
Ground-initiated trigger
AMS_Init_IND message
$m_0 = (H_0, \mathsf{Init\_IND}, \mathsf{U}, \mathsf{V}, t_\mathsf{V})$
$mac_{0,\mathsf{V}} = \mathsf{MAC}(K_{\mathsf{U},\mathsf{V}}, (\mathsf{V}, \mathsf{U}, m_0))$
Secure session initiation
AMS_Init_REQ message
$m_1 = (H_1, \mathsf{Init\_REQ}, \mathit{Policy}, \mathsf{U}, \mathsf{V}, \mathit{Algos}, t_\mathsf{U})$
$mac_{0,\mathsf{U}} = \mathsf{MAC}(K_{\mathsf{U},\mathsf{V}}, (\mathsf{U}, \mathsf{V}, m_1))$
AMS_Init_RSP message
$m_2 = (H_2, \mathsf{Init\_RSP+}, \mathit{AlgSel}, t_\mathsf{U}, \mathit{Rand}_\mathsf{V})$
$\mathsf{MAC}(\mathit{KMAC}_{\mathsf{U},\mathsf{V}}, (\mathsf{V}, \mathsf{U}, m_2, \mathit{Algos}, t_\mathsf{U}, mac_{0,\mathsf{U}}))$

Figure 2: Simplified AMS protocol using a shared key

variable $Z_{\mathsf{U},\mathsf{V}}$ in the aircraft process and in the ground entity process to make equality tests such as $m = d_\mathsf{V} Q_\mathsf{U}$ appear, use the gap Diffie-Hellman assumption to prove that some of these tests are false, apply the assumption on signatures under d_V and d_U, and finally use the automatic proof strategy to conclude the proof. The proofs for the strengthened protocol use a similar style of guidance, sometimes with additional case distinctions. We refer the interested reader to the CryptoVerif scripts for details.

12) Runtime. In total, the analysis of the public-key protocol took 10 h 47 min in ProVerif and 17 min in CryptoVerif. The analysis of the strengthened protocol exhausted memory in ProVerif and took 1 h 16 min in CryptoVerif on an Intel Xeon 1.2 GHz. In this example, CryptoVerif is more efficient than ProVerif, probably because ProVerif explores an abstraction of all states of the protocol, while CryptoVerif performs a proof by game transformation, so it does not need to explore all these states. The guidance also helps CryptoVerif to be more efficient by avoiding the exploration of unsuccessful proofs.

3 AMS Protocol using a Shared Key

In this section, we study the AMS protocol using a shared key as defined in [2, Attachment 8].

3.1 Protocol Description

In this protocol, the aircraft and the ground entity initially share a long-term key $K_{\mathsf{U},\mathsf{V}}$. They establish session keys as shown in Figure 2. The protocol is similar to the public-key protocol (see Section 2.1), but the AMS_Init_IND and AMS_Init_REQ messages are protected by MACs under $K_{\mathsf{U},\mathsf{V}}$ instead of signatures. The session keys are derived as follows:

$$X_{\mathsf{U},\mathsf{V}} = \mathsf{HASH}((\mathit{Algos}, t_\mathsf{U}, mac_{0,\mathsf{U}}, \mathit{Rand}_\mathsf{V}))$$

and $\mathit{KMAC}_{\mathsf{U},\mathsf{V}}$ and $\mathit{KENC}_{\mathsf{U},\mathsf{V}}$ are computed as in the public-key protocol except that $Z_{\mathsf{U},\mathsf{V}}$ is replaced with $K_{\mathsf{U},\mathsf{V}}$. The application data exchange proceeds exactly as in the public-key protocol.

3.2 Modeling Notes

Our comments of Section 2.2 also apply to the shared-key protocol, with the following exceptions:

1) Sessions. Except for unknown key share attacks, the models consider a single aircraft U and a single ground entity V that share a key $K_{\mathsf{U},\mathsf{V}}$ and talk to each other. All other executions of the protocol are considered as part of the adversary, since they use a different shared key. For unknown key share attacks, U and V may also talk to dishonest participants, as for the public-key protocol.

2) Certificates. Section 2.2 *3)* does not apply, since this protocol does not use certificates.

3) Replay protection. The shared-key protocol uses a replay check similar to the one used by the public-key protocol, and it suffers from the same weakness. However, the correction is slightly different, because in the shared-key protocol, the AMS_Init_IND and AMS_Init_REQ messages do not contain anything random. (In the public-key protocol, they contain a randomized signature.) Since the timestamp has a 2-second resolution, AMS_Init_IND or AMS_Init_REQ messages sent within the 2-second interval in which the timestamp does not change are equal. Hence, the receiver cannot distinguish a replay from a legitimate attempt at starting a session again within that interval.

To solve this problem, we assume that the aircraft never sends two AMS_Init_REQ messages to the same ground entity V with the same timestamp t_{U}. (Using a random nonce in addition to the timestamp t_{U} would guarantee that two sessions never start with the same AMS_Init_REQ message. However, it is an incompatible change in the protocol.)

Using this assumption, for each aircraft U, the ground entity should remember the timestamp t_{U} of the last AMS_Init_REQ message received from U, as long as $t - 120$ seconds $\leq t_{\mathsf{U}}$ where t is the ground entity's local time. If the timestamp t_{U} of a new AMS_Init_REQ message received from U is not strictly more recent than the remembered timestamp t_{U}, the message should be discarded. (Replays of AMS_Init_REQ messages older than 120 seconds are already discarded by checking the timestamp.)

Moreover, as for the public-key protocol, the original protocol has no protection against replays of a ground-initiated trigger (AMS_Init_IND message). To avoid replay attacks, we recommend protecting against replays of the AMS_Init_IND message by making sure that the ground entity never sends two AMS_Init_IND messages with the same timestamp t_{V} and using a check similar to the one mentioned above for the AMS_Init_REQ message.

3.3 Security Assumptions on Primitives

The shared secret key $K_{\mathsf{U},\mathsf{V}}$ is used as key both for the MAC and for the key derivation function KDF. Using the same key for different cryptographic schemes is problematic for proving the protocol in the computational model: one cannot prove that the protocol is secure under standard security assumptions, namely that the MAC is SUF-CMA and the key derivation function is a pseudo-random function (PRF). In this case, one needs a combined security assumption about the two schemes. (A correction to this problem would be to use the key derivation function KDF also to derive the key of the MAC in the AMS_Init_IND and AMS_Init_REQ messages. However, this correction would alter the protocol in an incompatible way.)

To study the protocol in CryptoVerif, we prove that HMAC-SHA256 and the two key derivation functions $\mathsf{KDF256}$ and $\mathsf{KDF128}$ are indistinguishable from independent pseudo-random functions, even when they use the same key k, in the following two situations:

- when the compression function of SHA256 is a random oracle;
- when the compression function of SHA256 is a pseudo-random function and, given a 256-bit

Protocols	O	ST	RP
Freshness of AMS_Init_REQ (7)	Y	Y	Y
Freshness of AMS_Init_IND (8)	Y	Y	Y
No replay for AMS_Init_REQ (11)			Y
No replay for AMS_Init_IND (12)			Y
Non-injective authentication of V to U	Y	Y	Y
Injective authentication of V to U (13)		Y	Y
Non-injective message authentication (15)	Y	Y	Y
Injective message authentication (14)		Y	Y
Secrecy of $KMAC_{\mathsf{U},\mathsf{V}}$	PV	PV	PV
Secrecy of $KENC_{\mathsf{U},\mathsf{V}}$	PV	Y	Y
Secrecy of messages	PV	Y	Y
No unknown key share attacks (17)	Y	Y	Y

Figure 3: Analysis of the shared-key protocol

random key k,

$$\mathsf{SHA256\_comp}(\mathsf{SHA256\_init}, k \oplus \mathsf{opad}) \text{ and}$$
$$\mathsf{SHA256\_comp}(\mathsf{SHA256\_init}, k \oplus \mathsf{ipad})$$

are independent pseudo-random bitstrings, and the function

$$x \mapsto \mathsf{SHA256\_comp}(\mathsf{SHA256\_init}, k \| x)$$

is a pseudo-random function independent from those two bitstrings. ($\mathsf{SHA256\_comp}$ is the compression function of SHA256, $\mathsf{SHA256\_init}$ is the initial value for SHA256 given in [41, Section 5.3.2], $\oplus$ is exclusive or, opad and ipad are distinct fixed 512-bits constants consisting of 64 copies of the bytes 0x5C and 0x36 respectively [42], x is 256-bit long, $\|$ is bitstring concatenation.)

The first assumption is very strong, but standard. The second assumption is not so strong but less standard. It still seems reasonable. The second result generalizes the proof that HMAC is a PRF given in [9]. These two results are proved using CryptoVerif, as detailed in Appendix D.

Knowing that HMAC-SHA256, $\mathsf{KDF256}$, and $\mathsf{KDF128}$ are independent PRFs (even when they use the same key), we show that

- HMAC-SHA256 is SUF-CMA, and $\mathsf{KDF256}$ and $\mathsf{KDF128}$ are independent PRFs (even when they use the same key). This result generalizes the well-known result that a PRF is an SUF-CMA MAC [10, Section 2.4]. It is proved using CryptoVerif as detailed in Appendix E.
- HMAC-SHA256 has the following collision-resistance property: when the key k is used only for MAC queries, $\mathsf{MAC}(k, m_1) = \mathsf{MAC}(k, m_2)$ implies $m_1 = m_2$ with overwhelming probability, where MAC is HMAC-SHA256. This result is proved using CryptoVerif as detailed in Appendix F.
- The truncation of HMAC-SHA256 (to 32, 64, and 128 bits, as used in the protocol) is also an SUF-CMA MAC. This result holds because the truncation of a PRF is also a PRF, and a PRF is an SUF-CMA MAC.

We use these properties in the proof of protocol. We also assume that SHA256 is collision-resistant and we make the same assumption on the AES128-CFB128 encryption scheme as in the public-key protocol.

3.4 Security Analysis

Much like for the public-key protocol, we use ProVerif and CryptoVerif to verify the security properties of the protocol. For this protocol, CryptoVerif concludes without user guidance. Our results are summarized in Figure 3. We refer to correspondences used for the public-key protocol in this figure; still, the correspondences used for the shared-key protocol differ in that the arguments s_{U} and s_{V} are omitted from events and in (7), (8), (11), and (12), the argument x contains the AMS_Init_IND or AMS_Init_REQ message including the MAC.

The authentication of U to V does not appear in the figure as it corresponds to the freshness and absence of replays for AMS_Init_REQ. The non-injective authentication of V to U is modeled by a non-injective variant of (13). A shared-key protocol cannot provide forward-secrecy nor protect against key compromise impersonation: the compromise of $K_{\mathsf{U},\mathsf{V}}$ allows the adversary to compute all keys, decrypt all messages, and impersonate participants from both sides. So we do not test these properties.

In Figure 3, we consider three variants of the protocol: the original protocol (O), the protocol with the additional assumption that the aircraft never sends two AMS_Init_REQ messages with the same V and t_{U} (ST, for “Single Timestamp”), and the protocol with our improved replay protection (RP). We write Y (Yes) when the property is proved both by ProVerif and CryptoVerif (using manual arguments for proving injectivity in ProVerif as for the public-key protocol), and PV when it is proved only by ProVerif.

To obtain absence of unknown key share attacks in CryptoVerif, we had to assume that $\mathsf{KDF256}$ and $\mathsf{KDF128}$ are independent random oracles, as for the public-key protocol. In the computational model, $\mathit{KMAC}_{\mathsf{U},\mathsf{V}}$ is not secret, that is, it is distinguishable from random, because it is used in the AMS_Init_RSP message, as in the public-key protocol.

All other properties that do not hold are due to replay attacks. As discussed in Section 3.2 *3)*, absence of replays for AMS_Init_REQ and AMS_Init_IND messages holds only with our improved protection against replays. Furthermore, consider the following scenario shown in Figure 4. The aircraft U and the ground entity V first run a secure session. In the 2-second interval during which the timestamp remains the same, the aircraft starts a new secure session with V with the same t_{U}, so exactly the same AMS_Init_REQ message. (This scenario is not entirely unrealistic since, in case the first session has been triggered by an AMS_Init_IND message from the ground entity, the adversary can force such a secure session restart by replaying the AMS_Init_IND message as mentioned in Section 2.2 *6).*) The adversary intercepts this AMS_Init_REQ message and replies by replaying the AMS_Init_RSP message of the first session. This message is accepted by the aircraft, so that the aircraft executes event $\mathsf{recv\_Init\_RSP}$ twice, while the ground entity executes event $\mathsf{send\_Init\_RSP}$ once. This point breaks injective authentication of V to U. The established sessions keys are the same in both sessions, which breaks secrecy of $\mathit{KENC}_{\mathsf{U},\mathsf{V}}$ in the computational model: indeed, the keys $\mathit{KENC}_{\mathsf{U},\mathsf{V}}$ are each indistinguishable from random, but they are not independent of each other, which we require for computational secrecy. Furthermore, since the keys are equal, application data messages of the first session can be replayed in the second one, which breaks injective message authentication. Finally, if the aircraft sends encrypted data in this second session, the adversary can break secrecy of messages in the computational model, as follows.

In both sessions, for the n-th message sent by the aircraft, the IV data is $\mathit{IVdata}(n) = (\text{0x00}\|n\|\text{zero-pad-bits})$, so the IV is also the same in both sessions:

$$\mathit{IV}(n) = \mathsf{E}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, 0; \mathit{IVdata}(n))\,.$$

Suppose that the k-th block of plaintext (that is, encoded and compressed payload) of the n-th message sent by the aircraft in the first session is P_k, and in the second session P'_k. Similarly, let

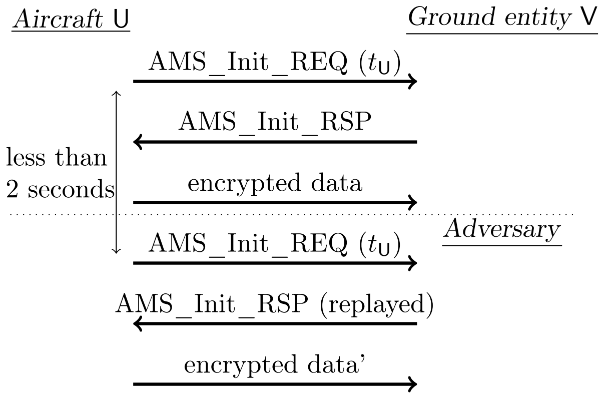

Figure 4: Replay attack

C_k be the k-th block of ciphertext of the n-th message sent by the aircraft in the first session, and C'_k in the second session. By definition of AES128-CFB128,

$$\begin{aligned} C_1 &= P_1 \oplus \mathsf{AES128}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, \mathit{IV}(n)) \text{ and} \\ C'_1 &= P'_1 \oplus \mathsf{AES128}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, \mathit{IV}(n)) \end{aligned}$$

so $C_1 \oplus C'_1 = P_1 \oplus P'_1$. The adversary knows C_1 and C'_1, so it can compute $C_1 \oplus C'_1$ and thus obtain $P_1 \oplus P'_1$. This is already a breach of secrecy, as the adversary obtains the exclusive or of two blocks of plaintext. In case $C_1 \neq C'_1$, that is, $P_1 \oplus P'_1 \neq 0$, the attack stops here because

$$\begin{aligned} C_2 &= P_2 \oplus \mathsf{AES128}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, C_1) \\ C'_2 &= P'_2 \oplus \mathsf{AES128}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, C'_1) \end{aligned}$$

so P_2 and P'_2 are xor-ed with different encryptions. However, in case $C_1 = C'_1$, that is, $P_1 \oplus P'_1 = 0$, the attack continues until a block differs: if for all $j < k$, $P_j = P'_j$, then for all $j < k$, $C_j = C'_j$ by an easy induction on j since $C_j = P_j \oplus \mathsf{AES128}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, C_{j-1}) = P'_j \oplus \mathsf{AES128}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, C'_{j-1}) = C'_j$, and

$$\begin{aligned} C_k &= P_k \oplus \mathsf{AES128}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, C_{k-1}) \\ C'_k &= P'_k \oplus \mathsf{AES128}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, C'_{k-1}) \\ &= P'_k \oplus \mathsf{AES128}(\mathit{KENC}_{\mathsf{U},\mathsf{V}}, C_{k-1}) \end{aligned}$$

so $C_k \oplus C'_k = P_k \oplus P'_k$, which can be computed by the adversary. Hence, the adversary can test whether the n-th encrypted message sent by the aircraft in the second session is the same as the corresponding message of the first session, and in case these messages differ, it gets the exclusive or of the first blocks of plaintext that differ. This attack does not appear in the symbolic model, both because the symbolic model ignores the details of the encryption algorithm and because we verified a weaker secrecy property: we showed that the adversary cannot recover the plaintext.

Therefore, replays may have more serious consequences in the shared-key protocol than in the public-key protocol. The analysis of the shared-key protocol took 8 min in ProVerif and 11 min in CryptoVerif on an Intel Xeon 1.2 GHz.

4 Conclusion

Our study confirms the main security properties of the public-key protocol: authentication and secrecy of the messages (provided messages are not sent in the clear due to the failure of secure session initiation). It leads to several recommendations for changes in the ARINC823 standard: fix the timestamp verification formula (1); make explicit the treatment of the MAC length received by the aircraft; make explicit the policy check for received messages; discuss the security risks associated to sending messages in the clear when the secure session initiation fails, and their possible mitigations; strengthen the replay protection. The last point is especially important for the shared-key protocol, in which replays may lead to secrecy breaches and replays of application data messages. We designed a strengthened public-key protocol, which satisfies forward secrecy, protects against key compromise impersonation attacks, and avoids sharing keys between several cryptographic primitives.

This case study also illustrates the complementarity between symbolic and computational verification. ProVerif allows us to easily find most attacks, but the attack against secrecy in the shared-key protocol does not appear at the symbolic level, and was discovered due to the failure of a proof in CryptoVerif. The computational analysis provides stronger security guarantees, but is also more difficult. The high level of automation of CryptoVerif was very helpful for that analysis, even if a bit of guidance was needed for the proofs of the public-key protocol. CryptoVerif came to the rescue in cases that exceeded ProVerif's capabilities (dealing with some equations; analyzing our strengthened protocol).

Acknowledgments

We thank Honeywell for their very helpful feedback on our analysis. This work was done in the frame of a collaboration with Airbus. It was partly supported by ANR project AnaStaSec (decision number ANR-14-CE28-0014-01).

References

[1] M. Aizatulin, A. D. Gordon, and J. Jürjens. Computational verification of C protocol implementations by symbolic execution. In *19th ACM Conference on Computer and Communications Security (CCS'12)*, pages 712–723. ACM Press, Oct. 2012.

[2] ARINC specification 823P1: Datalink security, part 1 – ACARS message security, Dec. 2007. Available for sale at `http://aviation-ia.com/cf/store/catalog_detail.cfm?item_id=872`.

[3] A. Armando et al. The AVISPA tool for automated validation of Internet security protocols and applications. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005*, volume 3576 of *LNCS*, pages 281–285. Springer, July 2005.

[4] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub. EasyCrypt: A tutorial. In A. Aldini, J. Lopez, and F. Martinelli, editors, *Foundations of Security Analysis and Design VII*, volume 8604 of *LNCS*, pages 146–166. Springer, 2014.

[5] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Z. Béguelin. Probabilistic relational verification for cryptographic implementations. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–206. ACM, 2014.

[6] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In P. Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, Aug. 2011.

[7] G. Barthe, B. Grégoire, and S. Zanella. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 90–101. ACM, Jan. 2009.

[8] S. Z. Béguelin, B. Grégoire, G. Barthe, and F. Olmedo. Formally certifying the security of digital signature schemes. In *30th IEEE Symposium on Security and Privacy, S&P 2009*, pages 237–250. IEEE, May 2009.

[9] M. Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In C. Dwork, editor, *CRYPTO'06*, volume 4117 of *LNCS*, pages 602–619. Springer, Aug. 2006. Extended version available at `http://cseweb.ucsd.edu/~mihir/papers/hmac-new.pdf`.

[10] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, Dec. 2000.

[11] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Computer and Communications Security (CCS'93)*, pages 62–73. ACM Press, 1993.

[12] J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems*, 33(2), 2011.

[13] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the tls 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P'17)*. IEEE, May 2017. To appear.

[14] K. Bhargavan, R. Corin, C. Fournet, and E. Zălinescu. Cryptographically verified implementations for TLS. In *15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 459–468. ACM, Oct. 2008.

[15] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security & Privacy*, pages 445–462, 2013.

[16] B. Blanchet. Proverif users. Available at `http://prosecco.inria.fr/personal/bblanche/proverif/proverif-users.html`.

[17] B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 97–111. IEEE, July 2007. Extended version available as ePrint Report 2007/128, `http://eprint.iacr.org/2007/128`.

[18] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, Oct.–Dec. 2008.

[19] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, July 2009.

[20] B. Blanchet. Automatically verified mechanized proof of one-encryption key exchange. In *25th IEEE Computer Security Foundations Symposium (CSF'12)*, pages 325–339. IEEE, June 2012.

[21] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, Oct. 2016.

[22] B. Blanchet, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Computationally sound mechanized proofs for basic and public-key Kerberos. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 87–99. ACM, Mar. 2008.

[23] D. Boneh, E. Shen, and B. Waters. Strongly unforgeable signatures based on computational Diffie-Hellman. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography - PKC 2006*, volume 3958 of *LNCS*, pages 229–240. Springer, Apr. 2006.

[24] D. Cadé and B. Blanchet. From computationally-proved protocol specifications to implementations and application to SSH. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1):4–31, Mar. 2013.

[25] Application security solution for the aeronautical telecommunications network with encryption support. Document ATNP SB3W0411: Certicom presentation to ICAO ATN Panel, Working Group B, Sub-group B3, Jan. 2002.

[26] L. Chen and Q. Tang. Bilateral unknown key-share attacks in key agreement protocols. *Journal of Universal Computer Science*, 14(3):416–440, Feb. 2008.

[27] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. In *Advances in Cryptology—CRYPTO 2005*, volume 3621 of *LNCS*, pages 430–448. Springer, 2005.

[28] C. J. Cremers. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In *15th ACM conference on Computer and Communications Security (CCS'08)*, pages 119–128. ACM, 2008.

[29] I. B. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology–CRYPTO'89 Proceedings*, volume 435 of *LNCS*, pages 416–427. Springer, Aug. 1989.

[30] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[31] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, Mar. 1983.

[32] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz. How secure is TextSecure? In *IEEE European Symposium on Security and Privacy (Euro S&P)*, pages 457–472, 2016.

[33] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptative chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, Apr. 1988.

[34] ICAO doc 9880 part IV: Manual on detailed technical specifications for the Aeronautical Telecommunication Network (ATN) using ISO/OSI standards and protocols, part IV — directory services, security and systems management, 2010. First edition. International Civil Aviation Organization.

[35] ICAO doc 9896: Manual on the Aeronautical Telecommunication Network (ATN) using Internet Protocol Suite (IPS) standards and protocols, 2010. First edition. International Civil Aviation Organization.

[36] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*, pages 435–450. IEEE, Apr. 2017.

[37] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.

[38] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[39] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, Dec. 1978.

[40] NIST. SP 800-38A: Recommendation for block cipher modes of operation, methods and techniques. Available at `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`, 2001.

[41] NIST. FIPS 180-2: Secure hash standard. Available at `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`, Aug. 2002.

[42] NIST. FIPS 198: The keyed-hash message authentication code (HMAC). Available at `http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf`, July 2008.

[43] T. Okamoto and D. Pointcheval. The gap-problems: a new class of problems for the security of cryptographic schemes. In K. Kim, editor, *International Workshop on Practice and Theory in Public Key Cryptography (PKC'2001)*, volume 1992 of *LNCS*, pages 104–118. Springer, Feb. 2001.

[44] A. Pironti, P.-Y. Strub, and K. Bhargavan. Identifying website users by TLS traffic analysis: New attacks and effective countermeasures. Technical Report 8067, INRIA Paris Rocquencourt, 2012. Available at `https://hal.inria.fr/hal-00732449/`.

[45] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *25th IEEE Computer Security Foundations Symposium (CSF'12)*, pages 78–94. IEEE Computer Society, June 2012.

[46] J.-K. Tsay and S. F. Mjølsnes. A vulnerability in the UMTS and LTE authentication and key agreement protocols. In I. Kotenko and V. Skormin, editors, *Computer Network Security (MMM-ACNS'12)*, volume 7531 of *LNCS*, pages 65–76. Springer, Oct. 2012.

[47] P. Wuille. Dealing with malleability. `https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki`, Mar. 2014.

[48] A. C. Yao. Theory and applications of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (FOCS'82)*, pages 80–91. IEEE Computer Society, Nov. 1982.

$$
\begin{array}{l}
!^{na}\mathbf{new}\ a : Z;(\\
\quad \mathsf{OA}() := \mathsf{exp}(g, a),\\
\quad !^{naDDH}\mathsf{ODDHa}(m : G, j \leq nb) := m = \mathsf{exp}(g, \mathsf{mult}(b[j], a))),\\
!^{nb}\mathbf{new}\ b : Z;(\\
\quad \mathsf{OB}() := \mathsf{exp}(g, b),\\
\quad !^{nbDDH}\mathsf{ODDHb}(m : G, j \leq na) := m = \mathsf{exp}(g, \mathsf{mult}(a[j], b)))\\
\approx\\
!^{na}\mathbf{new}\ a : Z;(\\
\quad \mathsf{OA}() := \mathsf{exp}(g, a),\\
\quad !^{naDDH}\mathsf{ODDHa}(m : G, j \leq nb) := \mathsf{false}),\\
!^{nb}\mathbf{new}\ b : Z;(\\
\quad \mathsf{OB}() := \mathsf{exp}(g, b),\\
\quad !^{nbDDH}\mathsf{ODDHb}(m : G, j \leq na) := \mathsf{false}).
\end{array}
$$

Figure 5: Formalization of the computational Diffie-Hellman (CDH) assumption

Appendices

A Joint Assumption on Diffie-Hellman and Signatures

CryptoVerif provides a fairly flexible language for modeling security assumptions on cryptographic primitives, by giving indistinguishability axioms $L \approx R$, which mean that L is indistinguishable from R. CryptoVerif uses these axioms to provide a game transformation: it replaces the left-hand side L with the right-hand side R.

In this appendix, we explain the formalization of the joint security assumption of gap Diffie-Hellman and SUF-CMA signatures in this language. (The case of Diffie-Hellman and UF-CMA signatures is similar.) We start by recalling the formalization of the computational Diffie-Hellman assumption, given in Figure 5. In this formalization, Z is the set of exponents (that is, $\mathbb{Z}_p$) and G is the Diffie-Hellman group. The function exp represents exponentiation: $\mathsf{exp}(g, a)$ stands for g^a, for $g \in G$ and $a \in Z$. The function mult is the product of exponents. The construct $!^n Q$ represents n copies of Q, and **new** $a : Z$ chooses a fresh random number a uniformly in Z. In the left-hand side, we generate na random exponents a and nb random exponents b, and we provide several oracles: OA returns the public Diffie-Hellman value g^a, $\mathsf{ODDHa}(m, j)$ can be called $naDDH$ times and returns true if and only if $m = g^{b[j].a}$ ($b[j]$ denotes the j-th exponent b); OB and ODDHb are similar for b. By the CDH assumption, the adversary cannot compute $g^{a.b}$ for any of the exponents a and b (except in cases of negligible probability), so it cannot find m such that $m = g^{a.b}$, hence the oracles ODDHa and ODDHb return false. We express this property by replacing the result of ODDHa and ODDHb with false in the right-hand side of the indistinguishability property. The rest is left unchanged.

The GDH assumption says that CDH still holds in case the adversary has access to a decisional Diffie-Hellman oracle $\mathsf{DDH}(G, A, B, C)$ that returns true if and only if there exist a and b such that $A = G^a$, $B = G^b$, $C = G^{ab}$. We formalize the GDH assumption (Figure 6, ignoring the oracles Osigna, Osigna', Osignb, Osignb') by additionally giving such oracles to the

$$
\begin{array}{l}
!^{na}\mathbf{new}\ a : Z; (\\
\quad \mathsf{OA}() := \mathsf{exp}(g, a),\\
\quad !^{naDDH1}\mathsf{ODDHa1}(m : G, m' : G) := m = \mathsf{exp}(m', a),\\
\quad !^{naDDH2}\mathsf{ODDHa2}(m : G, m' : G, j \leq nb) := \mathsf{exp}(m, b[j]) = \mathsf{exp}(m', a),\\
\quad !^{naDDH3}\mathsf{ODDHa3}(m : G, m' : G, j \leq na) := \mathsf{exp}(m, a[j]) = \mathsf{exp}(m', a),\\
\quad !^{naDDH}\mathsf{ODDHa}(m : G, j \leq nb) := m = \mathsf{exp}(g, \mathsf{mult}(b[j], a)),\\
\quad !^{naS}\mathbf{new}\ r : \mathsf{seed}; \mathsf{Osigna}(m : \mathsf{bitstring}) := \mathsf{sign}(a, m, r),\\
\quad !^{naS'}\mathbf{new}\ r : \mathsf{seed}; \mathsf{Osigna}'(m : \mathsf{bitstring}) := \mathsf{sign}'(a, m, r)),\\
!^{nb}\mathbf{new}\ b : Z; (\\
\quad \mathsf{OB}() := \mathsf{exp}(g, b),\\
\quad !^{nbDDH1}\mathsf{ODDHb1}(m : G, m' : G) := m = \mathsf{exp}(m', b),\\
\quad !^{nbDDH2}\mathsf{ODDHb2}(m : G, m' : G, j \leq nb) := \mathsf{exp}(m, b[j]) = \mathsf{exp}(m', b),\\
\quad !^{nbDDH3}\mathsf{ODDHb3}(m : G, m' : G, j \leq na) := \mathsf{exp}(m, a[j]) = \mathsf{exp}(m', b),\\
\quad !^{nbDDH}\mathsf{ODDHb}(m : G, j \leq na) := m = \mathsf{exp}(g, \mathsf{mult}(a[j], b)),\\
\quad !^{nbS}\mathbf{new}\ r : \mathsf{seed}; \mathsf{Osignb}(m : \mathsf{bitstring}) := \mathsf{sign}(b, m, r),\\
\quad !^{nbS'}\mathbf{new}\ r : \mathsf{seed}; \mathsf{Osignb}'(m : \mathsf{bitstring}) := \mathsf{sign}'(b, m, r))\\
\ \approx\\
!^{na}\mathbf{new}\ a : Z; (\\
\quad \mathsf{OA}() := \mathsf{exp}(g, a),\\
\quad !^{naDDH1}\mathsf{ODDHa1}(m : G, m' : G) := m = \mathsf{exp}(m', a),\\
\quad !^{naDDH2}\mathsf{ODDHa2}(m : G, m' : G, j \leq nb) := \mathsf{exp}(m, b[j]) = \mathsf{exp}(m', a),\\
\quad !^{naDDH3}\mathsf{ODDHa3}(m : G, m' : G, j \leq na) := \mathsf{exp}(m, a[j]) = \mathsf{exp}(m', a),\\
\quad !^{naDDH}\mathsf{ODDHa}(m : G, j \leq nb) := \mathsf{false},\\
\quad !^{naS}\mathbf{new}\ r : \mathsf{seed}; \mathsf{Osigna}(m : \mathsf{bitstring}) := \mathsf{sign}(a, m, r),\\
\quad !^{naS'}\mathbf{new}\ r : \mathsf{seed}; \mathsf{Osigna}'(m : \mathsf{bitstring}) := \mathsf{sign}'(a, m, r)),\\
!^{nb}\mathbf{new}\ b : Z; (\\
\quad \mathsf{OB}() := \mathsf{exp}(g, b),\\
\quad !^{nbDDH1}\mathsf{ODDHb1}(m : G, m' : G) := m = \mathsf{exp}(m', b),\\
\quad !^{nbDDH2}\mathsf{ODDHb2}(m : G, m' : G, j \leq nb) := \mathsf{exp}(m, b[j]) = \mathsf{exp}(m', b),\\
\quad !^{nbDDH3}\mathsf{ODDHb3}(m : G, m' : G, j \leq na) := \mathsf{exp}(m, a[j]) = \mathsf{exp}(m', b),\\
\quad !^{nbDDH}\mathsf{ODDHb}(m : G, j \leq na) := \mathsf{false},\\
\quad !^{nbS}\mathbf{new}\ r : \mathsf{seed}; \mathsf{Osignb}(m : \mathsf{bitstring}) := \mathsf{sign}(b, m, r),\\
\quad !^{nbS'}\mathbf{new}\ r : \mathsf{seed}; \mathsf{Osignb}'(m : \mathsf{bitstring}) := \mathsf{sign}'(b, m, r)).
\end{array}
$$

Figure 6: Formalization of the gap Diffie-Hellman (GDH) assumption, with additional signature oracles

```
$!^{N3}\mathbf{new}\ r : Z; (\mathsf{Opk}() := \mathsf{exp}(g, r),$
  $!^{N2}\mathbf{new} r2 : \mathsf{seed}; \mathsf{Osign}(x : \mathsf{bitstring}) := \mathsf{sign}(r, x, r2),$
  $!^{N}\mathsf{Ocheck}(m1 : \mathsf{bitstring}, si1 : \mathsf{signature}) := \mathsf{ver}(\mathsf{exp}(g, r), si1, m1),$
  $!^{NDDH1}\mathsf{ODDH1}(m2 : G, m3 : G) := m2 = \mathsf{exp}(m3, r),$
  $!^{NDDH2}\mathsf{ODDH2}(m2 : G, m3 : Z) := m2 = \mathsf{exp}(g, \mathsf{mult}(m3, r)),$
  $!^{NDDH3}\mathsf{ODDH3}(m2 : G) := m2 = \mathsf{exp}(g, \mathsf{mult}(r, r))),$
$!^{N4}\mathsf{Ocheck2}(m : \mathsf{bitstring}, y : G, si : \mathsf{signature}) := \mathsf{ver}(y, si, m)$
$\approx$
$!^{N3}\mathbf{new}\ r : Z; (\mathsf{Opk}() := \mathsf{exp}(g, r),$
  $!^{N2}\mathbf{new}\ r2 : \mathsf{seed}; \mathsf{Osign}(x : \mathsf{bitstring}) := \mathbf{let}\ s : \mathsf{signature} = \mathsf{sign}'(r, x, r2)\ \mathbf{in}\ s,$
  $!^{N}\mathsf{Ocheck}(m1 : \mathsf{bitstring}, si1 : \mathsf{signature}) :=$
    $\mathbf{find}\ j \leq N2\ \mathbf{suchthat}\ \mathbf{defined}(x[j], s[j]) \wedge m1 = x[j] \wedge si1 = s[j]\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ \mathsf{false},$
  $!^{NDDH1}\mathsf{ODDH1}(m2 : G, m3 : G) := m2 = \mathsf{exp}(m3, r),$
  $!^{NDDH2}\mathsf{ODDH2}(m2 : G, m3 : Z) := m2 = \mathsf{exp}(g, \mathsf{mult}(m3, r)),$
  $!^{NDDH3}\mathsf{ODDH3}(m2 : G) := m2 = \mathsf{exp}(g, \mathsf{mult}(r, r))),$
$!^{N4}\mathsf{Ocheck2}(m : \mathsf{bitstring}, y : G, si : \mathsf{signature}) :=$
  $\mathbf{find}\ j \leq N2, k \leq N3\ \mathbf{suchthat}\ \mathbf{defined}(x[j, k], s[j, k], r[k]) \wedge y = exp(g, r[k]) \wedge$
    $m = x[j, k] \wedge si = s[j, k]\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}$
  $\mathbf{find}\ k \leq N3\ \mathbf{suchthat}\ \mathbf{defined}(r[k]) \wedge y = \mathsf{exp}(g, r[k])\ \mathbf{then}\ \mathsf{false}\ \mathbf{else}$
  $\mathsf{ver}(y, si, m).$
```

Figure 7: Formalization of SUF-CMA signatures, with additional Diffie-Hellman oracles

adversary: $\mathsf{ODDHa1}(m, m')$ corresponds to the decisional Diffie-Hellman oracle $\mathsf{DDH}(g, g^a, m', m)$, $\mathsf{ODDHa2}(m, m', j)$ is $\mathsf{DDH}(g^a, g^{b[j]}, m, m')$, and $\mathsf{ODDHa3}(m, m', j)$ is $\mathsf{DDH}(g^a, g^{a[j]}, m, m')$; ODDHb1, ODDHb2, and ODDHb3 are similar. These oracles are left unchanged in the right-hand side of the indistinguishability property.

In the full formalization in CryptoVerif, additional variants of the decisional Diffie-Hellman oracles are given, which encode particular cases in which m' is of the form $g^{b[j]}$ (or $g^{a[j]}$). In this case, m'^a reduces to $g^{b[j] \cdot a}$ and the oracle must be written in the latter form, otherwise CryptoVerif will not recognize that it can use the oracle to compute $g^{b[j] \cdot a}$.

The novel point is that, in the particular case of the considered protocol, we need a joint security assumption with signatures, so we add signature oracles using the keys a and b to this definition, oracles Osigna, Osigna′, Osignb, Osignb′ in Figure 6. These oracles use both symbol sign and symbol sign′ for signatures because the game transformation for signatures (shown below) transforms sign into sign′. These signature oracles are left unchanged in the GDH transformation.

CryptoVerif also supports giving some a's and b's to the adversary. Obviously, the CDH assumption is not applied to $m = g^{ab}$ for such a's and b's. We do not formalize that here, since it is not necessary for the considered protocol.

Similarly, the assumption on signatures (Figure 7) includes Diffie-Hellman oracles that are left unchanged. It also differs from the standard formalization of signatures in CryptoVerif in that the private/public key pair is a Diffie-Hellman key pair. In detail, the left-hand side first chooses a private key r, then it provides several oracles: Opk returns the corresponding public key, Osign returns the signature of a message, Ocheck verifies a signature, $\mathsf{ODDH1}, \dots, \mathsf{ODDH3}$ are the Diffie-Hellman oracles which allow the adversary to compute exponentiations using r, and finally $\mathsf{Ocheck2}$ is a signature verification oracle which may use any key, not only the public key $\mathsf{exp}(g, r)$ corresponding to r. The right-hand side provides the same oracles, but modifies the signature verification oracles: when they verify signatures under $\mathsf{exp}(g, r)$, they return true if and only if the candidate message-signature pair comes from the signature oracle. Additionally, the signature symbol sign is replaced with sign' in Osign; this trick prevents CryptoVerif from repeatedly applying the transformation of the left-hand side into the right-hand side to the same signatures.

B KDF256 and KDF128 are Independent Random Oracles

We assume that the compression function of SHA256 is a random oracle, and show that $\mathsf{KDF256}$ and $\mathsf{KDF128}$ are indifferentiable [27] from independent random oracles. The function SHA256 is used with arguments that have a fixed length by $\mathsf{KDF256}$ and $\mathsf{KDF128}$, so SHA256 (for that fixed length) is indifferentiable from a random oracle, as a particular case of the prefix-free Merkle-Damgård construction [27].

We have $\mathsf{KDF256}(k, (X, U, V)) = \mathit{Hash1}$ where

$$\begin{aligned}
&\mathit{SharedInfoMAC} \leftarrow \mathsf{concatSI}(1, \mathsf{HMAC\text{-}SHA256}, X, U, V, 256)\\
&\mathit{Hash1} \leftarrow \mathsf{SHA256}(\mathsf{concatKDF}(k, 1, \mathit{SharedInfoMAC}))
\end{aligned}$$

and $\mathsf{KDF128}(k, (X, U, V)) = \mathsf{truncate128}(\mathit{Hash1}')$ where

$$\begin{aligned}
&\mathit{SharedInfoENC} \leftarrow \mathsf{concatSI}(2, \mathsf{AES128\text{-}CFB128}, X, U, V, 128)\\
&\mathit{Hash1}' \leftarrow \mathsf{SHA256}(\mathsf{concatKDF}(k, 1, \mathit{SharedInfoENC}))
\end{aligned}$$

where $\mathsf{concatSI}$ and $\mathsf{concatKDF}$ are concatenation functions and $\mathsf{truncate128}$ truncates its input to 128 bits. The two calls to $\mathsf{SHA256}$ have disjoint domains, since the first argument of $\mathsf{concatSI}$ differs, so they are indifferentiable from independent random oracles. Furthermore, the truncation of a random oracle is indifferentiable from a random oracle, so $\mathsf{KDF256}$ and $\mathsf{KDF128}$ are indifferentiable from independent random oracles. The concatenation of two independent random oracles is indifferentiable from a random oracle, so the concatenation of $\mathsf{KDF256}$ and $\mathsf{KDF128}$ is indifferentiable from a random oracle.

C Proof of Indistinguishability for Encryption

Our goal is to show that the probability of distinguishing game (5) from the game (6) is at most the probability of event $\mathsf{repeatedIV}$ plus a negligible probability. (These games are given in Section 2.3.) We introduce the following intermediate game

$$\begin{aligned}
&\mathcal{O}_{\mathrm{gen}}() := k \xleftarrow{R} \mathsf{enc\_key}; \mathbf{return}(); \mathbf{foreach}\ i \leq n\ \mathbf{do}\\
&\quad \mathcal{O}_{\mathrm{enc}}(\mathit{IVdata}, m) :=\\
&\quad \mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(\mathit{IVdata}[j]) \wedge \mathit{IVdata} = \mathit{IVdata}[j]\\
&\quad \mathbf{then\ event}\ \mathsf{repeatedIV}\ \mathbf{else\ return}(\mathsf{ENC}(k, \mathit{IVdata}, m))
\end{aligned} \tag{18}$$

```
Ogen() :=
$k \xleftarrow{R}$ enc_key;
$IV0$ : blocksize ← AES($k$, Zero);
return();
(
  Oenc1($IVdata1$ : blocksize, $m1$ : blocksize, $l$ : len) :=
  if defined($IVdata2$) $\wedge$ $IVdata1 = IVdata2$ then event repeatedIV else
  if defined($IVdata3$) $\wedge$ $IVdata1 = IVdata3$ then event repeatedIV else
  $IV1$ ← xor($IV0$, $IVdata1$);
  $c1$ ← xor(AES($k$, $IV1$), $m1$);
  return(truncate($c1$, $l$))
) | (
  Oenc2($IVdata2$ : blocksize, $m21$ : blocksize, $m22$ : blocksize, $l$ : len) :=
  if defined($IVdata1$) $\wedge$ $IVdata2 = IVdata1$ then event repeatedIV else
  if defined($IVdata3$) $\wedge$ $IVdata2 = IVdata3$ then event repeatedIV else
  $IV2$ ← xor($IV0$, $IVdata2$);
  $c21$ ← xor(AES($k$, $IV2$), $m21$);
  $c22$ ← xor(AES($k$, $c21$), $m22$);
  return($c21$, truncate($c22$, $l$))
) | (
  Oenc3($IVdata3$ : blocksize, $m31$ : blocksize, $m32$ : blocksize, $m33$ : blocksize, $l$ : len) :=
  if defined($IVdata1$) $\wedge$ $IVdata3 = IVdata1$ then event repeatedIV else
  if defined($IVdata2$) $\wedge$ $IVdata3 = IVdata2$ then event repeatedIV else
  $IV3$ ← xor($IV0$, $IVdata3$);
  $c31$ ← xor(AES($k$, $IV3$), $m31$);
  $c32$ ← xor(AES($k$, $c31$), $m32$);
  $c33$ ← xor(AES($k$, $c32$), $m33$);
  return($c31$, $c32$, truncate($c33$, $l$))
)
```

Figure 8: Encryption: initial game

This game executes event repeatedIV when *IVdata* has been previously used, and otherwise encrypts m. Let us explain this game in more detail. The **foreach** construct allows one to call the following oracle $\mathcal{O}_{\text{enc}}$ at most n times. In CryptoVerif, all variables defined under **foreach** are implicitly arrays indexed by the index of **foreach**. In particular, the argument *IVdata* of $\mathcal{O}_{\text{enc}}$ is an array $IVdata[i]$, which contains the IV data of all previous queries to $\mathcal{O}_{\text{enc}}$. The **find** construct **find** $j \leq n$ **suchthat defined**($vars$) $\wedge$ $cond$ **then** P **else** P' looks for an index $j \leq q$ such that the variables in *vars* are defined and the condition *cond* holds. When such an index is found, the process P is executed with that index; otherwise, the process Q is executed. In the case of $\mathcal{O}_{\text{enc}}$, the **find** construct looks for a previous query to $\mathcal{O}_{\text{enc}}$, with index j such that its argument $IVdata[j]$ is the same as the current argument *IVdata*. When it is found, that is, when *IVdata* has been previously used, it executes event repeatedIV; otherwise, it returns the encryption of m.

The probability of distinguishing (5) from (18) is the probability of event repeatedIV. It just remains to show that (18) is indistinguishable from (6). CryptoVerif cannot fully show this result, because it lacks looping capabilities. In this appendix, we show this result on an example with 3 messages, containing respectively 1, 2, and 3 blocks. The proof starts from the initial game

```
$\mathsf{Ogen}() := \mathbf{return}();$
(
  $\mathsf{Oenc1}(\mathit{IVdata1} : \mathsf{blocksize}, \mathit{m1} : \mathsf{blocksize}, l : \mathsf{len}) :=$
  $\mathbf{if\ defined}(\mathit{IVdata2}) \wedge \mathit{IVdata1} = \mathit{IVdata2}$ then event $\mathsf{repeatedIV}$ else
  $\mathbf{if\ defined}(\mathit{IVdata3}) \wedge \mathit{IVdata1} = \mathit{IVdata3}$ then event $\mathsf{repeatedIV}$ else
  $r_{69} \overset{R}{\leftarrow} \mathsf{blocksize};$
  $\mathbf{return}(\mathsf{truncate}(r_{69}, l))$
) | (
  $\mathsf{Oenc2}(\mathit{IVdata2} : \mathsf{blocksize}, \mathit{m21} : \mathsf{blocksize}, \mathit{m22} : \mathsf{blocksize}, l : \mathsf{len}) :=$
  $\mathbf{if\ defined}(\mathit{IVdata1}) \wedge \mathit{IVdata2} = \mathit{IVdata1}$ then event $\mathsf{repeatedIV}$ else
  $\mathbf{if\ defined}(\mathit{IVdata3}) \wedge \mathit{IVdata2} = \mathit{IVdata3}$ then event $\mathsf{repeatedIV}$ else
  $r_{67} \overset{R}{\leftarrow} \mathsf{blocksize};$
  $r_{65} \overset{R}{\leftarrow} \mathsf{blocksize};$
  $\mathbf{return}(r_{67}, \mathsf{truncate}(r_{65}, l))$
) | (
  $\mathsf{Oenc3}(\mathit{IVdata3} : \mathsf{blocksize}, \mathit{m31} : \mathsf{blocksize}, \mathit{m32} : \mathsf{blocksize}, \mathit{m33} : \mathsf{blocksize}, l : \mathsf{len}) :=$
  $\mathbf{if\ defined}(\mathit{IVdata1}) \wedge \mathit{IVdata3} = \mathit{IVdata1}$ then event $\mathsf{repeatedIV}$ else
  $\mathbf{if\ defined}(\mathit{IVdata2}) \wedge \mathit{IVdata3} = \mathit{IVdata2}$ then event $\mathsf{repeatedIV}$ else
  $r_{63} \overset{R}{\leftarrow} \mathsf{blocksize};$
  $r_{61} \overset{R}{\leftarrow} \mathsf{blocksize};$
  $r_{59} \overset{R}{\leftarrow} \mathsf{blocksize};$
  $\mathbf{return}(r_{63}, r_{61}, \mathsf{truncate}(r_{59}, l))$
)
```

Figure 9: Encryption: final game

shown in Figure 8, which is the instance of game (18) corresponding to the example we consider. In this game, *IVdata1*, *IVdata2*, and *IVdata3* are guaranteed to be pairwise distinct: in case they are not, the encryption oracles terminate with the event repeatedIV. AES is assumed to be a pseudo-random permutation. Each message consists of a certain number of 128-bits blocks; the last block may be partially used. The encryption oracles in the game above take as argument the blocks of the message as well as the length of the useful part of the last block.

CryptoVerif automatically transforms this game into the game of Figure 9. We can notice that this game does not depend on the actual content of the message blocks (*m1*, *m21*, *m22*, ...) but only on the length of the blocks: the ciphertext is a random bitstring of the same length as the plaintext. Therefore, we can apply the same game transformation to the game that encrypts zero-blocks instead of the real message (as in game (6)), and we are going to obtain exactly the same final game. Hence, the initial game in which the encrypt the real messages is indistinguishable from the game in which we encrypt zero-blocks (with overwhelming probability).

D HMAC-SHA256, KDF256, and KDF128 are Independent PRFs

We first prove that HMAC-SHA256, KDF256, and KDF128 are independent PRFs when the compression function of SHA256 is a pseudo-random function and, given a 256-bit random key

```
Ogen() := $k \stackrel{R}{\leftarrow}$ mac_key; return();
(
  foreach $i \leq$ qMAC do
  O_MAC($m$ : bitstring) :=
  $K0 \leftarrow$ padkey($k$);
  return(SHA256_comp(SHA256_comp(SHA256_init, xor($K0$, opad)),
    padsha256_2(SHA256_comp_star(SHA256_comp(SHA256_init, xor($K0$, ipad)),
    padsha256($m$)))))
) | (
  foreach $i \leq$ qKDF256 do
  O_KDF256($X2$ : t_SHA256_out, $U2$ : t_id, $V2$ : t_id) :=
  $SharedInfoMAC \leftarrow$ concatSI(MACcst, HMAC_SHA256num, $X2$, $U2$, $V2$, HMAC256keylength);
  $Hash1 \leftarrow$ SHA256_KDF(concatKDF($k$, count1, $SharedInfoMAC$));
  return($Hash1$)
) | (
  foreach $i \leq$ qKDF128 do
  O_KDF128($X3$ : t_SHA256_out, $U3$ : t_id, $V3$ : t_id) :=
  $SharedInfoENC \leftarrow$ concatSI(ENCcst, AES128num, $X3$, $U3$, $V3$, AES128keylength);
  $Hash1 \leftarrow$ SHA256_KDF(concatKDF($k$, count1, $SharedInfoENC$));
  return(truncate128($Hash1$))
)
```

Figure 10: MAC and key derivation: initial game

k, SHA256_comp(SHA256_init, $k \oplus$ opad) and SHA256_comp(SHA256_init, $k \oplus$ ipad) are independent pseudo-random bitstrings, and the function $x \mapsto$ SHA256_comp(SHA256_init, $k \| x$) is a pseudo-random function independent from those two bitstrings. This proof starts from the initial game shown in Figure 10. This game first defines an oracle Ogen, which generates the key k used by HMAC-SHA256, KDF256, and KDF128. Then it returns control to the adversary and makes available 3 oracles: O_MAC computes HMAC-SHA256, O_KDF256 computes KDF256, and O_KDF128 computes KDF128. All these oracles can be called several times, as defined by the **foreach** construct; for instance, O_MAC can be called at most qMAC times. We make explicit that the hash function SHA256 is built from the compression function SHA256_comp using the Merkle-Damgård construction [29]. In particular, the function SHA256_KDF computes SHA256 for a message that consists of two blocks, as follows:

$$\begin{aligned}&\mathsf{SHA256\_KDF}(x : \mathsf{t\_SHA256\_KDF\_in}) = \\ &\quad \mathsf{SHA256\_comp}(\mathsf{SHA256\_comp}(\mathsf{SHA256\_init}, \mathsf{first\_block}(x)), \mathsf{second\_block}(x))\end{aligned}$$

The key derivation functions KDF128 and KDF256 call SHA256 only with messages that consist of two blocks. In the same line, SHA256_comp_star is the iteration of SHA256_comp for any number of blocks. This game uses the following constants: MACcst $= 1$ and ENCcst $= 2$ are used in order to distinguish the information for generating MAC keys from the one for generating encryption keys; HMAC_SHA256num = 0xF1 designates the HMAC-SHA256 algorithm; AES128num = 0x01 designates the AES128-CFB128 algorithm; HMAC256keylength $= 256$ is the length of keys for HMAC-SHA256; AES128keylength $= 128$ is the length of keys for AES128-CFB128; count1 $= 1$ and count2 $= 2$ are two values of a counter.

Using the assumptions on the SHA256 compression function, and guided by proof indications,

```
Ogen() := return();
(
  foreach $i \leq$ qMAC do
  O_MAC($m$ : bitstring) :=
  find [unique] $j_1 = j_1' \leq$ qMAC suchthat defined($m[j_1'], r_1[j_1']$) $\wedge$ ($m = m[j_1']$) then
      return($r_1[j_1]$)
  else
      $r_1 \stackrel{R}{\leftarrow}$ t_SHA256_out; return($r_1$)
) | (
  foreach $i \leq$ qKDF256 do
  O_KDF256($X2$ : t_SHA256_out, $U2$ : t_id, $V2$ : t_id) :=
  find [unique] $j_2 = j_2' \leq$ qKDF256 suchthat defined($X2[j_2'], U2[j_2'], V2[j_2'], r_2[j_2']$) $\wedge$
    ($X2 = X2[j_2']$) $\wedge$ ($U2 = U2[j_2']$) $\wedge$ ($V2 = V2[j_2']$) then
      return($r_2[j_2]$)
  else
      $r_2 \stackrel{R}{\leftarrow}$ mac_key; return($r_2$)
) | (
  foreach $i \leq$ qKDF128 do
  O_KDF128($X3$ : t_SHA256_out, $U3$ : t_id, $V3$ : t_id) :=
  find [unique] $j_3 = j_3' \leq$ qKDF128 suchthat defined($X3[j_3'], U3[j_3'], V3[j_3'], r_3[j_3']$) $\wedge$
    ($X3 = X3[j_3']$) $\wedge$ ($U3 = U3[j_3']$) $\wedge$ ($V3 = V3[j_3']$) then
      return($r_3[j_3]$)
  else
      $r_3 \stackrel{R}{\leftarrow}$ enc_key; return($r_3$)
)
```

Figure 11: MAC and key derivation: final game

CryptoVerif transforms this initial game into the final game of Figure 11. (We renumbered some variables for readability.)

Hence, CryptoVerif proves that this final game is indistinguishable from the initial game (with overwhelming probability). This final game includes the same oracles as the initial game, but now O_MAC, O_KDF256, and O_KDF128 are independent pseudo-random functions. Consider for example the oracle O_MAC. In CryptoVerif, all variables defined under **foreach** are implicitly arrays indexed by the index of **foreach**. So the argument m of O_MAC is an array $m[i]$, which contains the arguments of all previous queries to O_MAC. The **find** construct **find** $j = j' \leq q$ **suchthat defined**($vars$)$\wedge$$cond$ **then** P **else** P' looks for an index $j' \leq q$ such that the variables in $vars$ are defined and the condition $cond$ holds. When such an index is found, it is stored in j and the process P is executed; otherwise, the process Q is executed. In the case of O_MAC, the **find** construct looks for a previous query to O_MAC, with index j_1' such that its argument $m[j_1']$ is the same as the current argument m. When it is found, it returns the previously returned result $r_1[j_1]$; otherwise, it generates a fresh random number r_1 and returns it. This is exactly the definition of a pseudo-random function: it returns a random output on a fresh query and the previously returned result when the query has already been asked. The situation is similar for O_KDF256 and O_KDF128.

So CryptoVerif has proved that HMAC-SHA256, KDF256, and KDF128 are indistinguishable from independent pseudo-random functions (with overwhelming probability).

```
Ostart() := $hk1 \xleftarrow{R}$ hashkey1; return();
(
  foreach $i2 \leq$ N2 do
  Ogen() := $k \xleftarrow{R}$ mac_key; return();
  (
    O1() := return(SHA256_comp($hk1$, SHA256_initxor(padkey($k$), opad)))
  |
    O2() := return(SHA256_comp($hk1$, SHA256_init, xor(padkey($k$), ipad)))
  |
    foreach $i \leq$ N do
    O3($x$ : mac_key) := return(SHA256_comp($hk1$, SHA256_init, concatSHA256($k$, $x$)))
  )
) | (
  foreach $i1 \leq$ N1 do
  OH($x1$ : t_SHA256_out, $x2$ : t_SHA256_block) := return(SHA256_comp($hk1$, $x1$, $x2$))
)
```

Figure 12: PRF properties of the compression function of SHA-256: initial game

Next, we prove that HMAC-SHA256, KDF256, and KDF128 are independent PRFs when the compression function of SHA256 is a random oracle. For this proof, we rely on the previous result. A random oracle is a fortiori a pseudo-random function, so we just have to prove that, given a 256-bit random key k,

$$\mathsf{SHA256\_comp}(\mathsf{SHA256\_init}, k \oplus \mathsf{opad}) \text{ and}$$
$$\mathsf{SHA256\_comp}(\mathsf{SHA256\_init}, k \oplus \mathsf{ipad})$$

are independent pseudo-random bitstrings, and the function

$$x \mapsto \mathsf{SHA256\_comp}(\mathsf{SHA256\_init}, k \| x)$$

is a pseudo-random function independent from those two bitstrings. This proof starts from the initial game shown in Figure 12. This game chooses N2 keys k and defines oracles O1 which returns SHA256_comp(SHA256_init, $k \oplus$opad), O2 which returns SHA256_comp(SHA256_init, $k \oplus$ ipad), and O3 which maps x to SHA256_comp(SHA256_init, $k \| x$). It also defines a random oracle for SHA256_comp (oracle OH). The function SHA256_comp takes as argument an additional key, chosen at the beginning of the game. This key models the choice of the random oracle itself.

Under the assumption that SHA256_comp is a random oracle, CryptoVerif automatically transforms this game into the final game shown in Figure 13. (We renumbered some variables for readability.) This game defines the same oracles as the initial game, but now O1 and O2 return a fresh random number, O3 is a pseudo-random function, OH is still a random oracle. (Its code is expanded into a function that returns the previous result when the argument has already been seen and a fresh random number otherwise.)

CryptoVerif proves that this final game is indistinguishable from the initial game (with overwhelming probability), which is the desired result.

```
Ostart() := return();
(
  foreach $i2 \leq \mathsf{N2}$ do
  Ogen() := $k \stackrel{R}{\leftarrow}$ mac_key; return();
  (
    O1() := $r_1 \stackrel{R}{\leftarrow}$ t_SHA256_out; return($r_1$)
  |
    O2() := $r_2 \stackrel{R}{\leftarrow}$ t_SHA256_out; return($r_2$)
  |
    foreach $i \leq \mathsf{N}$ do
    O3($x$ : mac_key) :=
    find [unique] $j_3 = j'_3 \leq \mathsf{N}$ suchthat defined($r_3[j'_3], x[j'_3]$) $\wedge$ ($x = x[j'_3]$) then
        return($r_3[j_3]$)
    else
        $r_3 \stackrel{R}{\leftarrow}$ t_SHA256_out; return($r_3$)
  )
) | (
  foreach $i1 \leq \mathsf{N1}$ do
  OH($x1$ : t_SHA256_out, $x2$ : t_SHA256_block) :=
  find [unique] $j_4 = j'_4 \leq \mathsf{N1}$ suchthat
    defined($r_4[j'_4], x2[j'_4], x1[j'_4]$) $\wedge$ ($x1 = x1[j'_4]$) $\wedge$ ($x2 = x2[j'_4]$) then
      return($r_4[j_4]$)
  else
      $r_4 \stackrel{R}{\leftarrow}$ t_SHA256_out; return($r_4$)
)
```

Figure 13: PRF properties of the compression function of SHA-256: final game

E HMAC-SHA256 is SUF-CMA, and KDF256 and KDF128 are Independent PRFs

Our goal is to show that the game G_1 shown in Figure 14 is indistinguishable from the game G_2 shown in the same figure (with overwhelming probability) using the assumption that MAC, KDF256, and KDF128 are independent pseudo-random functions, as shown in Appendix D from two different assumptions.

The game G_1 defines an oracle Ogen that generates the key k, and then provides 4 oracles, for computing a MAC, for verifying a MAC, and for computing KDF256 and KDF128. In game G_2, we have the same oracles, but the MAC verification oracle O_Ver additionally checks that the message is in the array of messages $m[j']$ whose MAC has been previously computed by the MAC oracle. This check guarantees that the MAC cannot be forged and corresponds to the SUF-CMA property. Moreover, the oracles for computing KDF256 and KDF128 are pseudo-random functions (defined as in Appendix D).

It is difficult to guide CryptoVerif to transform G_1 into G_2, so we proceed differently. We choose randomly a boolean b, write a game that runs as G_1 when b is true and as G_2 when b is false, and show secrecy of b. This property proves that G_1 is indistinguishable from G_2 (with overwhelming probability).

Game G_1:

```
Ogen() := $k \xleftarrow{R}$ mac_key; return();
(
  foreach $i \leq$ qMAC do
  O_MAC($m$ : bitstring) := return(MAC($k, m$))
) | (
  foreach $i \leq$ qVer do
  O_Ver($mv$ : bitstring, $mac$ : t_SHA256_out) := return(($mac$ = MAC($k, mv$)))
) | (
  foreach $i \leq$ qKDF256 do
  O_KDF256($X2$ : t_SHA256_out, $U2$ : t_id, $V2$ : t_id) := return(KDF256($k, X2, U2, V2$))
) | (
  foreach $i \leq$ qKDF128 do
  O_KDF128($X3$ : t_SHA256_out, $U3$ : t_id, $V3$ : t_id) := return(KDF128($k, X3, U3, V3$))
)
```

Game G_2:

```
Ogen() := $k \xleftarrow{R}$ mac_key; return();
(
  foreach $i \leq$ qMAC do
  O_MAC($m$ : bitstring) := return(MAC($k, m$))
) | (
  foreach $i \leq$ qVer do
  O_Ver($mv$ : bitstring, $mac$ : t_SHA256_out) :=
  find $j = j' \leq$ qMAC suchthat defined($m[j']$) $\wedge$ ($mv = m[j']$) $\wedge$ ($mac$ = MAC($k, mv$)) then
      return(true)
  else
      return(false)
) | (
  foreach $i \leq$ qKDF256 do
  O_KDF256($X2$ : t_SHA256_out, $U2$ : t_id, $V2$ : t_id) :=
  find [unique] $j_2 = j'_2 \leq$ qKDF256 suchthat defined($X2[j'_2], U2[j'_2], V2[j'_2], r_2[j'_2]$) $\wedge$
    ($X2 = X2[j'_2]$) $\wedge$ ($U2 = U2[j'_2]$) $\wedge$ ($V2 = V2[j'_2]$) then
      return($r_2[j_2]$)
  else
      $r_2 \xleftarrow{R}$ mac_key; return($r_2$)
) | (
  foreach $i \leq$ qKDF128 do
  O_KDF128($X3$ : t_SHA256_out, $U3$ : t_id, $V3$ : t_id) :=
  find [unique] $j_3 = j'_3 \leq$ qKDF128 suchthat defined($X3[j'_3], U3[j'_3], V3[j'_3], r_3[j'_3]$) $\wedge$
    ($X3 = X3[j'_3]$) $\wedge$ ($U3 = U3[j'_3]$) $\wedge$ ($V3 = V3[j'_3]$) then
      return($r_3[j_3]$)
  else
      $r_3 \xleftarrow{R}$ enc_key; return($r_3$)
)
```

Figure 14: HMAC-SHA256 is SUF-CMA, and KDF256 and KDF128 are independent PRFs

Game G_1:

```
$\mathsf{Ogen}() := k \xleftarrow{R} \mathsf{mac\_key}; \mathbf{return}();$
(
  $\mathbf{foreach}\ i \leq \mathsf{qMAC}\ \mathbf{do}$
  $\mathsf{O\_MAC}(m : \mathsf{bitstring}) := \mathbf{return}(\mathsf{MAC}(k, m))$
) | (
  $\mathbf{foreach}\ i \leq \mathsf{qColl}\ \mathbf{do}$
  $\mathsf{O\_Coll}(m1 : \mathsf{bitstring}, m2 : \mathsf{bitstring}) := \mathbf{return}((\mathsf{MAC}(k, m1) = \mathsf{MAC}(k, m2)))$
)
```

Game G_2:

```
$\mathsf{Ogen}() := k \xleftarrow{R} \mathsf{mac\_key}; \mathbf{return}();$
(
  $\mathbf{foreach}\ i \leq \mathsf{qMAC}\ \mathbf{do}$
  $\mathsf{O\_MAC}(m : \mathsf{bitstring}) := \mathbf{return}(\mathsf{MAC}(k, m))$
) | (
  $\mathbf{foreach}\ i \leq \mathsf{qColl}\ \mathbf{do}$
  $\mathsf{O\_Coll}(m1 : \mathsf{bitstring}, m2 : \mathsf{bitstring}) := \mathbf{return}((m1 = m2))$
)
```

Figure 15: HMAC-SHA256 is collision resistant

F HMAC-SHA256 is Collision Resistant

Our goal is to show that the game G_1 shown in Figure 15 is indistinguishable from the game G_2 shown in the same figure (with overwhelming probability) using the assumption that MAC, KDF256, and KDF128 are independent pseudo-random functions, as shown in Appendix D from two different assumptions. (In fact, knowing that MAC is a pseudo-random function would be sufficient.)

The game G_1 defines an oracle Ogen that generates the key k, and then provides 2 oracles: one for computing MACs, and one that tests equality between the MACs of two messages m_1 and m_2. The game G_2 is similar, but tests equality between messages m_1 and m_2.

As in Appendix E, it is difficult to guide CryptoVerif to transform G_1 into G_2, so we choose randomly a boolean b, write a game that runs as G_1 when b is true and as G_2 when b is false, and show secrecy of b. This property proves that G_1 is indistinguishable from G_2 (with overwhelming probability).

RESEARCH CENTRE
PARIS

2 rue Simone Iff
CS 42112
75589 Paris Cedex 12

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399