Analysis of authentication and key establishment in inter-generational mobile telephony (with appendix July 31, 2013)

Chunyu Tang, David A. Naumann, and Susanne Wetzel
Stevens Institute of Technology

***Abstract*—Second (GSM), third (UMTS), and fourth-generation (LTE) mobile telephony protocols are all in active use, giving rise to a number of interoperation situations. Although the standards address roaming by specifying switching and mapping of established security context, there is not a comprehensive specification of which are the possible interoperation cases. Nor is there comprehensive specification of the procedures to establish security context (authentication and short-term keys) in the various interoperation scenarios. This paper systematically enumerates the cases, classifying them as allowed, disallowed, or uncertain with rationale based on detailed analysis of the specifications. We identify the authentication and key agreement procedure for each of the possible cases. We formally model these scenarios and analyze their security, in the symbolic model, using the tool ProVerif. We find two scenarios that inherit a known false base station attack. We find an attack on the CMC message of another scenario.**

I. Introduction

Mobile telephony has become an integral part of our daily activities, in part due to the tremendous success and market penetration of smartphones and tablets. In many locations around the world, mobile communication is already facilitated through the fourth generation (4G) technology called Long Term Evolution (LTE)—which evolved from the third generation (3G) technology, Universal Mobile Telecommunications System (UMTS). Along with the opportunities created by technology evolution there are challenges. One challenge is the interoperation of different generations of technologies, i.e., communication involving mixed network components. Past experience has shown that such interoperation may introduce unexpected security vulnerabilities [1], [2].

The specifications promulgated by the 3GPP organization for UMTS [3] and LTE [4] do address interoperation between the different generations of technologies. The specification for UMTS systematically studies all possible combinations of interoperation between UMTS and second generation (2G) GSM. The LTE specification details the mechanisms for security context switching and mapping to facilitate interoperation between LTE, UMTS, and GSM. However, this applies to maintaining context during handover and idle mode mobility. To the best of our knowledge, the specification for LTE does not explicitly address establishing of an initial security context for interoperation. In particular, to date there is no comprehensive enumeration of all interoperation cases and their respective procedures for authentication and key agreement (AKA). In this paper we close this gap.

The first contribution of this paper is to systematically enumerate of all possible interoperation cases between LTE, UMTS, and GSM. We classify these cases as allowed, disallowed, or uncertain, with explicit rationale making detailed reference to the specifications. Of the 243 cases identified, 19 cases involve GSM and UMTS technologies only, and as such are fully treated by the UMTS specification [3]. For cases involving LTE components, 138 cases are clearly ruled out somewhere in the specification and 38 cases are clearly allowed. For the remaining 48 cases the specifications and documentation based on the specifications do not provide a clear indication whether these cases are allowed.

As a second contribution of this paper, we provide details on what we call the *AKA scenarios*,[1] i.e., the specific protocol steps for authentication and key agreement, in each allowed or uncertain case. For each uncertain case we identify conditions under which the case could occur. For all of the 19+38+48 cases we identify the corresponding AKA scenario. It turns out that there are only 10 distinct AKA scenarios, including the pure GSM, pure UMTS, and pure LTE scenarios which apply in some interoperation cases. Although the GSM/UMTS scenarios are described in the specifications, that is not the case for 86 roaming cases involving LTE. For three of those scenarios we identify two variations which are both consistent with the specifications and which have different authenticity properties.

As a third contribution, we provide formal models for all 10 of the AKA scenarios, including variations, in the symbolic (Dolev-Yao) model of cryptography and using the ProVerif tool [5]. We provide a security analysis based on these models. The models are composed in a modular fashion from the basic protocol models for GSM, UMTS, and LTE. This will facilitate adding or modifying scenarios, in case of changes to the specifications or the conditions for the uncertain cases to occur.

Our security analysis addresses authentication properties and secrecy. We show that two of the LTE interoperation scenarios inherit an attack, known from GSM and the interoperation between UMTS and GSM [6], in which a false base station can eavesdrop and modify data traffic. We show how the attack can be prevented in one scenario. We also show that one scenario is prone to an attack against the *Cipher Mode Command* (CMC) message.

Outline: Sect. II surveys related work and Sect. III is an overview of the GSM, UMTS, and LTE AKA protocols. Sect. IV presents the first of our main contributions, the systematic enumeration of possible cases and classification of what is (dis)allowed or uncertain according to the 3GPP. Sect. V presents our second contribution, the AKA scenarios for each allowed or uncertain case, justified with reference to

[1] In some standards, AKA has more specific meaning and varying terminology is used, e.g., depending on whether authentication is mutual.

the specifications. Sect. VI describes our ProVerif models for GSM, UMTS, and LTE, and the specifications of desired security properties. Sect. VII presents our third contribution, the ProVerif models for AKA scenarios involving interoperation between technologies, and analysis results for those models.

For reasons of space, we cannot present the complete classification of cases, scenarios, and analysis results; instead we present excerpts and highlights. A long version online includes full details [7].

II. Related work

Several attacks have been found against the GSM encryption algorithms [8], [9], [10], [11], [12]. Ahmadian et al. [13] show attacks which exploit weakness of one GSM cipher to eavesdrop or impersonate a UMTS subscriber in a mixed network. In this paper we focus on protocol flaws rather than cryptographic weaknesses.

Fox [14] finds the false base station attack on the GSM AKA due to the lack of authentication of the network. Meyer and Wetzel [1], [2], [15] show that a man-in-the-middle attack can be performed on one of the cases of interoperation between GSM and UMTS. In prior work [16], we use the ProVerif (PV) tool to analyze GSM, UMTS, and roaming cases between GSM and UMTS. The false base station attack [14] and the man-in-the-middle attack [1] were confirmed by the PV models.

PV is an automatic protocol verifier that can verify authentication, secrecy, and other properties, in the symbolic (Dolev-Yao) model, considering an unbounded number of sessions and unbounded message space. Quite a few protocols have been analyzed using PV. For example, Chang and Shmatikov [17] use PV to analyze the Bluetooth device pairing protocols; they rediscover an offline guessing attack [18] as well as a new attack. Blanchet and Chaudhuri [19] find an integrity attack against a file sharing protocol. Chen and Ryan analyze TPM authorization [20]. Kremer and Ryan use PV to verify an electronic voting protocol [21]. Arapinis et al. [22] find two attacks against anonymity in UMTS, using PV.

Han and Choi [23] demonstrate a threat against the LTE handover key management, involving a compromised base station. This is concerned with maintaining security context, whereas our work addresses establishing such context. Tsay and Mjølsnes [24] find an attack on the UMTS and LTE AKA protocols using CryptoVerif, an automated protocol analyzer based on a computational model. In fact the attack lives at the symbolic level. It depends on insecurity of the connection between the serving network and the home network. In our work we assume the connection between serving network and home network is secure. (Although the standard specifies the protocols, their implementations are operator-specific.)

Lee et al. [25] analyze the anonymity property of the UMTS and LTE AKA and connection establishment protocols using formal security (computational) models. The assumption in this work is that the attacker is not capable of impersonating any network devices and the underlying cryptographic system is perfect. They manually prove the protocols meet the anonymity requirement, under these assumptions.

Mobarhan et al. [26] evaluate the publically known attacks on GSM and UMTS (and the related technology GPRS), categorizing them in terms of secrecy, integrity, or authenticity properties. Possible security improvements are also discussed.

III. Overview of GSM, UMTS, and LTE security mechanisms

In GSM, UMTS, and LTE, the network architecture includes three main elements: the *Mobile Station* (MS), the *Serving Network* (SN), and the *Home Network* (HN).

The MS is the combination of the *Mobile Equipment* (ME) and an identity module. The ME is the user device that contains the radio functionality and the encryption/integrity mechanisms used to protect the traffic between the MS and the network. A 4G ME also includes the functionality to derive an LTE master secret key K_{ASME}. In GSM, the identity module of the *Subscriber Identity Module* (SIM) contains the unique *International Mobile Subscriber Identity* (IMSI), the subscriber's permanent secret key Ki, as well as the mechanisms used for GSM AKA and GSM session key derivation. The UMTS identity module (USIM) includes the IMSI, Ki, and the UMTS AKA and session key derivation functionality. It furthermore may contain the SIM functionality, i.e., the GSM AKA and key derivation functionality. In contrast to a 3G USIM, an LTE USIM (also refered to as enhanced USIM) provides for additional functionality including enhanced capability for the storing of a security context.

The SN typically consists of the *Base Station* (BS) and either the *Visitor Location Register/Serving GPRS Support Node* (VLR/SGSN) in GSM and UMTS, or the *Mobile Management Entity* (MME) in LTE. The BS is the network access point which manages the radio resources and establishes the connection to the MS. In GSM, the BS includes the *Base Transceiver Station* (BTS) which connects to the *Base Station Controller* (BSC). In GSM, encryption terminates at the BTS or at the SGSN in GPRS. In UMTS, the BS includes the NodeB which connects to the *Radio Network Controller* (RNC). Encryption and integrity protection in UMTS terminates in the RNC. In LTE, BS is the *evolved NodeB* (eNodeB). LTE distinguishes the protection of the connection between the MS and the eNodeB—the so-called *Access Stratum* (AS) and the connection between the MS and MME—the so-called *Non-Access Stratum* (NAS). In LTE, the MME is the end-point for the NAS and the respective protection mechanisms.

The HN includes the *Home Location Register* (HLR) and the *Authentication Center* (AuC) in GSM and UMTS, respectively the Home Subscriber Server (HSS) in LTE. The HN stores all subscriber data including the IMSI and permanent shared secret key Ki. It furthermore, holds its (own) algorithms for deriving session keys as well as generating authentication vectors. A 4G HN also includes the functionality for deriving an LTE master secret key K_{ASME}.

Overview of GSM Security Mechanisms. Fig. 1 shows the GSM AKA procedure. The goal of the GSM AKA is to authenticate the MS and to establish an encryption key that can then be used to protect the user data exchange between the MS and BS. The GSM AKA procedure can be triggered by the initial network attach request [27], the Routing Area Update (RAU) request [28], or the service request [29]. The service request happens after a dedicated channel has been established between the MS and the SN [29], which means

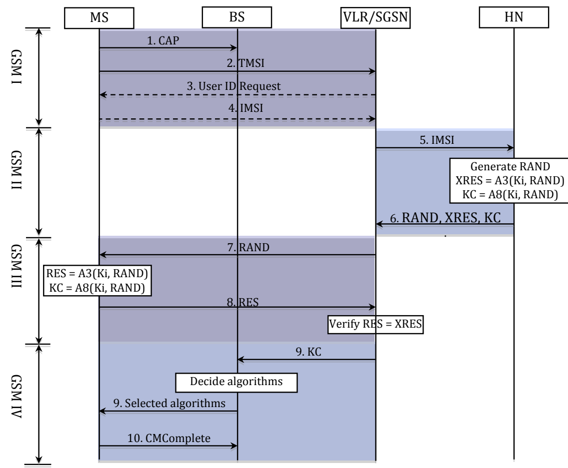

Fig. 1. GSM message sequence diagram [29], [15]

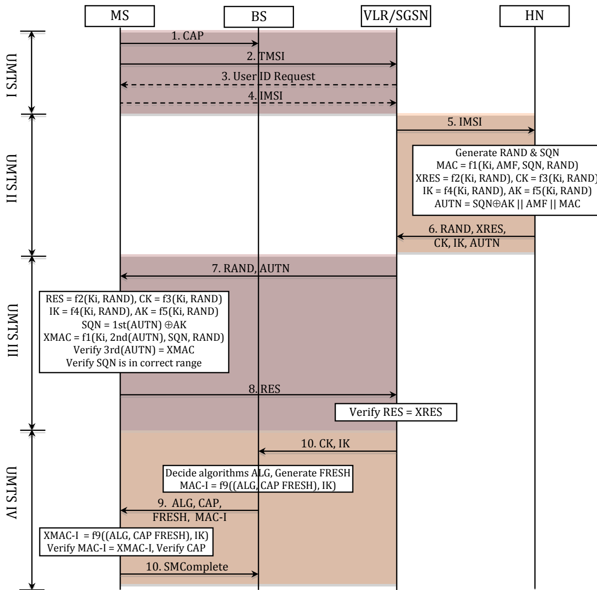

Fig. 2. UMTS message sequence diagram [3], [15]

that the attach request must have been executed previously. The identity and *CAPabilities* (CAP) in the attach request or the RAU request are used in the AKA procedure. Therefore, in GSM I, the MS sends the identity and the CAP to the SN. In GSM block II, the SN obtains authentication vectors from the HN. In GSM III a typical challenge-response procedure is carried out to authenticate the MS to SN. Then, in GSM IV, the VLR/SGSN provides the BS with the session key Kc. BS selects the encryption algorithm based on MS's capabilities and informs MS.

GSM is prone to a *false base station attack* [14] as the GSM AKA only authenticates the MS to the SN. Since a false BS can intercept and modify the sending of MS's capabilities, a false BS may force the use of no encryption thus enabling the false BS to control all traffic between the MS and the network.

Overview of UMTS Security Mechanisms. Similar to GSM, the UMTS AKA procedure can be triggered by the attach request or the RAU request. In UMTS I, the same

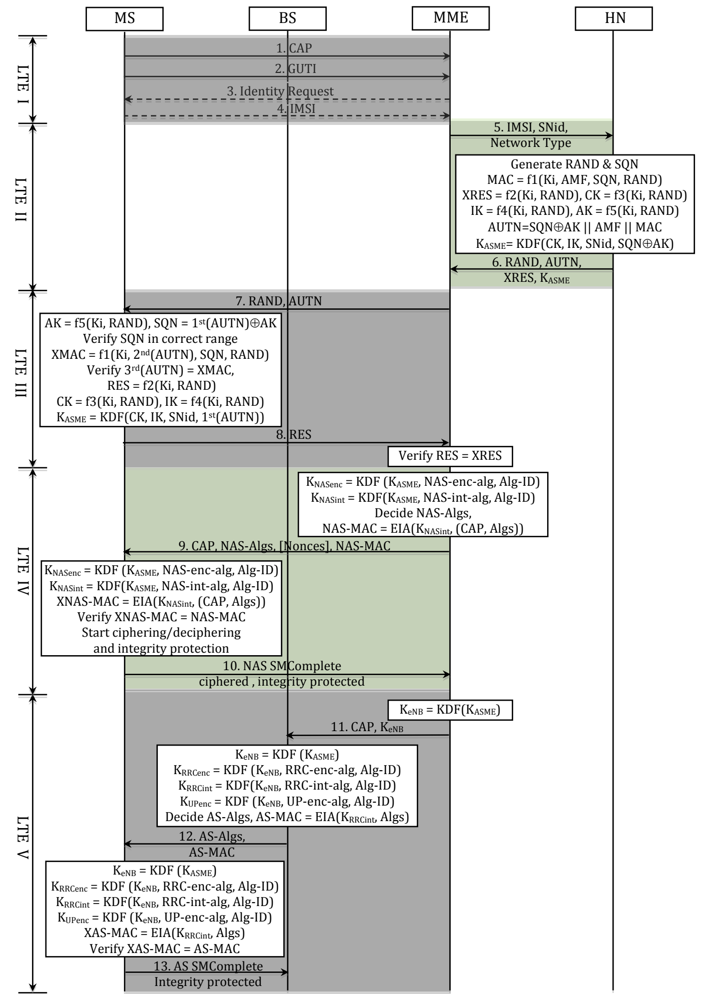

Fig. 3. LTE message sequence diagram [4]

messages are transmitted as in GSM I. In comparison to GSM, UMTS includes mechanisms for integrity protection. Specifically, as part of block UMTS II (see Fig. 2), the HN derives session keys for both encryption and integrity protection based on MS's long-term secret key Ki. Like in GSM, MS authenticates to the VLR/SGSN through a challenge-response protocol using the authentication vector that the VLR/SGSN obtained from HN. The authentication of the network to MS is achieved indirectly, as the BS integrity protects the sending of CAP which it can only do if it has received key IK from HN via VLR/SGSN. This prevents a false base station attack.

Overview of LTE Security Mechanisms. The LTE AKA (Fig. 3) is built on the UMTS AKA. In contrast to UMTS security, LTE introduces an enhanced key derivation hierarchy that allows the distinguishing of protection mechanisms on NAS and AS. Furthermore, inclusion of the id of SN as part of the key derivation enables the MS to indirectly authenticate the MME (through the successful use of derived keys). In addition, LTE defines a comprehensive security context framework, including native vs. mapped security contexts, full vs. partial security contexts, and current vs. non-current contexts [4]. A security context typically consists of a set of security parameters including cryptography keys and identifiers for respective cryptographic mechanisms.

The LTE AKA can be triggered by the initial network attach request, the Tracking Area Update (TAU) request or the service request [28]. When a NAS signalling connection exists, the network can initiate an authentication procedure at any time [28]. Before the service request or the NAS signalling connection establishing, the attach request must have already been executed [28]. Therefore, the first block of the LTE AKA can contain an attach request or a TAU request. If the AKA starts with an attach request, the first block (LTE I) contains the transmission of the identity and the security capabilities of the MS. If the AKA starts with a TAU request, in the first block (LTE I'), an additional nonce $NONCE_{MS}$ is sent to the MME. The nonce in the TAU request is only used when mapping an UMTS to an EPS security context. However, since the MS does not know when the mapping will happen, the nonce is always included in the TAU request message [2] [30]. In LTE, the master key K_{ASME} is derived and provided to MME together with the respective authentication vector. Unlike in GSM and UMTS, the id of the SN is an input to the key derivation, i.e., the derived key is bound to a specific MME. In block LTE IV, MME derives the keys which are used to protect NAS, while in LTE V the BS derives the keys to protect AS as well as user data. The MS does the corresponding key derivations in LTE IV and LTE V. Furthermore, the proper use of keys derived from K_{ASME} indirectly authenticates the MME to MS. Given that LTE distinguishes between the protection of NAS and AS, MME selects the respective algorithms to protect NAS (based on MS's capabilities) and announces the choice to MS in LTE IV as part of the NAS *Security Mode Command* (SMC). Similarly, BS announces its choice of algorithms in LTE V as part of the AS SMC.

IV. Establishing a native security context in interoperation

As mentioned previously, LTE introduces a comprehensive framework for handling security contexts [4]. In particular, this includes the mapping of security contexts in the case of interoperation of LTE with GSM or UMTS. The specification defines the use of existing native or mapped security contexts and recommends the performing of an AKA procedure once a mapped security context is used. However, to the best of our knowledge, the specification to date does not include details on what this AKA is to entail in case of interoperation of LTE, UMTS, and GSM, i.e., if the network components are from different generations of technologies.

In the following, we systematically enumerate all possible interoperation cases and classify them as allowed, disallowed, or uncertain based on various information in the 3GPP specifications for GSM, UMTS, and LTE. In Sect. V we then focus on the allowed and uncertain cases only determining the specific AKA scenarios for each one of these cases.

Enumeration of Interoperation Cases. As discussed in Sect. III, there are five main system components: the identity module, the ME, the BS, the VLR/SGSN/MME and the HN. Each one of those components can be 2G, 3G, or 4G—thus resulting in $3^5 = 243$ possible combinations. Table I shows the details for five cases. In order to improve readability of the table, we have adopted a color/font scheme: Rows in normal font with no color indicate cases which are explicitly allowed based on the 3GPP specifications (e.g., the case with ID 1). Green color and bold font indicates uncertain cases (e.g., cases 2 and 3). Grey color and italic font indicates cases which are ruled out by the specifications (e.g., case 4). The cases involving only 2G/3G components are marked with blue color and in bold italic font (e.g., case 122). There are 19 such cases which are not further detailed in this paper as they have been analyzed previously [6], [16]. For the disallowed and uncertain cases the table includes the details for the reasoning to determine the respective classification.

[2] In LTE AKA, the nonce is never used. So the first block is always LTE I. LTE I' will be used in one interoperation scenario (S8).

TABLE I. Excerpt of classifying the 243 interoperation cases (the full table is in [7])

ID	Components					Condition to support Occurrence	Reasons for Disallowance
	Identity Module	ME	BS	VLR/SGSN /MME	HN		
1	4G	4G	4G	4G	4G		
2			**3G**	**4G**	**4G**	A1	
3			**2G**	**4G**	**4G**	A1	
4			*4G*	*3G*	*4G*		R5
...	...	...	...	...	...		
122	***3G***	***3G***	***3G***	***3G***	***3G***		
...	...	...	...	...	...		

Allowed Interoperation Cases. For cases involving a mix of 4G, 3G, and 2G network components, we have identified 38 cases which are explicitly allowed by the 3GPP specifications.

For the identity module, the SIM supports 2G AKA only [6]. A USIM supports both 2G and 3G AKA [6]. Similarly, a 4G USIM supports 2G, 3G, and 4G AKA [30]. Since a large number of USIMs is in current use, a 4G ME with the USIM is allowed to access the 4G network. Since the 4G ME is capable of deriving LTE keys and storing security contexts [31], the combination of a 4G ME with a USIM supports 4G AKA.

For the ME, it is possible to use a SIM or a USIM with a 2G ME [6]. Since the 4G USIM is an enhanced version of the USIM, this implies that it is possible to also use a 4G USIM with a 2G ME. Similarly, since a SIM or USIM can be used with a 3G ME, [6], it is also possible to use a 4G USIM with a 3G ME. A 4G ME can be used with a SIM [30] or USIM [4] and certainly with a 4G USIM. A 2G ME only supports GERAN [6]. A 3G ME supports GERAN and UTRAN [6], and a 4G ME supports GERAN, UTRAN, and E-UTRAN [30].

For the BS, a 2G BS is only capable of handling a GSM session key *Kc* [6], which means a 2G BS only supports a 2G ciphering mode setting [29]. Similarly, a 3G BS requires the UMTS cipher key *CK* and the UMTS integrity key *IK* [6]—supporting only the 3G security mode set-up and operation [3]. A 4G BS requires K_{eNB} and only supports the 4G AS security mode command procedure and operation [4].

For the VLR/SGSN/MME, a 2G VLR/SGSN can only control a 2G BS and only supports 2G AKA [6]. A 3G VLR/SGSN can control both a 2G BS and a 3G BS and can support both 2G and 3G AKA [6]. An MME can control a 4G BS and can support the 4G AKA [4].

For the HN, a 2G HN can maintain 2G and 3G subscriptions [6]. A 3G HN can maintain 2G and 3G subscriptions [6]. A 4G HN can maintain 3G and 4G subscriptions [4].

Disallowed Interoperation Cases. We found that the 3GPP specifications rule out 138 cases which include some 4G components. Table I refers to the following reasons for disallowing various cases:

R1 Use of SIMs to access the 4G network is not allowed [4].
R2 A 2G ME cannot interoperate with a 3G or 4G BS [6].
R3 A 3G ME does not support the 4G radio access interface [6].
R4 A 2G VLR/SGSN cannot control a 3G or 4G BS [6].
R5 A 3G VLR/SGSN cannot control a 4G BS [6].
R6 An MME refuses to convert a GSM security context to 4G security context. Consequently, this rules out all interoperation cases which would require the deriving of the master key K_{ASME} from the GSM cipher key K_c [4].
R7 A 3G ME with USIM attaching to a 3G BS shall only participate in 3G AKA and shall not participate in 2G AKA [3]. This rules out the case in which a USIM subscribed to a 2G HN is used in a 3G ME that connects to a 3G BS, as the 2G HN can only support 2G AKA.

Uncertain Interoperation Cases. The remaining 48 cases involving 4G components are classified as uncertain. This is due to the fact that the specifications do not provide clear indication as to whether or not these cases are allowable. For those cases, Table I refers to these conditions under which they could occur:

A1 An MME can control a 3G BS or a 2G BS.
A2 A 3G HN or 2G HN can maintain 4G subscriptions.
A3 A 4G HN can maintain 2G subscriptions.
A4 An MME can support the 2G or 3G AKA.

V. AKA SCENARIOS

Focusing on the allowable and uncertain interoperation cases determined in the previous section, we now detail the respective AKA for each of these cases. Specifically, based on the 3GPP specifications for GSM, UMTS, and LTE we have determined which of the building blocks GSM I–IV, UMTS I–IV, and LTE I–V need to be combined in what fashion to comprise a suitable AKA for the respective interoperation case. For each case we provide the rationale based on which the building blocks are combined.

Overall, this approach allowed us to categorize all allowable and uncertain interoperation cases into 10 distinct scenarios. For five of the scenarios, the respective AKA was already specified by 3GPP in the context of enabling interoperation between GSM and UMTS (including the two native 2G and 3G scenarios as outlined in Sect. III). One of the remaining five scenarios is the native 4G AKA (see Sect. III). To the best of our knowledge, the other four are new and are specified for the first time in this paper.

Determining the Scenarios. In order to determine a suitable AKA for a specific interoperation case, first we consider which message might trigger the AKA to determine the messages transmitted in the first block. Then, we consider the authentication vector that is generated in the HN and subsequently provided to the VLR/SGSN/MME. In particular, the authentication vector determines what kind of challenge-response procedure is carried out, i.e., whether GSM III, UMTS III, or LTE III. How and what kind of authentication vector is generated by the HN depends on HN's capabilities, the type of VLR/SGSN/MME requesting/receiving the authentication vector, the type of BS, and the type of the identity module. Table II provides the details for the eleven distinct instances for obtaining an authentication vector. While the first six (A, B, C, D, E, F) are based on methods described in the 3GPP specifications (mostly w.r.t. security context switching and mapping in LTE), the latter ones are interpretations derived from specified methods. Second, we consider the type of the BS, which determines the type of the security mode setup procedure. Third, depending on the type of BS it controls, the VLR/SGSN/MME might have to convert the encryption/integrity keys.

TABLE III. EXCERPT OF DETERMINING AKA SCENARIOS AND RESPECTIVE REASONING (FULL TABLE IN [7])

ID	Components					Scenario	Reason	
	Identity Module	ME	BS	VLR/ SGSN/ MME	HN		Stated in spec	Interpre- tation
1	4G	4G	4G	4G	4G	S3	AGKW	
2			**3G**	**4G**	**4G**	S10	BHV	T
3			**2G**	**4G**	**4G**	S9	BINV	T
...	...	...	...	...	...	...	...	...
91	3G	4G	4G	4G	HN	S7,S8	GKWX	QU
...	...	...	...	...	...	...	...	...

Using this approach, we categorize the 105 allowable and uncertain interoperation cases into 10 distinct scenarios:

S1 GSM I–IV.
S2 UMTS I–IV.
S3 LTE I–V.
S4 GSM I–IV, conv(3G AV → 2G AV).
S5 GSM I–III || UMTS IV, conv($Kc \rightarrow CK\ IK$, VLR/SGSN).
S6 UMTS I–III || GSM IV, conv($CK\ IK \rightarrow Kc$, VLR/SGSN).
S7 LTE I || UMTS II–III || [optionally, LTE IV] || LTE V, conv($CK\ IK \rightarrow K_{ASME}$, MME).
S8 LTE I' || UMTS II–III || LTE IV–V, conv($CK\ IK\ nonces \rightarrow K_{ASME}$, MME).
S9 GSM I || LTE II–III || [optionally, LTE IV] || GSM IV, conv($CK\ IK \rightarrow Kc$, MME), AV = 4G AV + CK + IK.
S10 UMTS I || LTE II–III || [optionally, LTE IV] || UMTS IV, AV = 4G AV + CK + IK.

The blocks are as introduced in Figs. 1, 2, and 3. With notation "a || b" we indicate that block b follows after block a. Scenarios S7, S9, and S10 have blocks marked in brackets as optional. It is consistent with the specifications to either include or omit these blocks, so we analyze versions with and without the block. The notation "conv($K1 \rightarrow K2$, C)" denotes that network component C converts key $K1$ into $K2$. Furthermore, "AV = 4G AV + CK + IK" indicates that the 4G HN provides not only the 4G authentication vector to the MME but also includes the UMTS encryption key CK and integrity key IK.

Table III shows an excerpt of determining and categorizing the AKA for all of the 105 allowed and uncertain cases—including the respective reasoning to obtain the categorization (with reference to Table II).

In the following we focus on detailing Scenarios S7–S10. Scenarios S1–S3 are discussed in Sect. III and Scenarios S4–S6 coincide with scenarios described in the 3GPP specifications as well as prior work [1], [2], [15], [16].

TABLE II. REASONS USED FOR DETERMINING AKA SCENARIOS

Type of AV	A	Upon request by an MME with network type equals E-UTRAN, the 4G HN generates and delivers the 4G AVs (separation bit = 1) [4].
	B	Upon request by an MME with network type equals UTRAN or GERAN, the 4G HN generates and delivers the 4G AVs, plus CK and IK (separation bit = 0) [4].
	C	Upon request by a 3G VLR/SGSN, the 3G HN generates and sends out 3G AVs [3].
	D	Upon request by a 2G VLR/SGSN with a 3G IMSI, the 3G HN generates 2G AVs from 3G AVs [6].
	E	2G HN only supports to generate 2G AVs [6].
	F	Upon request by a VLR/SGSN/MME with a 2G IMSI, the 3G HN always generates and delivers the 2G AVs [6].
	O	Upon request by a 3G VLR/SGSN, the 4G HN generates 3G AVs [4] [or derived from D].
	P	Upon request by a 2G VLR/SGSN with 3G/4G IMSI, the 4G HN generates 2G AVs from UMTS AVs [Derived from D].
	Q	Upon request by an MME, the 3G HN generates 3G AVs [Derived from E and [6]].
	R	Upon request by a 2G VLR/SGSN with a 4G IMSI, the 3G HN generates 2G AVs from 3G AVs [Derived from D].
	S	Upon request by a VLR/SGSN/MME with a 2G IMSI, the 4G HN always generates and delivers the 2G AVs [Derived from F].
Type of BS	G	4G BS only supports 4G SMC [4].
	H	3G BS only supports 3G SMC [6].
	I	2G BS only supports 2G SMC [6].
Type of ME	J	The 4G ME supports to derive K_{ASME} and store the security contexts. [31]
	K	XG ME supports XG SMC [4], [3]
	L	3G ME supports 2G SMC [6].
	T	4G ME supports 2G/3G SMC [Derived from L]
Conversion	M	The 3G BS requires CK and IK, the VLR/SGSN/MME generates them from Kc by applying conversion function c3 [6].
	N	2G BS is not capable of handling of cipher and integrity keys. The VLR/SGSN/MME converts the CK and IK into Kc [6].
	U	Because the 4G BS requires K_{eNB}, which is derived from the K_{ASME}, the VLR/SGSN/MME generates K_{ASME} from the CK, IK and sends it to the BS [Derived from M or [4]].
First Block	V	Triggered by attach request or RAU request, the first block is GSM I or UMTS I [27], [28]
	W	Triggered by LTE attach request or TAU request in which the nonce is never used, the first block is LTE I [4].
	X	Triggered by TAU request and the nonce is used in latter blocks, the first block is LTE I' [28], [30].

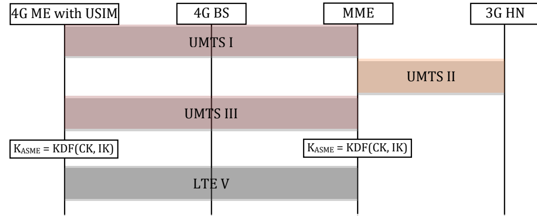

Fig. 4. AKA scenario S7; in alternate version, LTE IV added before LTE V

S7. This scenario is characterized by a 4G ME, a 4G SN (i.e., 4G BS and 4G MME) and a USIM or 4G USIM identity module subscribed to a 3G HN. Two of the allowable/uncertain cases fall into this category. The AKA is triggered by the attach request. The identity request and response procedure is the same as in LTE I (Fig. 3). The 3G HN can generate 2G or 3G authentication vectors, but cannot generate 4G authentication vectors. Upon request by the MME, the 3G HN therefore generates and delivers a 3G authentication vector which thus is identical to UMTS II. Upon receiving the authentication vector, the MME communicates with the MS as in UMTS III. The 4G BS requires 4G AS keys, which are derived from the intermediate key K_{eNB}. Because the intermediate key K_{eNB} is derived from the local master key K_{ASME}, the MME applies a key derivation function to generate the local master key K_{ASME} from the UMTS encryption key CK and integrity key IK. Including LTE IV is optional in this scenario. Later, we analyze both variations and show that the AKA without LTE IV is prone to an attack in which a false base station can both eavesdrop and modify the messages between the MS and the SN. Executing both LTE IV and V prevents this attack. LTE V (Fig. 3) is executed which includes the deriving of K_{eNB}. Fig. 4 shows this AKA scenario without LTE IV.

S8. This scenario is characterized by the same cases as in S7. The difference to S7 is that this scenario is triggered by the TAU request and the $NONCE_{MS}$ in the TAU request is used in LTE IV. The retrieving and generating of the authentication vector and the challenge-response procedure are the same as in S7. In LTE IV, the MME generates a nonce $NONCE_{MME}$ and uses it with the CK, IK, and $NONCE_{MS}$ as input parameters to derive the K_{ASME}. The MME sends the integrity protected SMC message containing the nonce $NONCE_{MS}$ received from the MS and the nonce $NONCE_{MME}$. Upon receiving the SMC message, the MS checks whether the $NONCE_{MS}$ and the capabilities match what it originally sent in LTE I'. If the check successes, the MS uses the same key derivation function as in the MME to derive the K_{ASME} and sends out the SMC complete message. Subsequently, LTE V is executed.

S9. This scenario is characterized by a mixed SN including a 2G BS and a 4G MME as well as an MS that is subscribed to a 4G HN where either the identity module is a 4G USIM or it is a 4G ME, i.e., the MS supports 4G AKA. Four of the allowable/uncertain cases fall into this category. Because the 2G BS covers routing areas, the initial message can be the attach request or the RAU request. So the transmitting of the identity and the capabilities is as in GSM I (Fig. 1). When the MME requests the authentication vector from the HN by sending the IMSI and the network type, because the network type is GERAN (because of the 2G BS), the 4G HN generates and delivers the 4G authentication vector with the UMTS cipher key CK and integrity key IK. Because the NAS signaling is transparent to the BS, the LTE challenge-response procedure LTE III (Fig. 3) is executed between the MS and the MME. In this interoperation scenario, we consider the two variations with and without LTE IV (Fig. 3). The first variation sticks to the LTE AKA as long as possible (i.e., until executing LTE IV before setting up the cipher between the MS and the BS). The other one goes to set the cipher between MS and the BS as soon as finishing the challenge-response procedure (without executing LTE IV). Later we show that the AKA without LTE IV is prone to a false base station attack and the AKA with LTE IV is prone to an attack against the CMC message between the 2G BS and the MS. Because the 2G BS requires the GSM session key Kc, the MME derives the encryption key K_c from the UMTS cipher key CK and integrity key IK. Since only the 2G cipher mode setting is supported by the 2G BS, the 2G cipher mode setting procedure GSM IV (Fig. 1) is executed between the 2G BS and the MS—which

also includes the MME sending the GSM session key to the 2G BS.

S10. This scenario is characterized by a 4G HN, a mixed SN consisting of a 3G BS and a 4G MME, as well as an MS that is subscribed to a 4G HN where either the identity module is a 4G USIM or it is a 4G ME with a 3G USIM, i.e., the MS supports 4G AKA. Three of the allowable/uncertain cases fall into this category. The 3G BS covers routing areas, so the initial message can be the attach request or a RAU request. Thus, the transmitting of identity and capabilities is as in UMTS I (Fig. 2). In order to obtain an authentication vector, the MME sends the authentication data request with the IMSI and the network type to the 4G HN. Because the access network is UTRAN, the 4G HN generates and delivers the 4G authentication vector as well as the UMTS encryption key CK and integrity key IK. Subsequently, the LTE challenge-response procedure LTE III (Fig. 3) is executed between the MS and the MME. As in scenarios S7 and S9, the LTE IV is optional. Later we show that the authentication properties hold in both variations. The 3G BS obtains the UMTS encryption key CK, the UMTS integrity key IK, as well as the capabilities as part of UMTS IV (Fig. 2)—which also includes the UMTS security mode set-up procedure between the 3G BS and the MS.

VI. Modeling and analyzing the pure protocols in ProVerif

The ProVerif (PV) tool has been described well elsewhere, and we use standard idioms in our modeling. We give here a brief overview of our design decisions followed by a few details concerning the LTE model. Details of the GSM and UMTS models can be found in [16]. The roaming models are discussed in Sect. VII, together with our analysis results. The complete models are available with the long version of the paper [7].

In PV, protocols are defined using process algebra. Properties are specified as correspondence assertions [32] that refer to *events*. Events are instrumentation that mark important points reached by the principals and have no effect on protocol behavior. For example, the correspondence assertion **event**(e1(M)) ⇝**event**(e2(M)) says that if event e1 occurs, with argument value M, then event e2 must have happened previously with the same argument M. In checking an assertion, PV may terminate having successfully proved the property, with respect to unbounded message space and number of sessions, or having found a possible or definite attack.

Here are some design decisions that apply to all of our models. Each message has a header to indicate the type of the message content. The secure communications between SN and HN are modeled as private channels. Registration of the MS, i.e., pre-sharing of each long-term credential pair (*IMSI*, *Ki*), is modeled using PV's table construct. We do not model details of algorithm capabilities/selection. The capability is a nondeterministically chosen boolean value interpreted to mean whether the MS has encryption capability. (Integrity protection is mandatory in 3G/4G, and absent in 2G.) Because the value is nondeterministically chosen, our analysis considers all cases. Following authentication, a single data message is included, which suffices to specify the secrecy of data traffic. In the long version of the paper we consider integrity of data traffic, which is also specified using this message.

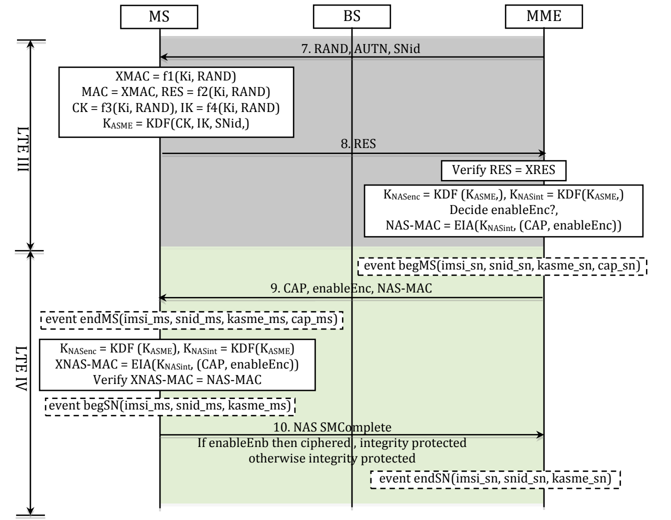

Fig. 5. Part of the 4G AKA scenario (Fig. 3) annotated in accord with our model

The 4G model in ProVerif. There are four main processes in our PV model, representing the behavior of the MS, the eNB, the MME and the HN respectively. Fig. 5 shows the details of part of the model, specifically Blocks III and IV from Fig. 3. Fig. 5 shows the events for correspondence assertions. It also shows the name of the variables which are used in our model, which facilitates checking that the models accurately reflect the protocol diagrams.

In the LTE protocol, the MS already has the SN id before starting the AKA shown in Fig. 3. In our model, we add the SN id to the authentication challenge (message 7). In addition,our model omits sequence numbering and the key AK, so they do not appear in Fig. 5. Sequence numbers aid in preventing re-use of authentication vectors. Instead of modeling sequence numbers, our models simply do not re-use AVs.

Fig. 6 shows the code of the MS process. The registration process of the MS device is in lines 22–24. Lines 28–29 model that the MS receives and checks the authentication challenge message. The process awaits a message on the public channel, with designated format and particular values: the format must be (msgHdr, nonce, mac, ident) where msgHdr is the literal CHALLENGE and the mac must equal f1(ki, rand_ms). In lines 38–39, the MS receives the NAS SMC message and verifies the integrity and the received capabilities. The MS then sends out the SMC complete message which is ciphered if the encryption is enabled and integrity protected (lines 43–51). Lines 46 and 52 call a parameterized process which specifies the AS SMC procedure in lines 3–11 and receives the data message in lines 18–19. The other three processes representing the BS, the MME, and the HN are similar to this (see [7]).

The secrecy and authentication properties are specified as follows.

```
1 query attacker(payload).
2 query attacker(payload) ⇝ event(disableEnc).
3 query attacker(secret).
4 query x1: ident, x2: ident, x3: asmeKey;
5   event(endSN(x1, x2, x3)) ⇝ event(begSN(x1, x2, x3)).
6 query x1: ident, x2: ident, x3: asmeKey, x4: bool;
```

```
1  let pMSAS(kasme_ms: asmeKey, imsi_ms: ident, cap_ms: bool) =
2    let kenb_ms: enbKey = kdf_enb(kasme_ms) in
3    let kasenc_ms: asEncKey = kdf_as_enc(kenb_ms) in
4    let kasint_ms: asIntKey = kdf_as_int(kenb_ms) in
5    let kupenc_ms: upEncKey = kdf_up_enc(kenb_ms) in
6    in(pubChannel, (=ASSMC, enableEnc_as_ms: bool,
7      =finteg_as(bool2bitstring(enableEnc_as_ms), kasint_ms)));
8    event begENB(imsi_ms, kenb_ms);
9    out(pubChannel, (ASSMComplete, as_smcomplete_msg,
10     finteg_as(as_smcomplete_msg, kasint_ms)));
11   event endMS_ENB(imsi_ms, kenb_ms, cap_ms);
12   in(pubChannel, (=MSG, datamsg: bitstring,
13     =finteg_as(datamsg, kasint_ms)));
14   out(pubChannel, sencrypt_as(secret, kasenc_ms));
15   out(pubChannel, senc_int_as(secret, kasint_ms));
16   out(pubChannel, senc_up(secret, kupenc_ms));
17   if enableEnc_as_ms = true then
18     let msgcontent: bitstring = sdecrypt_as(datamsg,
19     kasenc_ms) in 0.
20
21 let processMS =
22   new imsi_ms: ident;
23   new ki: key;
24   insert keys(imsi_ms, ki);
25   let cap_ms: bool = encCapability() in
26   out(pubChannel, (CAP, cap_ms));
27   out(pubChannel, (ID, imsi_ms));
28   in(pubChannel, (=CHALLENGE, rand_ms: nonce,
29     =f1(ki, rand_ms), snid_ms: ident));
30   let res_ms: resp = f2(ki, rand_ms) in
31   let ck_ms: cipherKey = f3(ki, rand_ms) in
32   let ik_ms: integKey = f4(ki, rand_ms) in
33   let kasme_ms: asmeKey = kdf_asme(ck_ms, ik_ms, snid_ms) in
34   event begSN(imsi_ms, snid_ms, kasme_ms);
35   out(pubChannel, (RES, res_ms));
36   let knasenc_ms = kdf_nas_enc(kasme_ms) in
37   let knasint_ms = kdf_nas_int(kasme_ms) in
38   in(pubChannel, (=NASSMC, enableEnc_nas_ms: bool, =cap_ms,
39     =finteg_nas((enableEnc_nas_ms, cap_ms), knasint_ms)));
40   event endMS(imsi_ms, snid_ms, kasme_ms, cap_ms);
41   out(pubChannel, sencrypt_nas(secret, knasenc_ms));
42   out(pubChannel, senc_int_nas(secret, knasint_ms));
43   if enableEnc_nas_ms = false then
44     out(pubChannel, (NASSMComplete, nas_smcomplete_msg,
45       finteg_nas(nas_smcomplete_msg, knasint_ms)));
46     pMSAS(kasme_ms, imsi_ms, cap_ms)
47   else
48     out(pubChannel,
49       (NASSMComplete, sencrypt_nas(nas_smcomplete_msg,
50       knasenc_ms), finteg_nas(sencrypt_nas(
51       nas_smcomplete_msg, knasenc_ms), knasint_ms)));
52     pMSAS(kasme_ms, imsi_ms, cap_ms).
```

Fig. 6. MS process for LTE

```
7    event(endMS(x1, x2, x3, x4)) ⇝
   event(begMS(x1, x2, x3, x4)).
8  query x1: ident, x2: enbKey, x3: bool;
9    event(endMS_ENB(x1, x2, x3)) ⇝
   event(begMS_ENB(x1, x2, x3)).
10 query x1: ident, x2: enbKey; event(endENB(x1, x2)) ⇝
   event(begENB(x1, x2)).
```

The payload can be learned by the attacker when the MS is not capable of encryption, and indeed PV finds violations of the secrecy property in query line 1. Conditional secrecy, query line 2, says that if the attacker obtains the secret payload then the event disableEnc must have previously taken place —this is proved by PV. To test the secrecy of the keys, the MS encrypts a fresh secret (a private free name in the code, not shown in Fig. 6) under each of the keys and sends the ciphertexts on the public channel (lines 43–44 and 17–19), and query line 3 tests the secrecy. PV proves conditional secrecy and key secrecy.

Authentication of the MS to the MME is specified in query lines 4–5. This refers to event endSN placed following message 10 (Fig. 5) so that it follows both verification of the challenge and successful use of the keys derived from K_{ASME}. Authentication of the MME to MS is specified in query lines 6–7; it includes authenticity of the security capabilities. The authentication of the eNB to the MS is specified in query lines 8–9. The encryption capability is included in the parameters of the events to specify that the events should agree on the encryption option. These authentication properties are proved successfully.

Because communication between eNB and MME is assumed secure, it is authentication of MS to MME implies authentication to eNB as well. However, as a sanity check on the model, query line 10 says that if eNB believes that it has established the K_{eNB} associated with an MS using the particular IMSI, then indeed there is an MS that reached that stage of its protocol role, for that IMSI and K_{eNB}.

VII. Modeling and Analyzing Interoperation in ProVerif

In Sect. III we annotate the protocol diagrams to mark "blocks" of message exchanges, which are composed to form the interoperation scenarios in Sect. V. Where it is convenient, we use sub-processes in our PV models to express this structure. To make the PV model for an interoperation scenario, we can easily combine these sub-processes and other code fragments that correspond to blocks, with minor modifications (adding conversion functions that enable a BS to perform a particular SMC procedure, and adding keys to the AV in S9 and S10).

For example, for the MS in LTE we factor out a process processMS that models the first four blocks of LTE. This process is reused in the models for scenarios S9 and S10. If the assumptions that underly our scenarios for uncertain cases turn out to be wrong, we expect to be able to easily model the corrected scenarios as well. For security specifications, the blocks already include events and the queries are easily adapted from queries for the pure protocols.

Of the 10 AKA scenarios, 5 of them are the same scenarios as in the roaming cases of GSM and UMTS, for which the models and analysis appears in [16]. One of them is the pure 4G AKA, which is modeled and analyzed in Sect. VI. In this section, we discuss the models of the 4 new scenarios, and then summarize results for all 10 scenarios.

Scenario S7: LTE I || UMTS II–III || [LTE IV] || LTE V, conv($CK\ IK \rightarrow K_{ASME}$, MME). Fig. 7 elaborates the scenario in Fig. 4 with details of the PV model and shows the locations of the events which are used to specify authentication and conditional payload secrecy. Most of the code in this model is inherited from the 4G model and the UMTS model. The secrecy and authentication properties are specified similar to the ones in the 4G model:

```
1 query attacker(payload) ⇝ event(disableEnc).
2 query attacker(secret).
3 query x1: ident, x2: cipherKey, x3: integKey;
4   event(endSN(x1, x2, x3)) ⇝ event(begSN(x1, x2, x3)).
5 query x1: ident, x2: enbKey, x3: bool;
6   event(endMS(x1, x2, x3)) ⇝ event(begMS(x1, x2, x3)).
```

As in pure 4G, plain secrecy of the message payload does not hold because the attacker can always learn the payload if the MS is not capable of encryption. Conditional secrecy (query

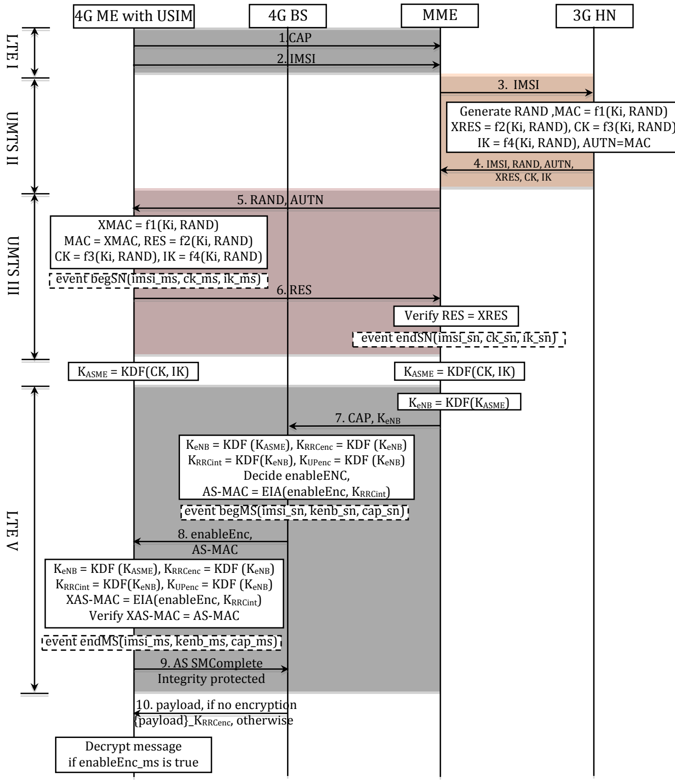

Fig. 7. Authentication scenario S7, version without LTE IV, annotated in accord with our model

TABLE IV. ANALYSIS RESULTS

Scenario	Conditional secrecy	Key secrecy	Auth. of MS to VLR/ SGSN/MME	Auth. of VLR/ SGSN/MME to MS	Auth. of BS to MS
S1	Proved	Proved	Proved	N/A	known false base station attack
S2	Proved	Proved	Proved	N/A	Proved
S3	Proved	Proved	Proved	Proved	Proved
S4	Proved	Proved	Proved	N/A	known false base station attack
S5	Proved	Proved	Proved	N/A	Proved
S6	Proved	Proved	Proved	N/A	known false base station attack
S7 w/o LTE IV	Proved	Proved	Proved	N/A	false base station attack
S7 w/ LTE IV	Proved	Proved	Proved	Proved	Proved
S8	Proved	Proved	Proved	Proved	Proved
S9 w/ LTE IV	Proved	Proved	Proved	Proved	CMC attack
S9 w/o LTE IV	Proved	Proved	Proved	N/A	known false base station attack
S10 w/ LTE IV	Proved	Proved	Proved	Proved	Proved
S10 w/o LTE IV	Proved	Proved	Proved	N/A	Proved

line 1) does hold. Secrecy of keys (line 2) is also proved. Authentication of the MS to the SN is specified in lines 3–4 and is proved. Authentication of the SN to the MS is specified in lines 5–6. For the version without LTE IV, PV finds an attack that violates the property. The attacker intercepts the capability message sent by the MS and replaces the capabilities with different ones. The event endMS is executed after the MS receives the SMC message. Because the SMC message does not contain the received MS's capabilities, the MS has no way to confirm whether the SN receives the correct capabilities. PV detects the violation because, although there was a begMS event, it has a different value for capabilities. For the version with LTE IV, the property is proved.

Scenario S8: LTE I' || UMTS II–III || LTE IV–V, conv($CK\ IK\ nonces \rightarrow K_{ASME}$**, MME).** ProVerif proves all the properties except the payload secrecy.

Scenario S9: GSM I || LTE II–III || [LTE IV] || GSM IV, conv($CK\ IK \rightarrow Kc$**, MME), AV = 4G AV + CK + IK.** In the models (with or without LTE IV) of this scenario, the MME uses the key conversion function **fun** c3(cipherKey, integKey): gsmKey to derive the GSM session key from the UMTS cipher and integrity keys. Because the BS is the GSM BS, the false base station attack on the AKA without LTE IV is found when checking the authentication of the BS to the MS. In the model with LTE IV, an attack is found when checking the authentication of the BS to the MS. In the attack, the attacker modifies the CMC message (which is not integrity protected) to tell the MS to use no encryption. This attack will be detected by the BS once the MS sends messages to the BS. As in other scenarios, the payload secrecy could be violated because the BS could choose to disable encryption when communicating with the MS.

Scenario S10: UMTS I || LTE II–III || [LTE IV] || UMTS IV, AV = 4G AV + CK + IK. ProVerif proves all the properties except the payload secrecy.

Analysis Results. Table IV gives results for all the 10 scenarios. In the model of scenario S9 without LTE IV, we find the known false base station attack [1] which has the same attack scenario as in the native GSM AKA, i.e., in S1 and S4. In this attack, the attacker intercepts the CAP message and modifies the capabilities of the MS as no-encryption. When the BS decides which algorithm to use, the BS has to choose not to enable encryption. Because the subsequent traffic between the MS and the 2G BS is not encrypted nor integrity protected, the attacker can both eavesdrop and modify the messages.

The attack found in scenario S7 without LTE IV is similar. The attacker intercepts and modifies the capabilities of MS to no-encryption to force the 4G BS to choose not to use encryption. Although integrity protection is mandatory for the signaling traffic of 4G BS, there is no integrity protection on the user plane traffic, so the attack can both eavesdrop and modify the data traffic.

In the model of scenario S9 with LTE IV, we find an attack in which the attacker simply modifies the CMC message to tell the MS to use no encryption. This attack would be detected once the MS sends unencrypted messages to the BS.

VIII. CONCLUSION

In this paper we study authentication and key agreement (AKA) for interoperation among GSM, UMTS and LTE. To determine the AKA procedures in each interoperation case, we consider all combinations of the five relevant system

components. We classify some cases as allowed or disallowed, based on information about component compatibility gleaned from the standards documents. Some cases are classified as uncertain, for lack of definite information in the standards. For each possible (allowed or uncertain) interoperation case, we identify and justify a particular AKA scenario built from elements ("blocks") of the pure GSM, UMTS, and LTE protocols.

It turns out that 10 scenarios are needed to cover all the 105 possible interoperation cases. Of these scenarios, 5 involve just GSM and UMTS and were identified previously (see Sect. II); one is the pure LTE; the remaining 4 are new. In most cases, the AKA scenario is completely determined by the components involved. However, a few cases have two feasible versions of the scenarios which differ by whether block LTE IV is included or whether the nonce in TAU request is used.

We model and analyze pure LTE and the 4 new AKA scenarios involving LTE components, using ProVerif, focusing in this paper on authentication and secrecy properties. For the scenarios involving LTE, we find three attacks. One is the false base station attack which is inherited from the GSM system and is also found in GSM-UMTS interoperation. Another attack, on one version of scenario S7, is a similar false base station attack but with a 4G BS. The attack is prevented by including block LTE IV. In the third attack on the AKA of scenario S9 with LTE IV, the CMC message is modified. Aside from these attacks, the desired authentication and secrecy properties are proved (in the symbolic model of perfect crypto, with unbounded sessions) for all other cases.

For further work, we would like to analyze the handover in GSM, UMTS, and LTE, as well as across the technologies. We also are interested in exploring the interworking between 4G and non-3GPP networks.

References

[1] U. Meyer and S. Wetzel, "A man-in-the-middle attack on UMTS," in *ACM WiSec*, 2004, pp. 90–97.

[2] ——, "On the impact of GSM encryption and man-in-the-middle attacks on the security of interoperating GSM/UMTS networks," in *IEEE Symposium on Personal, Indoor and Mobile Radio Communications*, 2004.

[3] "3GPP TS 33.102 version 11.4.0 Release 11 Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; 3G security; Security architecture ," http://www.3gpp.org/ftp/Specs/html-info/33102.htm.

[4] "3GPP TS 33.401 v10.0.0; Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; 3gpp System Architecture Evolution (SAE); Security architecture," http://www.3gpp.org/ftp/Specs/html-info/33401.htm.

[5] B. Blanchet, "Automatic verification of correspondences for security protocols," *Journal of Computer Security*, vol. 17, no. 4, pp. 363–434, Jul. 2009.

[6] "3GPP TR 31.900 v11.0.0 Release 11; Universal Mobile Telecommunications System (UMTS); LTE; SIM/USIM internal and external interworking aspects," http://www.3gpp.org/ftp/Specs/html-info/31900.htm.

[7] C. Tang, D. A. Naumann, and S. Wetzel, "Analysis of authentication and key establishment in inter-generational mobile telephony (long version)," 2013, http://www.cs.stevens.edu/~naumann/pub/TangNaumannWetzel2013.pdf.

[8] J. D. Golic, "Cryptanalysis of alleged a5 stream cipher," in *EUROCRYPT*, 1997.

[9] L. G. I. Goldberg, D. Wagner, "The real-time cryptanalysis of A5/2," Rumpsession of CRYPTO, 1999.

[10] S. Petrovic and A. Fster-Sabater, "Cryptanalysis of the A5/2 algorithm," Cryptology ePrint Archive, Report 2000/052, 2000, http://eprint.iacr.org/.

[11] E. Barkan, E. Biham, and N. Keller, "Instant ciphertext-only cryptanalysis of GSM encrypted communication," *J. Cryptol.*, vol. 21, pp. 392–429, March 2008.

[12] O. Dunkelman, N. Keller, and A. Shamir, "A practical-time related-key attack on the KASUMI cryptosystem used in GSM and 3G telephony," in *CRYPTO*, 2010.

[13] Z. Ahmadian, S. Salimi, and A. Salahi, "New attacks on UMTS network access," in *Wireless Telecommunications Symposium, 2009. WTS 2009*. IEEE, 2009, pp. 1–6.

[14] D. Fox, "Der IMSI catcher," in *DuD Datenschutz und Datensicherheit*, 2002.

[15] U. Meyer, "Secure roaming and handover procedures in wireless access networks," Ph.D. dissertation, Darmstadt University of Technology, Germany, 2005.

[16] C. Tang, D. A. Naumann, and S. Wetzel, "Symbolic analysis for security of roaming protocols in mobile networks," in *SecureComm 2011 : Seventh International ICST Conference on Security and Privacy in Communication Networks*, 2011.

[17] R. Chang and V. Shmatikov, "Formal analysis of authentication in Bluetooth device pairing," in *FCS-ARSPA*, 2007.

[18] M. Jakobsson and S. Wetzel, "Security weaknesses in Bluetooth," in *Cryptographer's Track at the RSA Conference (CT-RSA)*, 2001, pp. 176–191.

[19] B. Blanchet and A. Chaudhuri, "Automated formal analysis of a protocol for secure file sharing on untrusted storage," in *IEEE Symp. on Sec. and Priv.*, 2008.

[20] L. Chen and M. Ryan, "Attack, solution and verification for shared authorisation data in TCG TPM," in *Formal Aspects in Security and Trust*, 2009, pp. 201–216.

[21] S. Kremer and M. Ryan, "Analysis of an electronic voting protocol in the applied pi calculus," in *ESOP*, 2005, pp. 186–200.

[22] M. Arapinis, L. Mancini, E. Ritter, M. Ryan, N. Golde, K. Redon, and R. Borgaonkar, "New privacy issues in mobile telephony: fix and verification," in *ACM CCS*. ACM, 2012, pp. 205–216.

[23] C. Han and H. Choi, "Security analysis of handover key management in 4G LTE/SAE network," 2012.

[24] J.-K. Tsay and S. F. Mjølsnes, "A vulnerability in the UMTS and LTE authentication and key agreement protocols," in *Computer Network Security*. Springer, 2012, pp. 65–76.

[25] M.-F. Lee, N. P. Smart, B. Warinschi, and G. J. Watson, "Anonymity guarantees of the UMTS/LTE authentication and connection protocol," *IACR Cryptology ePrint Archive*, vol. 2013, p. 27.

[26] M. A. Mobarhan, M. A. Mobarhan, and A. Shahbahrami, "Evaluation of security attacks on UMTS authentication mechanism," *International Journal of Network Security & Its Applications*, vol. 4, no. 4, 2012.

[27] "TS 23.060 version 11.5.0 Release 11; Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); General Packet Radio Service (GPRS); Service description; Stage 2," http://www.3gpp.org/ftp/Specs/html-info/23060.htm.

[28] "3GPP TS 23.401 version 11.4.0 Release 11; LTE; General Packet Radio Service (GPRS) enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) access," http://www.3gpp.org/ftp/Specs/html-info/23401.htm.

[29] "Digital cellular telecommunications system (Phase 2+); Mobile radio interface layer 3 specification (GSM 04.08 version 7.8.0 Release 1998)," 1998.

[30] D. Forsberg, G. Horn, W.-D. Moeller, and V. Niemi, *LTE Security*. John Wiley and Sons, Ltd, 2010.

[31] "3GPP TS 24.301 version 11.4.0 Release 11; Universal Mobile Telecommunications System (UMTS); LTE; Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3," http://www.3gpp.org/ftp/Specs/html-info/24301.htm.

[32] A. D. Gordon and A. Jeffrey, "Authenticity by typing for security protocols." *Journal of Computer Security*, pp. 451–520, 2003.

Appendices

Sect. A gives the complete table of cases and Sect. B gives the complete table of scenarios. Sect. C presents the models of the scenarios and Sect. D gives the complete code.

Appendix A
Table of Cases

Figures 8 – 14 show the classification the 243 interoperation cases. The table makes reference to the list of reasons R1–R7 in Sect. IV and the list of conditions A1–A4 at the end of Sect. IV.

As stated in the main body of the paper, rows in normal font with no color indicate the allowed cases. Green color and bold font indicates the uncertain cases. Grey color and italic font indicates the disallowed cases. Blue color and bold italic font indicates the cases involving only 2G/3G components.

ID	Components					Condition to support Occurrence	Reasons for Disallowance
	Identity Module	ME	BS	VLR/SG SN/ MME	HN		
1	4G USIM	4G	4G	4G	4G		
2			**3G**	**4G**	**4G**	A1, A4	
3			**2G**	**4G**	**4G**	A1, A4	
4			*4G*	*3G*	*4G*		R5
5			3G	3G	4G		
6			2G	3G	4G		
7			*4G*	*2G*	*4G*		R4
8			*3G*	*2G*	*4G*		R4
9			2G	2G	4G		
10			**4G**	**4G**	**3G**	A2, A4	
11			**3G**	**4G**	**3G**	A1, A2, A4	
12			**2G**	**4G**	**3G**	A1, A2, A4	
13			*4G*	*3G*	*3G*		R5
14			**3G**	**3G**	**3G**	A2	
15			**2G**	**3G**	**3G**	A2	
16			*4G*	*2G*	*3G*		R4
17			*3G*	*2G*	*3G*		R4
18			**2G**	**2G**	**3G**	A2	
19			*4G*	*4G*	*2G*		R6
20			**3G**	**4G**	**2G**	A1, A2, A3, A4	
21			**2G**	**4G**	**2G**	A1, A2, A3, A4	
22			*4G*	*3G*	*2G*		R5
23			**3G**	**3G**	**2G**	A2, A3	
24			**2G**	**3G**	**2G**	A2, A3	
25			*4G*	*2G*	*2G*		R4
26			*3G*	*2G*	*2G*		R4
27			**2G**	**2G**	**2G**	A2, A3	
28	4G USIM	3G	*4G*	*4G*	*4G*		R3
29			**3G**	**4G**	**4G**	A1, A4	
30			**2G**	**4G**	**4G**	A1, A4	
31			*4G*	*3G*	*4G*		R3, R5
32			3G	3G	4G		
33			2G	3G	4G		
34			*4G*	*2G*	*4G*		R3, R4
35			*3G*	*2G*	*4G*		R4
36			2G	2G	4G		
37			*4G*	*4G*	*3G*		R3
38			**3G**	**4G**	**3G**	A1, A2, A4	
39			**2G**	**4G**	**3G**	A1, A2, A4	
40			*4G*	*3G*	*3G*		R3, R5

Fig. 8. Table of cases, part 1 of 7

ID	Components					Condition to support Occurrence	Reasons for Disallowance
	Identity Module	ME	BS	VLR/SG SN/ MME	HN		
41		3G	**3G**	**3G**	**3G**	A2	
42			**2G**	**3G**	**3G**	A2	
43			*4G*	*2G*	*3G*		R3, R4
44			*3G*	*2G*	*3G*		R4
45			**2G**	**2G**	**3G**	A2	
46			*4G*	*4G*	*2G*		R3
47			**3G**	**4G**	**2G**	A1, A2, A3, A4	
48			**2G**	**4G**	**2G**	A1, A2, A3, A4	
49			*4G*	*3G*	*2G*		R3, R5
50			**3G**	**3G**	**2G**	A2, A3	
51			**2G**	**3G**	**2G**	A2, A3	
52			*4G*	*2G*	*2G*		R3, R4
53			*3G*	*2G*	*2G*		R4
54			**2G**	**2G**	**2G**	A2, A3	
55	4G USIM	2G	*4G*	*4G*	*4G*		R2
56			*3G*	*4G*	*4G*		R2
57			**2G**	**4G**	**4G**	A1, A4	
58			*4G*	*3G*	*4G*		R2, R5
59			*3G*	*3G*	*4G*		R2
60			2G	3G	4G		
61			*4G*	*2G*	*4G*		R2, R4
62			*3G*	*2G*	*4G*		R2, R4
63			2G	2G	4G		
64			*4G*	*4G*	*3G*		R2
65			*3G*	*4G*	*3G*		R2
66			**2G**	**4G**	**3G**	A1, A2, A4	
67			*4G*	*3G*	*3G*		R2, R5
68			*3G*	*3G*	*3G*		R2
69			**2G**	**3G**	**3G**	A2	
70			*4G*	*2G*	*3G*		R2, R4
71			*3G*	*2G*	*3G*		R2, R4
72			**2G**	**2G**	**3G**	A2	
73			*4G*	*4G*	*2G*		R2
74			*3G*	*4G*	*2G*		R2
75			**2G**	**4G**	**2G**	A1, A2, A3, A4	
76			*4G*	*3G*	*2G*		R2, R5
77			*3G*	*3G*	*2G*		R2
78			**2G**	**3G**	**2G**	A2, A3	
79			*4G*	*2G*	*2G*		R2, R4
80			*3G*	*2G*	*2G*		R2, R4

Fig. 9. Table of cases, part 2

ID	Components					Condition to support Occurrence	Reasons for Disallowance
	Identity Module	ME	BS	VLR/SGSN/MME	HN		
81		2G	2G	2G	2G	A2, A3	
82	USIM	4G	4G	4G	4G		
83			3G	4G	4G	A1, A4	
84			2G	4G	4G	A1, A4	
85			4G	3G	4G		R5
86			3G	3G	4G		
87			2G	3G	4G		
88			4G	2G	4G		R4
89			3G	2G	4G		R4
90			2G	2G	4G		
91			4G	4G	3G	A1, A4	
92			3G	4G	3G	A1, A4	
93			2G	4G	3G	A1, A4	
94			4G	3G	3G		R5
95			3G	3G	3G	A2	
96			2G	3G	3G	A2	
97			4G	2G	3G		R4
98			3G	2G	3G		R4
99			2G	2G	3G	A2	
100			4G	4G	2G		R6
101			3G	4G	2G	A1, A4	
102			2G	4G	2G	A1, A4	
103			4G	3G	2G		R5
104			3G	3G	2G	A2	
105			2G	3G	2G	A2	
106			4G	2G	2G		R4
107			3G	2G	2G		R4
108			2G	2G	2G	A2	
109	USIM	3G	4G	4G	4G		R3
110			3G	4G	4G	A1, A4	
111			2G	4G	4G	A1, A4	
112			4G	3G	4G		R3, R5
113			3G	3G	4G		
114			2G	3G	4G		
115			4G	2G	4G		R3, R4
116			3G	2G	4G		R4
117			2G	2G	4G		
118			4G	4G	3G		R3
119			3G	4G	3G	A1, A4	
120			2G	4G	3G	A1, A4	

Fig. 10. Table of cases, part 3

ID	Components					Condition to support Occurrence	Reasons for Disallowance
	Identity Module	ME	BS	VLR/SGSN/MME	HN		
121	USIM	3G	4G	3G	3G		R3, R5
122			3G	3G	3G		
123			2G	3G	3G		
124			4G	2G	3G		R3, R4
125			3G	2G	3G		R4
126			2G	2G	3G		
127			4G	4G	2G		R3
128			3G	4G	2G	A1, A4	
129			2G	4G	2G	A1, A4	
130			4G	3G	2G		R3, R5
131			3G	3G	2G		R7
132			2G	3G	2G		
133			4G	2G	2G		R3, R4
134			3G	2G	2G		R4
135			2G	2G	2G		
136	USIM	2G	4G	4G	4G		R2
137			3G	4G	4G		R2
138			2G	4G	4G	A1, A4	
139			4G	3G	4G		R2, R5
140			3G	3G	4G		R2
141			2G	3G	4G		
142			4G	2G	4G		R2, R4
143			3G	2G	4G		R2, R4
144			2G	2G	4G		
145			4G	4G	3G		R2
146			3G	4G	3G		R2
147			2G	4G	3G	A1, A4	
148			4G	3G	3G		R2, R5
149			3G	3G	3G		R2
150			2G	3G	3G		
151			4G	2G	3G		R2, R4
152			3G	2G	3G		R2, R4
153			2G	2G	3G		
154			4G	4G	2G		R2
155			3G	4G	2G		R2
156			2G	4G	2G	A1, A4	
157			4G	3G	2G		R2, R5
158			3G	3G	2G		R2
159			2G	3G	2G		
160			4G	2G	2G		R2, R4

Fig. 11. Table of cases, part 4

ID	Components					Condition to support Occurrence	Reasons for Disallowance
	Identity Module	ME	BS	VLR/SGSN/MME	HN		
161	USIM	2G	3G	2G	2G		R2, R4
162			2G	2G	2G		
163	SIM	4G	4G	4G	4G		R1
164			3G	4G	4G		R1
165			2G	4G	4G		R1
166			4G	3G	4G		R1, R5
167			3G	3G	4G		
168			2G	3G	4G		
169			4G	2G	4G		R1, R4
170			3G	2G	4G		R1, R4
171			2G	2G	4G		
172			4G	4G	3G		R1
173			3G	4G	3G		R1
174			2G	4G	3G		R1
175			4G	3G	3G		R1, R5
176			3G	3G	3G		
177			2G	3G	3G		
178			4G	2G	3G		R1, R4
179			3G	2G	3G		R4
180			2G	2G	3G		
181			4G	4G	2G		R1
182			3G	4G	2G		R1
183			2G	4G	2G		R1
184			4G	3G	2G		R1, R5
185			3G	3G	2G		
186			2G	3G	2G		
187			4G	2G	2G		R1, R4
188			3G	2G	2G		R4
189			2G	2G	2G		
190	SIM	3G	4G	4G	4G		R1, R3
191			3G	4G	4G		R1
192			2G	4G	4G		R1
193			4G	3G	4G		R1, R3, R5
194			3G	3G	4G		
195			2G	3G	4G		
196			4G	2G	4G		R1, R3, R4
197			3G	2G	4G		R1, R4
198			2G	2G	4G		
199			4G	4G	3G		R1, R3
200			3G	4G	3G		R1

Fig. 12. Table of cases, part 5

ID	Components					Condition to support Occurrence	Reasons for Disallowance
	Identity Module	ME	BS	VLR/SGSN/MME	HN		
201	SIM	3G	2G	4G	3G		R1
202			4G	3G	3G		R1, R3, R5
203			3G	3G	3G		
204			2G	3G	3G		
205			4G	2G	3G		R1, R3, R4
206			3G	2G	3G		R4
207			2G	2G	3G		
208			4G	4G	2G		R1, R3
209			3G	4G	2G		R1
210			2G	4G	2G		R1
211			4G	3G	2G		R1, R3, R5
212			3G	3G	2G		
213			2G	3G	2G		
214			4G	2G	2G		R1, R3, R4
215			3G	2G	2G		R4
216			2G	2G	2G		
217	SIM	2G	4G	4G	4G		R1, R2
218			3G	4G	4G		R1, R2
219			2G	4G	4G		R1
220			4G	3G	4G		R1, R2, R5
221			3G	3G	4G		R1, R2
222			2G	3G	4G		
223			4G	2G	4G		R1, R2, R4
224			3G	2G	4G		R1, R2, R4
225			2G	2G	4G		
226			4G	4G	3G		R1, R2
227			3G	4G	3G		R1, R2
228			2G	4G	3G		R1
229			4G	3G	3G		R1, R2, R5
230			3G	3G	3G		R2
231			2G	3G	3G		
232			4G	2G	3G		R1, R2, R4
233			3G	2G	3G		R2, R4
234			2G	2G	3G		
235			4G	4G	2G		R1, R2
236			3G	4G	2G		R1, R2
237			2G	4G	2G		R1
238			4G	3G	2G		R1, R5
239			3G	3G	2G		R1, R2
240			2G	3G	2G		

Fig. 13. Table of cases, part 6

ID	Components					Condition to support Occurrence	Reasons for Disallowance
	Identity Module	ME	BS	VLR/SGSN/MME	HN		
241	SIM	2G	4G	2G	2G		R1, R2, R4
242			3G	2G	2G		R2, R4
243			2G	2G	2G		

Fig. 14. Table of cases, part 7

APPENDIX B
TABLE OF SCENARIOS

Figure 15 – 17 show the determination of the AKA scenarios and respective reasoning, with reference to the list A–X in Table II.

ID	Components					Scenarios	Reason	
	Identity Module	ME	BS	VLR/SGSN/MME	HN		Stated in Spec	Interpretation
1	4G USIM	4G	4G	4G	4G	S3	A, G, K, W	
2			3G	4G	4G	S10	B, H, V	T
3			2G	4G	4G	S9	B, I, N, V	T
5			3G	3G	4G	S2	H, V	O, T
6			2G	3G	4G	S6	I, N, V	O, T
9			2G	2G	4G	S4	I, V	P, T
10			4G	4G	3G	S7/S8	G, K, W, X	Q, U
11			3G	4G	3G	S2	H, V	Q, T
12			2G	4G	3G	S6	I, N, V	Q, T
14			3G	3G	3G	S2	C, H, V	T
15			2G	3G	3G	S6	C, I, N, V	T
18			2G	2G	3G	S4	I, V	R, T
20			3G	4G	2G	S5	E, H, M, V	T
21			2G	4G	2G	S1	E, I, V	T
23			3G	3G	2G	S5	E, H, M, V	T
24			2G	3G	2G	S1	E, I, V	T
27			2G	2G	2G	S1	E, I, V	T
29		3G	3G	4G	4G	S10	B, H, K, V	
30			2G	4G	4G	S9	B, I, L, N, V	
32			3G	3G	4G	S2	H, K, V	O
33			2G	3G	4G	S6	I, L, N, V	O
36			2G	2G	4G	S4	I, L, V	P
38			3G	4G	3G	S2	H, K, V	Q
39			2G	4G	3G	S6	I, L, N, V	Q
41			3G	3G	3G	S2	C, H, K, V	
42			2G	3G	3G	S6	C, I, L, N, V	
45			2G	2G	3G	S4	I, L, V	R
47			3G	4G	2G	S5	E, H, K, M, V	
48			2G	4G	2G	S1	E, I, L, V	
50			3G	3G	2G	S5	E, H, K, M, V	
51			2G	3G	2G	S1	E, I, L, V	
54			2G	2G	2G	S1	E, I, L, V	
57		2G	2G	4G	4G	S9	B, I, K, N, V	
60			2G	3G	4G	S6	I, K, N, V	O
63			2G	2G	4G	S4	I, K, V	P
66			2G	4G	3G	S6	I, K, N, V	Q
69			2G	3G	3G	S6	C, I, K, N, V	
72			2G	2G	3G	S4	I, K, V	R
75			2G	4G	2G	S1	E, I, K, V	
78			2G	3G	2G	S1	E, I, K, V	

Fig. 15. Table of scenarios, part 1 of 3

ID	Components					Scenarios	Reason	
	Identity Module	ME	BS	VLR/SGSN/MME	HN		Stated in Spec	Interpretation
81		2G	2G	2G	2G	S1	E, I, K, V	
82	USIM	4G	4G	4G	4G	S3	A, G, J, K, W	
83			3G	4G	4G	S10	B, H, V	T
84			2G	4G	4G	S9	B, I, N, V	T
86			3G	3G	4G	S2	H, V	O, T
87			2G	3G	4G	S6	I, N, V	O, T
90			2G	2G	4G	S4	I, V	P, T
91			4G	4G	3G	S7/S8	G, K, W, X	Q, U
92			3G	4G	3G	S2	H, V	Q, T
93			2G	4G	3G	S6	I, N, V	Q, T
95			3G	3G	3G	S2	C, H, V	T
96			2G	3G	3G	S6	C, I, N, V	T
99			2G	2G	3G	S4	D, I, V	T
101			3G	4G	2G	S5	E, H, M, V	T
102			2G	4G	2G	S1	E, I, V	T
104			3G	3G	2G	S5	E, H, M, V	T
105			2G	3G	2G	S1	E, I, V	T
108			2G	2G	2G	S1	E, I, V	T
110	USIM	3G	3G	4G	4G	S2	B, H, K, V	
111			2G	4G	4G	S6	B, I, L, N, V	
113			3G	3G	4G	S2	H, K, V	O
114			2G	3G	4G	S6	I, L, N, V	O
117			2G	2G	4G	S4	I, L, V	P
119			3G	4G	3G	S2	H, K, V	Q
120			2G	4G	3G	S6	I, L, N, V	Q
122			3G	3G	3G	S2	C, H, K, V	
123			2G	3G	3G	S6	C, I, L, N, V	
126			2G	2G	3G	S4	D, I, L, V	
128			3G	4G	2G	S5	E, H, K, M, V	
129			2G	4G	2G	S1	E, I, L, V	
132			2G	3G	2G	S1	E, I, L, V	
135			2G	2G	2G	S1	E, I, L, V	
138	USIM	2G	2G	4G	4G	S6	B, I, K, N, V	
141			2G	3G	4G	S6	I, K, N, V	O
144			2G	2G	4G	S4	I, K, V	P
147			2G	4G	3G	S6	I, K, N, V	Q
150			2G	3G	3G	S6	C, I, K, N, V	
153			2G	2G	3G	S4	D, I, K, V	
156			2G	4G	2G	S1	E, I, K, V	
159			2G	3G	2G	S1	E, I, K, V	

Fig. 16. Table of scenarios, part 2

ID	Components					Scenarios	Reason	
	Identity Module	ME	BS	VLR/SGSN/MME	HN		Stated in Spec	Interpretation
162	USIM	2G	2G	2G	2G	S1	E, I, K, V	
167	SIM	4G	3G	3G	4G	S5	H, M, V	S, T
168			2G	3G	4G	S1	I, V	S, T
171			2G	2G	4G	S1	I, V	S, T
176			3G	3G	3G	S5	F, H, M, V	T
177			2G	3G	3G	S1	F, I, V	T
180			2G	2G	3G	S1	F, I, V	T
185			3G	3G	2G	S5	E, H, M, V	T
186			2G	3G	2G	S1	E, I, V	T
189			2G	2G	2G	S1	E, I, V	T
194	SIM	3G	3G	3G	4G	S5	H, K, M, V	S
195			2G	3G	4G	S1	I, L, V	S
198			2G	2G	4G	S1	I, L, V	S
203			3G	3G	3G	S5	F, H, K, M, V	
204			2G	3G	3G	S1	F, I, L, V	
207			2G	2G	3G	S1	F, I, L, V	
212			3G	3G	2G	S5	E, H, K, M, V	
213			2G	3G	2G	S1	E, I, L, V	
216			2G	2G	2G	S1	E, I, L, V	
222	SIM	2G	2G	3G	4G	S1	I, K, V	S
225			2G	2G	4G	S1	I, K, V	S
231			2G	3G	3G	S1	F, I, K, V	
234			2G	2G	3G	S1	F, I, K, V	
240			2G	3G	2G	S1	E, I, K, V	
243			2G	2G	2G	S1	E, I, K, V	

Fig. 17. Table of scenarios, part 3

APPENDIX C
MODELS OF THE SCENARIOS

This section presents scenarios S7 (without LTE IV), S8, S9+ (with LTE IV) and S10+ (with LTE IV) with some explanation. The pure 4G model is discussed in Sect. VI. The other pure models and scenarios are presented in [16]. Appendix D gives the complete code files for all models.

S7. LTE I ∥ UMTS II–III ∥ LTE V, conv(CK IK → K_{ASME}, MME)

Most of the code in this model is inherited from the 4G model and the UMTS model. The key derivation function used by the MME and the MS to generate the local master key K_{ASME} is declared as:

```
fun kdf_asme(cipherKey, integKey): asmeKey.
```

There are three main processes in our model representing the behavior of the MS, the SN and the HN respectively.

```
  (*MS non-deterministically choose
      the capability of encryption*)
  let cap_ms: bool = encCapability() in
  (*Send out cap_ms to SN *)
  out(pubChannel, (CAP, cap_ms));
  (*Send out permanent ID *)
  out(pubChannel, (ID, imsi_ms));
  (*Input challenge message from SN *)
  in(pubChannel, (=CHALLENGE, rand_ms: nonce, mac_ms: mac));
  if f1(ki, rand_ms) = mac_ms then
    (*Compute response and encryption key*)
    let res_ms: resp = f2(ki, rand_ms) in
    let ck_ms: cipherKey = f3(ki, rand_ms) in
    let ik_ms: integKey = f4(ki, rand_ms) in
    (*MS is authenticating itself to SN*)
    event begSN(imsi_ms, ck_ms, ik_ms);
    (*Send out response to SN *)
    out(pubChannel, (RES, res_ms));
    let kasme_ms = kdf_asme(ck_ms, ik_ms) in
    let kenb_ms: enbKey = kdf_enb(kasme_ms) in
        let kasenc_ms: asEncKey = kdf_as_enc(kenb_ms) in
        let kasint_ms: asIntKey = kdf_as_int(kenb_ms) in
        let kupenc_ms: upEncKey = kdf_up_enc(kenb_ms) in
    (*Receive GSM cipher mode command *)
    in(pubChannel, (=ASSMC, enableEnc_as_ms: bool,
        =finteg_as(bool2bitstring(enableEnc_as_ms),
            kasint_ms)));
        out(pubChannel, (ASSMComplete, as_smcomplete_msg,
            finteg_as(as_smcomplete_msg, kasint_ms)));
    event endMS(imsi_ms, kenb_ms, cap_ms);
    in(pubChannel, (=MSG, datamsg: bitstring,
        =finteg_as(datamsg, kasint_ms)));
        out(pubChannel, sencrypt(secret, ck_ms));
        out(pubChannel, sencryptInteg(secret, ik_ms));
    if enableEnc_as_ms = true then
        let msgcontent: bitstring = sdecrypt_as(datamsg,
            kasenc_ms) in 0.

let processSN =
  (*Receive MS's capability *)
  in(pubChannel, (=CAP, cap_sn: bool));
  (*Receive permanent ID *)
  in(pubChannel, (=ID, imsi_sn: ident));
  (*Send out authentication vector request *)
  out(secureChannel, (AV_REQ, imsi_sn));
  (*Receive authentication vector *)
  in(secureChannel, (=AV, =imsi_sn, rand_sn: nonce,
      xres_sn: resp, ck_sn: cipherKey, ik_sn: integKey,
      mac_sn: mac));
  (*Send authentication challenge to MS *)
  out(pubChannel, (CHALLENGE, rand_sn, mac_sn));
  (*Receive response *)
  in(pubChannel, (=RES, res_sn: resp));
  (*Check whether received response equal to XRES*)
  if res_sn = xres_sn then
    (*At this point, SN authenticated MS*)
    event endSN(imsi_sn, ck_sn, ik_sn);
    let kasme_sn = kdf_asme(ck_sn, ik_sn) in
    let kenb_sn: enbKey = kdf_enb(kasme_sn) in
    let kasenc_sn: asEncKey = kdf_as_enc(kenb_sn) in
        let kasint_sn: asIntKey = kdf_as_int(kenb_sn) in
        let kupenc_sn: upEncKey = kdf_up_enc(kenb_sn) in
        event begMS(imsi_sn, kenb_sn, cap_sn);
        out(pubChannel, (ASSMC, cap_sn,
            finteg_as(bool2bitstring(cap_sn), kasint_sn)));
        in(pubChannel, (=ASSMComplete, =as_smcomplete_msg,
            =finteg_as(as_smcomplete_msg, kasint_sn)));
        if cap_sn = false then
            event disableEnc;
            out(pubChannel, (MSG, payload,
                finteg_as(payload, kasint_sn)))
        else
            out(pubChannel, (MSG, sencrypt_as(payload,
                 kasenc_sn), finteg_as(sencrypt_as(payload,
                 kasenc_sn), kasint_sn))).

let processHN =
  (*Receive authentication vector request *)
  in(secureChannel, (=AV_REQ, imsi_hn: ident));
  (*Generate a fresh random number*)
  new rand_hn: nonce;
  (*Computes expected response and Kc*)
  get keys(=imsi_hn, ki_hn) in
  let mac_hn: mac = f1(ki_hn, rand_hn) in
  let xres_hn: resp = f2(ki_hn, rand_hn) in
  let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
  let ik_hn: integKey = f4(ki_hn, rand_hn) in
  (*Send out authentication vector *)
  out(secureChannel, (AV, imsi_hn, rand_hn,
      xres_hn, ck_hn, ik_hn, mac_hn)).
```

The HN process is the same as the one in the UMTS model. In line 19 and line 58, the MS and the SN derive the local master key K_{ASME} from the cipher key and the integrity key.

Security Property Specifications and Findings The events used in the correspondence assertions to specify the authentication properties are declared as:

```
event begSN(ident, cipherKey, integKey).
event endSN(ident, cipherKey, integKey).
event begMS(ident, enbKey, bool).
event endMS(ident, enbKey, bool).
```

The secrecy and authentication properties are specified as:

```
query attacker(payload).
query attacker(payload) ~> event(disableEnc).
query attacker(secret).
query x1: ident, x2: cipherKey, x3: integKey;
  event(endSN(x1, x2, x3)) ~> event(begSN(x1, x2, x3)).
query x1: ident, x2: enbKey, x3: bool;
  event(endMS(x1, x2, x3)) ~> event(begMS(x1, x2, x3)).
```

The secrecy property of the message payload does not hold, because the attacker can always learn the payload if the MS is not capable of encryption. The conditional secrecy (line 2) holds. That means if the encryption is enabled, the attacker can never learn the message payload. The property specified in line 3 is used to test the secrecy of the keys. ProVerif proves the key secrecy. The authentication of the MS to the SN is specified in lines 4–5. ProVerif proves this authentication property. The authentication of the SN to the MS is specified in lines 6–7. Proverif finds a attack trace that violates the property:

```
new imsi_ms creating imsi_ms_5870 at {2} in copy a
new ki creating ki_5871 at {3} in copy a
insert keys(imsi_ms_5870,ki_5871) at {4} in copy a
out(pubChannel, (CAP,true)) at {6} in copy a
out(pubChannel, (ID,imsi_ms_5870)) at {7} in copy a
in(pubChannel, (CAP,a_5860)) at {31} in copy a_5861
in(pubChannel, (ID,imsi_ms_5870)) at {32} in copy a_5861
in(pubChannel, (CAP,as_smcomplete_msg)) at {31} in copy a_5862
in(pubChannel, (ID,imsi_ms_5870)) at {32} in copy a_5862
```

```
10  in(pubChannel, (CAP,a_5863)) at {31} in copy a_5864
11  in(pubChannel, (ID,imsi_ms_5870)) at {32} in copy a_5864
12  in(pubChannel, (CAP,as_smcomplete_msg)) at {31} in copy a_5865
13  in(pubChannel, (ID,imsi_ms_5870)) at {32} in copy a_5865
14  in(pubChannel, (CAP,a_5866)) at {31} in copy a_5867
15  in(pubChannel, (ID,imsi_ms_5870)) at {32} in copy a_5867
16  in(pubChannel, (CAP,a_5868)) at {31} in copy a_5869
17  in(pubChannel, (ID,imsi_ms_5870)) at {32} in copy a_5869
18  out(secureChannel, (AV_REQ,imsi_ms_5870)) at {33} in
19    copy a_5865 received at {55} in copy a_5859
20  new rand_hn creating rand_hn_5872 at {56} in copy a_5859
21  get keys(imsi_ms_5870,ki_5871) at {57} in copy a_5859
22  out(secureChannel, (AV,imsi_ms_5870,rand_hn_5872,
23    f2(ki_5871,rand_hn_5872), f3(ki_5871,rand_hn_5872),
24    f4(ki_5871,rand_hn_5872), f1(ki_5871,rand_hn_5872)))
25    at {62} in copy a_5859 received at {34} in copy a_5865
26  out(pubChannel, (CHALLENGE,rand_hn_5872,f1(ki_5871,
27    rand_hn_5872))) at {35} in copy a_5865
28  in(pubChannel, (CHALLENGE,rand_hn_5872,f1(ki_5871,
29    rand_hn_5872))) at {8} in copy a
30  event(begSN(imsi_ms_5870,f3(ki_5871,rand_hn_5872),
31    f4(ki_5871,rand_hn_5872))) at {13} in copy a
32  out(pubChannel, (RES,f2(ki_5871,rand_hn_5872)))
33    at {14} in copy a
34  in(pubChannel, (RES,f2(ki_5871,rand_hn_5872)))
35    at {36} in copy a_5865
36  event(endSN(imsi_ms_5870,f3(ki_5871,rand_hn_5872),
37    f4(ki_5871,rand_hn_5872))) at {38} in copy a_5865
38  event(begMS(imsi_ms_5870,kdf_enb(kdf_asme(f3(ki_5871,
39    rand_hn_5872), f4(ki_5871,rand_hn_5872))),
40    as_smcomplete_msg)) at {44} in copy a_5865
41  out(pubChannel, (ASSMC,as_smcomplete_msg,
42    finteg_as(as_smcomplete_msg, kdf_as_int(kdf_enb(kdf_asme(
43    f3(ki_5871,rand_hn_5872),f4(ki_5871, rand_hn_5872)))))))
44    at {46} in copy a_5865
45  in(pubChannel, (ASSMComplete,as_smcomplete_msg,
46    finteg_as(as_smcomplete_msg, kdf_as_int(kdf_enb(kdf_asme(
47    f3(ki_5871,rand_hn_5872),f4(ki_5871, rand_hn_5872)))))))
48    at {47} in copy a_5865
49  out(pubChannel, (MSG,sencrypt_as(payload,
50    kdf_as_enc(kdf_enb(kdf_asme(f3(ki_5871,rand_hn_5872),
51    f4(ki_5871,rand_hn_5872))))),finteg_as(sencrypt_as(payload,
52    kdf_as_enc(kdf_enb(kdf_asme(f3(ki_5871,rand_hn_5872),
53    f4(ki_5871,rand_hn_5872))))),
54    kdf_as_int(kdf_enb(kdf_asme(f3(ki_5871,rand_hn_5872),
55    f4(ki_5871,rand_hn_5872)))))))
56    at {53} in copy a_5865
57  in(pubChannel, (ASSMC,sencrypt_as(payload,
58    kdf_as_enc(kdf_enb(kdf_asme(f3(ki_5871,rand_hn_5872),
59    f4(ki_5871,rand_hn_5872))))),finteg_as(sencrypt_as(payload,
60    kdf_as_enc(kdf_enb(kdf_asme(f3(ki_5871,rand_hn_5872),
61    f4(ki_5871,rand_hn_5872))))),
62    kdf_as_int(kdf_enb(kdf_asme(f3(ki_5871,rand_hn_5872),
63    f4(ki_5871,rand_hn_5872)))))))
64    at {20} in copy a
65  out(pubChannel, (ASSMComplete,as_smcomplete_msg,
66    finteg_as(as_smcomplete_msg, kdf_as_int(
67    kdf_enb(kdf_asme(f3(ki_5871,rand_hn_5872),
68    f4(ki_5871,rand_hn_5872))))))) at {22} in copy a
69  event(endMS(imsi_ms_5870,kdf_enb(kdf_asme(f3(ki_5871,
70    rand_hn_5872),f4(ki_5871,rand_hn_5872))),true))
71    at {23} in copy a
72  The event endMS(imsi_ms_5870,kdf_enb(kdf_asme(f3(ki_5871,
73    rand_hn_5872), f4(ki_5871,rand_hn_5872))), true)
74    is executed.
```

In this trace, the attacker intercepts the capability message sent by the MS and replaces the capabilities with different ones. Since the event beginMS in process SN records the capabilities received by the SN, which are the replaced ones. The event endMS is executed after the MS receives the security mode command message. Because the security mode command message does not contain the received MS's capabilities, the MS has no way to confirm whether the SN receives the correct capabilities. The event endMS is executed with recording the original capabilities of the MS. The two events do not agree the third parameter (the capabilities), which violates the correspondence assertion.

S8. LTE I' ∥ UMTS II–III ∥ LTE IV–V, conv(CK IK nonces → K_{ASME}, MME)

Figure 18 shows details of the ProVerif model and the locations of the events which are used to specify the conditional payload secrecy and the authentication properties. Most of the code in this model is inherited from the 4G model and the UMTS model. The key derivation function used to derive the K_{ASME} is defined as:

```
fun kdf_asme(cipherKey, integKey, nonce, nonce): asmeKey.
```

There are four main processes in our model representing the behavior of the MS, the BS, the SN and the HN respectively.

```
1   (*AS SMC procedure in process MS*)
2   let pMSAS(kasme_ms: asmeKey, imsi_ms: ident, cap_ms: bool) =
3       let kenb_ms: enbKey = kdf_enb(kasme_ms) in
4       let kasenc_ms: asEncKey = kdf_as_enc(kenb_ms) in
5       let kasint_ms: asIntKey = kdf_as_int(kenb_ms) in
6       let kupenc_ms: upEncKey = kdf_up_enc(kenb_ms) in
7       in(pubChannel, (=ASSMC, enableEnc_as_ms: bool,
8           =finteg_as(bool2bitstring(enableEnc_as_ms),
9               kasint_ms)));
10      out(pubChannel, (ASSMComplete, as_smcomplete_msg,
11          finteg_as(as_smcomplete_msg, kasint_ms)));
12      event endMS_ENB(imsi_ms, kenb_ms, cap_ms);
13      in(pubChannel, (=MSG, datamsg: bitstring,
14          =finteg_as(datamsg, kasint_ms)));
15      out(pubChannel, sencrypt_as(secret, kasenc_ms));
16      out(pubChannel, senc_int_as(secret, kasint_ms));
17      out(pubChannel, senc_up(secret, kupenc_ms));
18      if enableEnc_as_ms = true then
19          let msgcontent: bitstring =
20              sdecrypt_as(datamsg, kasenc_ms) in 0.
21
22  (*process respresenting MS*)
23  let processMS =
24      (*The identity of the MS*)
25      new imsi_ms: ident;
26      (*Pre-shared key*)
27      new ki: key;
28      (*Insert id/pre-shared key pair into the private table*)
29      insert keys(imsi_ms, ki);
30      (*MS non-deterministically choose the capability of encryption*)
31      let cap_ms: bool = encCapability() in
32      out(pubChannel, (CAP, cap_ms));
33      out(pubChannel, (ID, imsi_ms));
34      new nonce_ms: nonce;
35      out(pubChannel, (NONCE_TAU, nonce_ms));
36      in(pubChannel, (=CHALLENGE, rand_ms: nonce, =f1(ki, rand_ms)));
37      let res_ms: resp = f2(ki, rand_ms) in
38      let ck_ms: cipherKey = f3(ki, rand_ms) in
39      let ik_ms: integKey = f4(ki, rand_ms) in
40      event begSN(imsi_ms, ck_ms, ik_ms);
41      out(pubChannel, (RES, res_ms));
42      (*NAS SMC procedure*)
43      in(pubChannel, (=NASSMC, enableEnc_nas_ms: bool, =cap_ms,
44          =nonce_ms, nonce_mme_ms: nonce, nas_mac: msgMac));
45      let kasme_ms: asmeKey = kdf_asme(ck_ms, ik_ms,
46          nonce_ms, nonce_mme_ms) in
47      let knasenc_ms: nasEncKey = kdf_nas_enc(kasme_ms) in
48      let knasint_ms: nasIntKey = kdf_nas_int(kasme_ms) in
49      if(nas_mac = finteg_nas((enableEnc_nas_ms, cap_ms,
50          nonce_ms, nonce_mme_ms), knasint_ms)) then
51          event endMS(imsi_ms, ck_ms, ik_ms, cap_ms);
52          (*NAS key secrecy*)
53          out(pubChannel, sencrypt_nas(secret, knasenc_ms));
54          out(pubChannel, senc_int_nas(secret, knasint_ms));
55          if enableEnc_nas_ms = false then
56              out(pubChannel, (NASSMComplete, nas_smcomplete_msg,
57                  finteg_nas(nas_smcomplete_msg, knasint_ms)));
58              pMSAS(kasme_ms, imsi_ms, cap_ms)
59          else
60              out(pubChannel, (NASSMComplete,
61                  sencrypt_nas(nas_smcomplete_msg, knasenc_ms),
62                  finteg_nas(sencrypt_nas(nas_smcomplete_msg,
63                  knasenc_ms), knasint_ms)));
64              pMSAS(kasme_ms, imsi_ms, cap_ms).
65
66  (*process representing e-nodeB*)
```

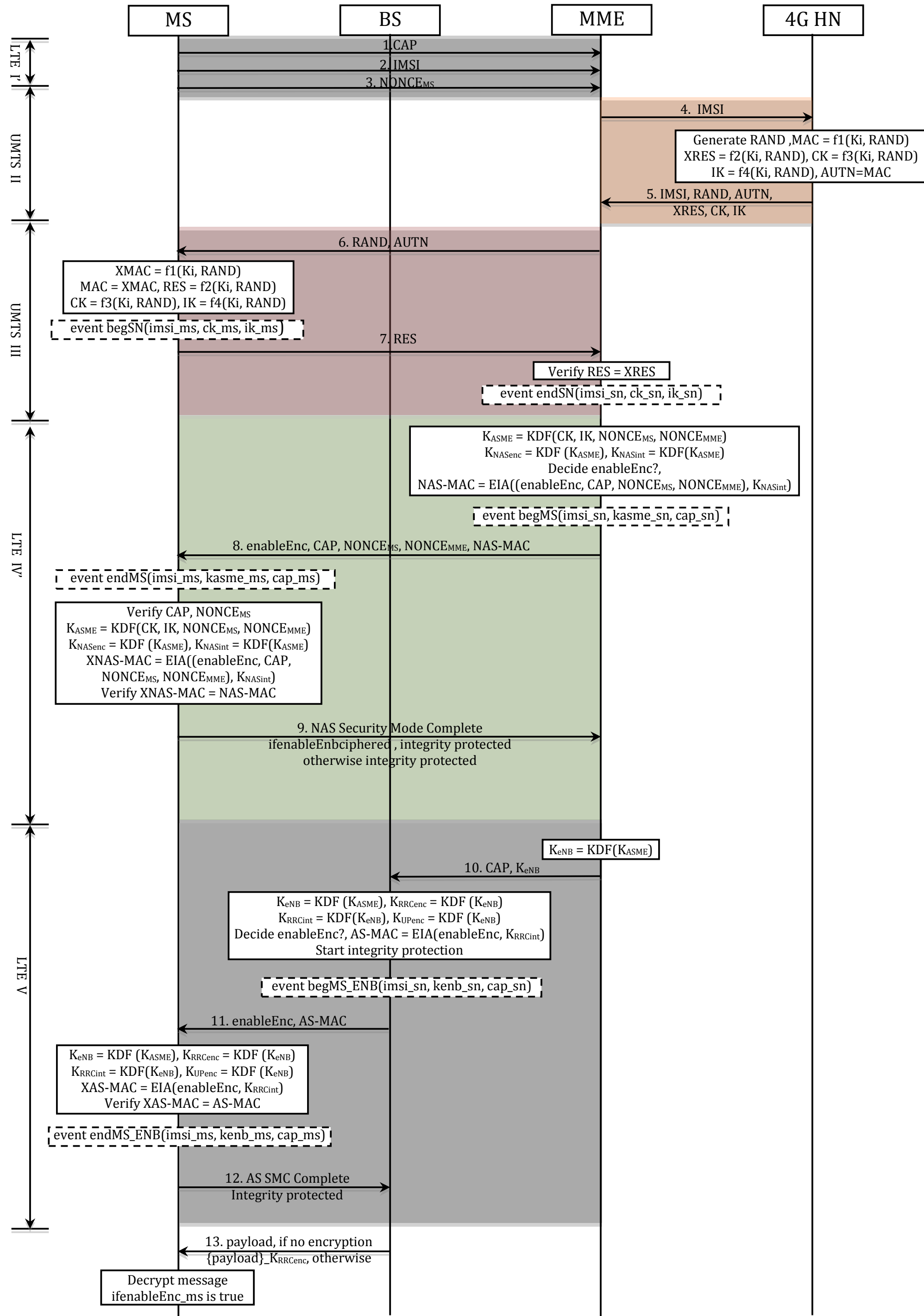

Fig. 18. Scenario S8 annotated in accord with our model

```
67  let processENB =
68      in(sChannelSnBts, (kasme_enb: asmeKey, imsi_enb: ident,
69          cap_enb: bool));
70      let kenb_enb: enbKey = kdf_enb(kasme_enb) in
71      let kasenc_enb: asEncKey = kdf_as_enc(kenb_enb) in
72      let kasint_enb: asIntKey = kdf_as_int(kenb_enb) in
73      let kupenc_enb: upEncKey = kdf_up_enc(kenb_enb) in
74      event begMS_ENB(imsi_enb, kenb_enb, cap_enb);
75      out(pubChannel, (ASSMC, cap_enb,
76          finteg_as(bool2bitstring(cap_enb), kasint_enb)));
77      in(pubChannel, (=ASSMComplete, =as_smcomplete_msg,
78          =finteg_as(as_smcomplete_msg, kasint_enb)));
79      if cap_enb = false then
80          event disableEnc;
81          out(pubChannel, (MSG, payload,
82              finteg_as(payload, kasint_enb)))
83      else
84          out(pubChannel, (MSG, sencrypt_as(payload, kasenc_enb),
85              finteg_as(sencrypt_as(payload, kasenc_enb),
86              kasint_enb))).
87
88  (*process representing MME*)
89  let processMME =
90      in(pubChannel, (=CAP, cap_sn: bool));
91      in(pubChannel, (=ID, imsi_sn: ident));
92      in(pubChannel, (=NONCE_TAU, nonce_ms_sn:nonce));
93      out(secureChannel, (AV_REQ, imsi_sn));
94      in(secureChannel, (=AV, =imsi_sn, rand_sn: nonce,
```

```
95            xres_sn: resp, ck_sn: cipherKey,
96            ik_sn: integKey, mac_sn: mac));
97        out(pubChannel, (CHALLENGE, rand_sn, mac_sn));
98        in(pubChannel, (=RES, =xres_sn));
99        event endSN(imsi_sn, ck_sn, ik_sn);
100       new nonce_mme: nonce;
101       (*NAS SMC procedure*)
102       let kasme_sn: asmeKey = kdf_asme(ck_sn, ik_sn,
103           nonce_ms_sn, nonce_mme) in
104       let knasenc_sn: nasEncKey = kdf_nas_enc(kasme_sn) in
105       let knasint_sn: nasIntKey = kdf_nas_int(kasme_sn) in
106       event begMS(imsi_sn, ck_sn, ik_sn, cap_sn);
107       out(pubChannel, (NASSMC, cap_sn, cap_sn, nonce_ms_sn,
108           nonce_mme, finteg_nas((cap_sn, cap_sn,
109           nonce_ms_sn, nonce_mme), knasint_sn)));
110       in(pubChannel, (=NASSMComplete, msg_nas:bitstring,
111           =finteg_nas(msg_nas, knasint_sn)));
112       if cap_sn = true then
113           if sdecrypt_nas(msg_nas, knasenc_sn)
114               = nas_smcomplete_msg then
115                   out(sChannelSnBts, (kasme_sn,
116                       imsi_sn, cap_sn))
117           else 0
118       else
119           if cap_sn = false then
120               if msg_nas = nas_smcomplete_msg then
121                   out(sChannelSnBts, (kasme_sn,
122                       imsi_sn, cap_sn))
123               else 0
124           else 0.
125
126   (*process representing HN*)
127   let processHN =
128     (*Receive authentication vector request *)
129     in(secureChannel, (=AV_REQ, imsi_hn: ident));
130     (*Generate a fresh random number*)
131     new rand_hn: nonce;
132     (*Computes expected response and Kc*)
133     get keys(=imsi_hn, ki_hn) in
134     let mac_hn: mac = f1(ki_hn, rand_hn) in
135     let xres_hn: resp = f2(ki_hn, rand_hn) in
136     let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
137     let ik_hn: integKey = f4(ki_hn, rand_hn) in
138     (*Send out authentication vector *)
139     out(secureChannel, (AV, imsi_hn, rand_hn,
140         xres_hn, ck_hn, ik_hn, mac_hn)).
```

This scenario is triggered by the TAU request, in addition to the IMSI and capabilities, the MS generates a nonce and sends it to the MME (lines 34–35). The authentication vector request and response procedure is modeled in lines 93–96 and lines 129–140. The MME generates a nonce (line 100) and derives the K_{ASME} using the nonces and cipher and integrity keys (lines 102–103). The MME then sends the integrity protected NAS SMC messages which includes the received capabilities and both nonces in lines 107–109. Upon receiving the NAS SMC messages, the MS derives the K_{ASME} using the nonces and cipher and integrity keys as in MME (lines 45–46). The MS then verifies the MAC of the messages and sends out the NAS Complete messages to the MME. The AS SMC procedure is the same as in 4G model.

Security Property Specifications and Findings The events used to specify the authentication properties are specified as:

```
1 event begSN(ident, cipherKey, integKey).
2 event endSN(ident, cipherKey, integKey).
3 event begMS(ident, cipherKey, integKey, bool).
4 event endMS(ident, cipherKey, integKey, bool).
5 event begMS_ENB(ident, enbKey, bool).
6 event endMS_ENB(ident, enbKey, bool).
```

We specify the security properties as following:

- Key Secrecy

```
1 not attacker(new ki).
2 query attacker(secret).
```

- Conditional Payload Secrecy

```
query attacker(payload)⇝event(disableEnc).
```

- Mutual Authentication between the MS and the MME

```
1 query x1: ident, x2: cipherKey, x3: integKey;
2     event(endSN(x1, x2, x3)) ⇝
3         event(begSN(x1, x2, x3)).
4 query x1: ident, x2: cipherKey, x3: integKey, x4: bool;
5     event(endMS(x1, x2, x3, x4)) ⇝
6         event(begMS(x1, x2, x3, x4)).
```

- Authentication of the BS to the MS

```
1 query x1: ident, x2: enbKey, x3: bool;
2     event(endMS_ENB(x1, x2, x3)) ⇝
3         event(begMS_ENB(x1, x2, x3)).
```

- Payload Secrecy

```
query attacker(payload).
```

The analysis results are the same as the ones in 4G authentication (Section VI). ProVerif proves all the properties except the payload secrecy, because the BS could choose not to enable encryption when communicating with the MS.

S9+. GSM I ∥ LTE II–IV ∥ GSM IV, conv(CK IK → Kc, MME), AV = 4G AV + CK + IK

Figure 19 shows details of the ProVerif model and the locations of the events which are used to specify the conditional payload secrecy and the authentication properties. Most of the code in this model is inherited from the 4G model and the GSM model. The MME uses the conversion function c3 to derive the GSM session key from the UMTS cipher and integrity keys:

```
fun c3(cipherKey, integKey): gsmKey.
```

There are four main processes in our model representing the behavior of the MS, the BS, the SN and the HN respectively.

```
1  (*AS SMC procedure in process MS*)
2  let pMSAS(kc_ms:gsmKey, imsi_ms: ident, cap_ms: bool) =
3      in(pubChannel, (=ASSMC, enableEnc_as_ms: bool));
4      event endMS_AS(imsi_ms, kc_ms, cap_ms);
5      out(pubChannel, CMComplete);
6      in(pubChannel, (=MSG, datamsg: bitstring));
7      out(pubChannel, sencrypt_as(secret, kc_ms));
8      if enableEnc_as_ms = true then
9          let msgcontent: bitstring = sdecrypt_as(datamsg, kc_ms)
10         in 0.
11
12 (*process respresenting MS*)
13 let processMS =
14     (*The identity of the MS*)
15     new imsi_ms: ident;
16     (*Pre-shared key*)
17     new ki: key;
18     (*Insert id/pre-shared key pair into the private table*)
19     insert keys(imsi_ms, ki);
20     (*MS non-deterministically choose
21         the capability of encryption*)
22     let cap_ms: bool = encCapability() in
23     out(pubChannel, (CAP, cap_ms));
24     out(pubChannel, (ID, imsi_ms));
25     in(pubChannel, (=CHALLENGE, rand_ms: nonce,
26         =f1(ki, rand_ms), snid_ms: ident));
27     let res_ms: resp = f2(ki, rand_ms) in
28     let ck_ms: cipherKey = f3(ki, rand_ms) in
29     let ik_ms: integKey = f4(ki, rand_ms) in
30     let kasme_ms: asmeKey = kdf_asme(ck_ms, ik_ms, snid_ms) in
31     event begSN(imsi_ms, snid_ms, kasme_ms);
```

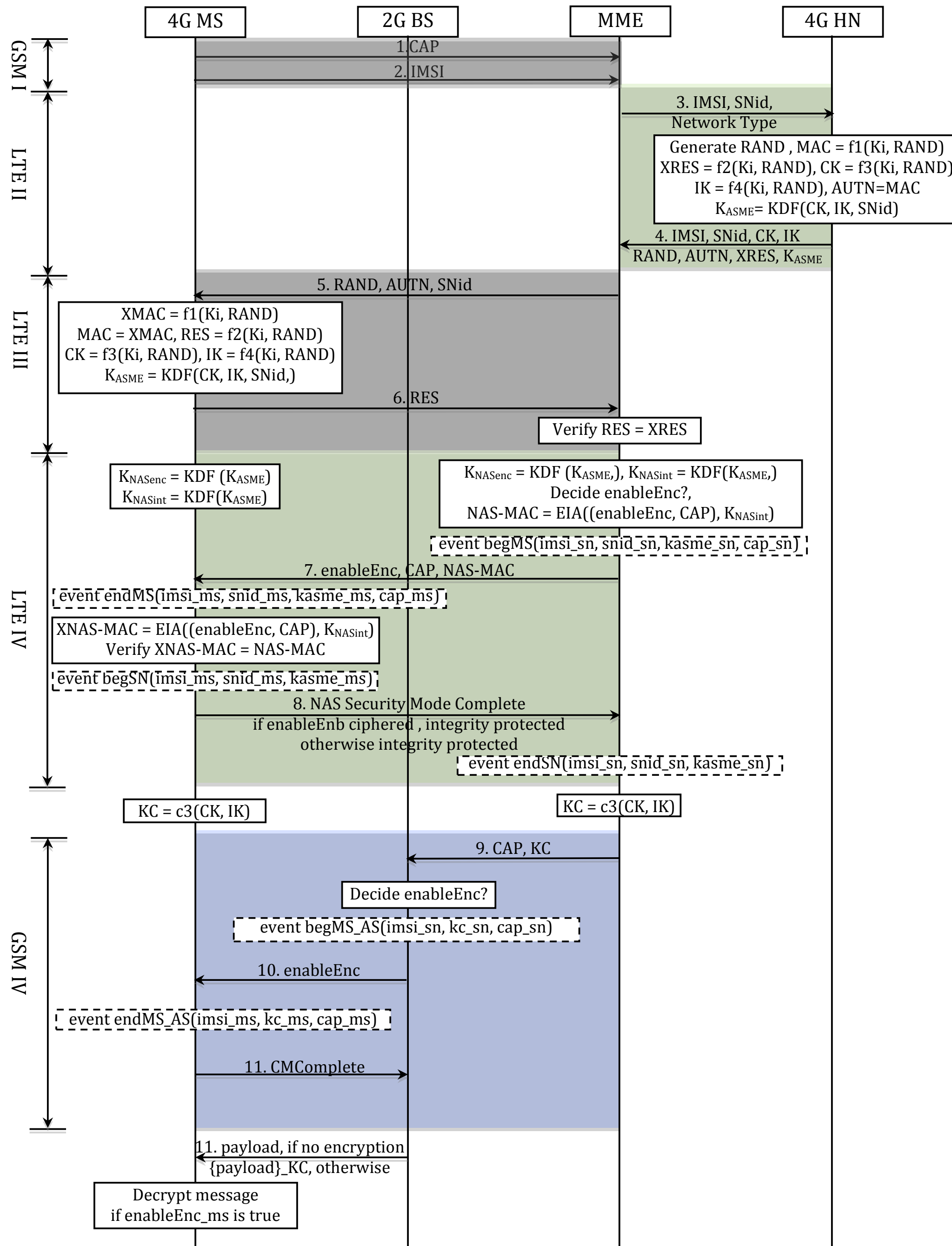

Fig. 19. Scenario S9+ annotated in accord with our model

```
32        out(pubChannel, (RES, res_ms));
33        (*NAS SMC procedure*)
34        let knasenc_ms: nasEncKey = kdf_nas_enc(kasme_ms) in
35        let knasint_ms: nasIntKey = kdf_nas_int(kasme_ms) in
36        in(pubChannel, (=NASSMC, enableEnc_nas_ms: bool, =cap_ms,
37            =finteg_nas((enableEnc_nas_ms, cap_ms), knasint_ms)));
38        event endMS(imsi_ms, snid_ms, kasme_ms, cap_ms);
39        (*NAS key secrecy*)
40        out(pubChannel, sencrypt_nas(secret, knasenc_ms));
41        out(pubChannel, senc_int_nas(secret, knasint_ms));
42        let kc_ms:gsmKey = c3(ck_ms, ik_ms) in
43        if enableEnc_nas_ms = false then
44            out(pubChannel, (NASSMComplete, nas_smcomplete_msg,
45                finteg_nas(nas_smcomplete_msg, knasint_ms)));
46            pMSAS(kc_ms, imsi_ms, cap_ms)
47        else
48            out(pubChannel, (NASSMComplete,
49                sencrypt_nas(nas_smcomplete_msg, knasenc_ms),
50                finteg_nas(sencrypt_nas(nas_smcomplete_msg,
51                knasenc_ms), knasint_ms)));
52            pMSAS(kc_ms, imsi_ms, cap_ms).
53
54    (*process representing e-nodeB*)
55    let processBS =
56        in(sChannelSnBts, (kc_bs: gsmKey,
57            imsi_bs: ident, cap_bs: bool));
58        event begMS_AS(imsi_bs, kc_bs, cap_bs);
59        out(pubChannel, (ASSMC, cap_bs));
60        in(pubChannel, =CMComplete);
61        if cap_bs = false then
62            event disableEnc;
63            out(pubChannel, (MSG, payload))
64        else
65            out(pubChannel, (MSG, sencrypt_as(payload, kc_bs))).
66
67    (*process representing MME*)
```

```
68  let processSN =
69      in(pubChannel, (=CAP, cap_sn: bool));
70      in(pubChannel, (=ID, imsi_sn: ident));
71      new snid_sn: ident;
72      out(secureChannel, (AV_REQ, imsi_sn, snid_sn));
73      in(secureChannel, (=AV, imsi_hn_sn: ident,
74          snid_hn_sn: ident, rand_sn: nonce,
75          xres_sn: resp, mac_sn: mac, kasme_sn: asmeKey,
76          ck_sn: cipherKey, ik_sn: integKey));
77      out(pubChannel, (CHALLENGE, rand_sn, mac_sn, snid_sn));
78      in(pubChannel, (=RES, =xres_sn));
79      event begMS(imsi_hn_sn, snid_hn_sn, kasme_sn, cap_sn);
80      (*NAS SMC procedure*)
81      let knasenc_sn: nasEncKey = kdf_nas_enc(kasme_sn) in
82      let knasint_sn: nasIntKey = kdf_nas_int(kasme_sn) in
83      out(pubChannel, (NASSMC, cap_sn, cap_sn,
84          finteg_nas((cap_sn, cap_sn), knasint_sn)));
85      in(pubChannel, (=NASSMComplete, msg_nas:bitstring,
86          =finteg_nas(msg_nas, knasint_sn)));
87      let kc_sn: gsmKey = c3(ck_sn, ik_sn) in
88      if cap_sn = true then
89          if sdecrypt_nas(msg_nas, knasenc_sn) =
90              nas_smcomplete_msg then
91              event endSN(imsi_hn_sn, snid_hn_sn, kasme_sn);
92              out(sChannelSnBts, (kc_sn, imsi_hn_sn, cap_sn))
93          else 0
94      else
95          if cap_sn = false then
96              if msg_nas = nas_smcomplete_msg then
97                  event endSN(imsi_hn_sn, snid_hn_sn, kasme_sn);
98                  out(sChannelSnBts, (kc_sn, imsi_hn_sn, cap_sn))
99              else 0
100         else 0.
101
102 (*process representing HN*)
103 let processHN =
104     in(secureChannel, (=AV_REQ, imsi_hn: ident, snid_hn: ident));
105     (*Generate athenication vectors*)
106     new rand_hn: nonce;
107     get keys(=imsi_hn, ki_hn) in
108     let mac_hn: mac = f1(ki_hn, rand_hn) in
109     let xres_hn: resp = f2(ki_hn, rand_hn) in
110     let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
111     let ik_hn: integKey = f4(ki_hn, rand_hn) in
112     let kasme_hn: asmeKey = kdf_asme(ck_hn, ik_hn, snid_hn) in
113     out(secureChannel, (AV, imsi_hn, snid_hn, rand_hn,
114         xres_hn, mac_hn, kasme_hn, ck_hn, ik_hn)).
```

The authentication vector request and response procedure is modeled in lines 72–76 and lines 104-114. The MME derives the GSM session key in line 87 and sends the key to the GSM BS in line 92 or 98 through the private channel between the MME and the GSM BS. The MS also computes the GSM session key in line 42. The code of the GSM SMC procedure (lines 3–5 and lines 59–60) is inherited from the GSM model.

Security Property Specifications and Findings Since the GSM BS uses the GSM session key K_c, the events used to specify the authentication of the BS to the MS use this key as one of the parameters:

```
1 event begMS_AS(ident, gsmKey, bool).
2 event endMS_AS(ident, gsmKey, bool).
```

The authentication properties between the MME and the MS are specified the same as the ones in the 4G model. We specify the security properties as following:

- Key Secrecy

```
1 not attacker(new ki).
2 query attacker(secret).
```

- Conditional Payload Secrecy

```
query attacker(payload)⇝event(disableEnc).
```

- Mutual Authentication between the MS and the MME

```
1 query x1: ident, x2: ident, x3: asmeKey;
2   event(endSN(x1, x2, x3)) ⇝ event(begSN(x1, x2, x3)).
3 query x1: ident, x2: ident, x3: asmeKey, x4: bool;
4   event(endMS(x1, x2, x3, x4)) ⇝
  event(begMS(x1, x2, x3, x4)).
```

- Authentication of the BS to the MS

```
1 query x1: ident, x2: gsmKey, x3: bool;
2   event(endMS_AS(x1, x2, x3)) ⇝
  event(begMS_AS(x1, x2, x3)).
```

- Payload Secrecy

```
query attacker(payload).
```

All the keys are proved to be remained secret. The conditional payload secrecy also holds, that means, if the encryption is enabled, the content of the encrypted data messages cannot be learned by the attacker. The mutual authentication properties between the MS and the MME are proved. However, because the BS is the GSM BS, the CMC attack is found when checking the authentication of the BS to the MS. As in other models, the payload secrecy could be violated because the BS could choose to disable encryption when communicating with the MS.

S10+. UMTS I || LTE II–IV || UMTS IV, AV = 4G AV + CK + IK

Figure 20 shows details of the ProVerif model and the locations of the events which are used to specify the conditional payload secrecy, authentication properties. Most of the code in this model is inherited from the 4G model and the UMTS model. There are four main processes in our model representing the behavior of the MS, the BS, the SN and the HN respectively.

```
1  (*AS SMC procedure in process MS*)
2  let pMSAS(ck_ms: cipherKey, ik_ms: integKey,
3  imsi_ms: ident, cap_ms: bool) =
4      in(pubChannel, (=ASSMC, =cap_ms, enableEnc_as_ms: bool,
5          =f9((cap_ms, enableEnc_as_ms), ik_ms)));
6      out(pubChannel, (ASSMComplete, as_smcomplete_msg,
7          f9(as_smcomplete_msg, ik_ms)));
8      event endMS_AS(imsi_ms, ck_ms, ik_ms, cap_ms);
9      in(pubChannel, (=MSG, datamsg: bitstring,
10         =f9(datamsg, ik_ms)));
11     out(pubChannel, sencrypt_as(secret, ck_ms));
12     out(pubChannel, senc_int_as(secret, ik_ms));
13     if enableEnc_as_ms = true then
14         let msgcontent: bitstring =
15             sdecrypt_as(datamsg, ck_ms) in 0.
16
17 (*process respresenting MS*)
18 let processMS =
19     (*The identity of the MS*)
20     new imsi_ms: ident;
21     (*Pre-shared key*)
22     new ki: key;
23     (*Insert id/pre-shared key pair into the private table*)
24     insert keys(imsi_ms, ki);
25     (*MS non-deterministically choose
26         the capability of encryption*)
27     let cap_ms: bool = encCapability() in
28     out(pubChannel, (CAP, cap_ms));
29     out(pubChannel, (ID, imsi_ms));
30     in(pubChannel, (=CHALLENGE, rand_ms: nonce,
31         =f1(ki, rand_ms), snid_ms: ident));
32     let res_ms: resp = f2(ki, rand_ms) in
33     let ck_ms: cipherKey = f3(ki, rand_ms) in
34     let ik_ms: integKey = f4(ki, rand_ms) in
35     let kasme_ms: asmeKey = kdf_asme(ck_ms, ik_ms, snid_ms) in
36     event begSN(imsi_ms, snid_ms, kasme_ms);
37     out(pubChannel, (RES, res_ms));
38     (*NAS SMC procedure*)
```

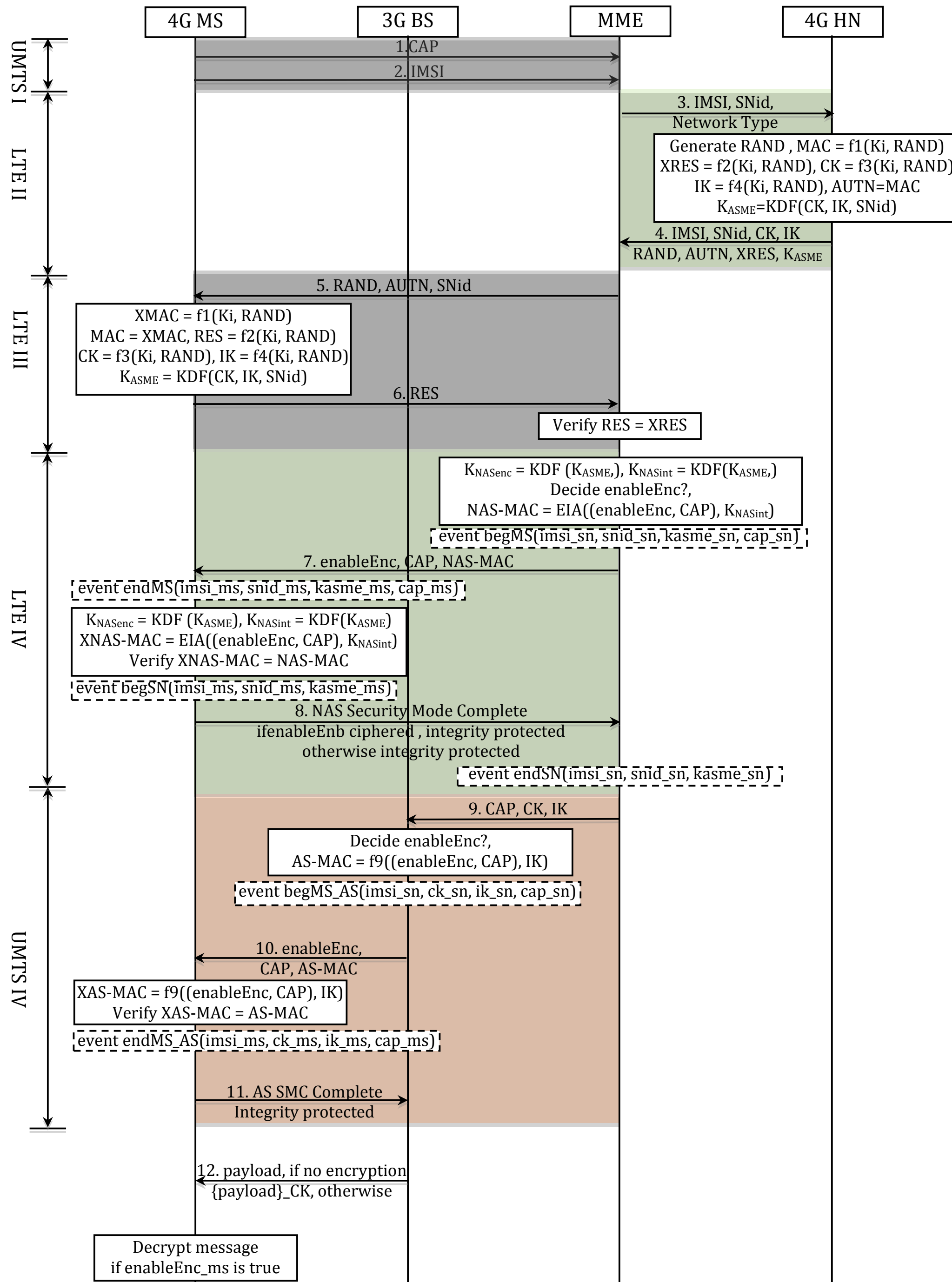

Fig. 20. Scenario S10+ annotated in accord with our model

```
39        let knasenc_ms: nasEncKey = kdf_nas_enc(kasme_ms) in
40        let knasint_ms: nasIntKey = kdf_nas_int(kasme_ms) in
41        in(pubChannel, (=NASSMC, enableEnc_nas_ms: bool, =cap_ms,
42            =finteg_nas((enableEnc_nas_ms, cap_ms), knasint_ms)));
43        event endMS(imsi_ms, snid_ms, kasme_ms, cap_ms);
44        (*NAS key secrecy*)
45        out(pubChannel, sencrypt_nas(secret, knasenc_ms));
46        out(pubChannel, senc_int_nas(secret, knasint_ms));
47        if enableEnc_nas_ms = false then
48            out(pubChannel, (NASSMComplete, nas_smcomplete_msg,
49                finteg_nas(nas_smcomplete_msg, knasint_ms)));
50            pMSAS(ck_ms, ik_ms, imsi_ms, cap_ms)
51        else
52            out(pubChannel, (NASSMComplete,
53                sencrypt_nas(nas_smcomplete_msg, knasenc_ms),
54                finteg_nas(sencrypt_nas(nas_smcomplete_msg,
55                    knasenc_ms), knasint_ms))); (*[Msg 8]*)
56            pMSAS(ck_ms, ik_ms, imsi_ms, cap_ms).
57
58    (*process representing e-nodeB*)
59    let processBS =
60        in(sChannelSnBts, (ck_bs: cipherKey, ik_bs: integKey,
61            imsi_bs: ident, cap_bs: bool));
62        event begMS_AS(imsi_bs, ck_bs, ik_bs, cap_bs);
63        out(pubChannel, (ASSMC, cap_bs, cap_bs,
64            f9((cap_bs, cap_bs), ik_bs)));
65        in(pubChannel, (=ASSMComplete, =as_smcomplete_msg,
66            =f9(as_smcomplete_msg, ik_bs)));
67        if cap_bs = false then
68            event disableEnc;
69            out(pubChannel, (MSG, payload, f9(payload, ik_bs)))
70        else
```

```
71          out(pubChannel, (MSG, sencrypt_as(payload, ck_bs),
72              f9(sencrypt_as(payload, ck_bs), ik_bs))).
73
74  (*process representing MME*)
75  let processSN =
76      in(pubChannel, (=CAP, cap_sn: bool));
77      in(pubChannel, (=ID, imsi_sn: ident));
78      new snid_sn: ident;
79      out(secureChannel, (AV_REQ, imsi_sn, snid_sn));
80      in(secureChannel, (=AV, =imsi_sn, snid_hn_sn: ident,
81          rand_sn: nonce, xres_sn: resp, mac_sn: mac,
82          kasme_sn: asmeKey, ck_sn: cipherKey, ik_sn: integKey));
83      out(pubChannel, (CHALLENGE, rand_sn, mac_sn, snid_sn));
84      in(pubChannel, (=RES, =xres_sn));
85      event begMS(imsi_sn, snid_hn_sn, kasme_sn, cap_sn);
86      (*NAS SMC procedure*)
87      let knasenc_sn: nasEncKey = kdf_nas_enc(kasme_sn) in
88      let knasint_sn: nasIntKey = kdf_nas_int(kasme_sn) in
89      out(pubChannel, (NASSMC, cap_sn, cap_sn,
90          finteg_nas((cap_sn, cap_sn), knasint_sn)));
91      in(pubChannel, (=NASSMComplete, msg_nas:bitstring,
92          =finteg_nas(msg_nas, knasint_sn)));
93      if cap_sn = true then
94          if sdecrypt_nas(msg_nas, knasenc_sn)
95          = nas_smcomplete_msg then
96              event endSN(imsi_sn, snid_hn_sn, kasme_sn);
97              out(sChannelSnBts, (ck_sn, ik_sn, imsi_sn, cap_sn))
98          else 0
99      else
100         if cap_sn = false then
101             if msg_nas = nas_smcomplete_msg then
102                 event endSN(imsi_sn, snid_hn_sn, kasme_sn);
103                 out(sChannelSnBts, (ck_sn, ik_sn,
104                     imsi_sn, cap_sn))
105             else 0
106         else 0.
107
108 (*process representing HN*)
109 let processHN =
110     in(secureChannel, (=AV_REQ, imsi_hn: ident, snid_hn: ident));
111     (*Generate athenication vectors*)
112     new rand_hn: nonce;
113     get keys(=imsi_hn, ki_hn) in
114     let mac_hn: mac = f1(ki_hn, rand_hn) in
115     let xres_hn: resp = f2(ki_hn, rand_hn) in
116     let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
117     let ik_hn: integKey = f4(ki_hn, rand_hn) in
118     let kasme_hn: asmeKey = kdf_asme(ck_hn, ik_hn, snid_hn) in
119     out(secureChannel, (AV, imsi_hn, snid_hn, rand_hn,
120         xres_hn, mac_hn, kasme_hn, ck_hn, ik_hn)).
```

The MME sends out the authentication vector request in line 79. Upon receiving the authentication request (line 110), the HN generates the 4G authentication vector based on the UMTS authentication vector. The HN then sends the 4G authentication vectors plus the *CK* and the *IK* to the MME (line 119–120). And the MME receives the authentication vectors in line 80–82. The NAS authentication procedure (lines 39–55 and lines 87–92) is the same as in the 4G model. In line 97 and line 103–104 , the MME sends the *CK* and the *IK* to the UMTS BS on the private channel. The code of the UMTS SMC procedure (lines 4–7 and lines 63–66) is inherited from the UMTS model.

Security Property Specifications and Findings Since the UMTS BS uses the CK and the IK instead of the keys derived from K_{eNB}, the events used to specify the authentication of the BS to the MS use the CK and the IK as their parameters:

```
1 event begMS_AS(ident, cipherKey, integKey, bool).
2 event endMS_AS(ident, cipherKey, integKey, bool).
```

We specify the security properties as following:

- Key Secrecy

```
1 not attacker(new ki).
2 query attacker(secret).
```

- Conditional Payload Secrecy

```
query attacker(payload)⇝event(disableEnc).
```

- Mutual Authentication between the MS and the SN

```
1 query x1: ident, x2: ident, x3: asmeKey;
2   event(endSN(x1, x2, x3)) ⇝ event(begSN(x1, x2, x3)).
3 query x1: ident, x2: ident, x3: asmeKey, x4: bool;
4   event(endMS(x1, x2, x3, x4)) ⇝
  event(begMS(x1, x2, x3, x4)).
```

- Authentication of the BS to the MS

```
1 query x1: ident, x2: cipherKey, x3: integKey, x4: bool;
2   event(endMS_AS(x1, x2, x3, x4)) ⇝
  event(begMS_AS(x1, x2, x3, x4)).
```

- Payload Secrecy

```
query attacker(payload).
```

The analysis results are the same as the ones in 4G authentication (Section VI). ProVerif proves all the properties except the payload secrecy, because the BS could choose not to enable encryption when communicating with the MS.

APPENDIX D
COMPLETE CODE LISTINGS FOR ALL SCENARIOS

All models are checked by ProVerif version 1.86pl4.

S1. GSM I – IV

```
(* Public channel between the MS and the SN *)
free pubChannel: channel.
(* Secure channel between the SN and the HN *)
free secureChannel: channel [private].

(* types *)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type sessKey.

(* constant message headers *)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const CMC: msgHdr.
const MSG: msgHdr.

(* Functions *)
fun a3(nonce, key) : resp.
fun a8(nonce, key): sessKey.
fun sencrypt(bitstring, sessKey): bitstring.

reduc forall m: bitstring, k: sessKey;
   sdecrypt(sencrypt(m, k), k) = m.

reduc encCapability() = true;
    encCapability() = false.

(* The key table consists of pairs
   (ident, key) shared between the MS and the HN.
   Table is not accessible by the attacker *)
table keys(ident, key).

free s: bitstring [private].
query attacker(s).

(* The standard secrecy queries of ProVerif only *)
(* deal with the secrecy of private free names*)
(* secretKc is secret if and only if all kcs are secret*)
free secretKc: bitstring [private].
query attacker(secretKc).

not attacker(new ki).

(* Authentication queries *)
event begSN(ident, sessKey).
event endSN(ident, sessKey).
event begMS(ident, sessKey).
event endMS(ident, sessKey).

query x1: ident, x2: sessKey;
  event(endSN(x1, x2)) ~> event(begSN(x1, x2)).
query x1: ident, x2: sessKey;
  event(endMS(x1, x2)) ~> event(begMS(x1, x2)).

event disableEnc.
(*When the attacker knows s,
    the event disableEnc has been executed.*)
query attacker(s) ~> event(disableEnc).


let processMS =
  (* The ident and pre-shared key of the MS *)
  new imsi_ms: ident;
  new ki: key;
  insert keys(imsi_ms, ki);
  (*MS non-deterministically choose
      the capability of encryption*)
  let cap_ms: bool = encCapability() in
  (*Send out cap_ms to SN[Msg 1]*)
  out(pubChannel, (CAP, cap_ms));
  (*Send out permanent ID [Msg 2]*)
  out(pubChannel, (ID, imsi_ms));
  (*Input challenge message from SN [Msg 5]*)
  in(pubChannel, (=CHALLENGE, rand_ms: nonce));
  (*Compute response and encryption key*)
  let res_ms: resp = a3(rand_ms, ki) in
  let kc_ms: sessKey = a8(rand_ms, ki) in
  (*MS is authenticating itself to SN*)
  event begSN(imsi_ms, kc_ms);
  (*Send out response to SN [Msg 6]*)
  out(pubChannel, (RES, res_ms));
  (*Receive GSM cipher mode command [Msg 7]*)
  in(pubChannel, (=CMC, enableEnc_ms: bool));
  event endMS(imsi_ms, kc_ms);
  (*Receive message from SN [Msg 8]*)
  in(pubChannel, (=MSG, msg: bitstring));
  out(pubChannel, sencrypt(secretKc, kc_ms));
  if enableEnc_ms = true then
    let msgcontent: bitstring = sdecrypt(msg, kc_ms) in
      0.

let processSN =
  (*Receive MS's capability [Msg 1]*)
  in(pubChannel, (=CAP, cap_sn: bool));
  (*Receive permanent ID [Msg 2]*)
  in(pubChannel, (=ID, imsi_sn: ident));
  (*Send out authentication vector request [Msg 3]*)
  out(secureChannel, (AV_REQ, imsi_sn));
  (*Receive authentication vector [Msg 4]*)
  in(secureChannel, (=AV, imsi_hn_sn: ident, rand_sn: nonce,
        xres_sn: resp, kc_sn: sessKey));
  (*Send authentication challenge to MS [Msg 5]*)
  out(pubChannel, (CHALLENGE, rand_sn));
  (*Receive response [Msg 6]*)
  in(pubChannel, (=RES, res_sn: resp));
  (*Check whether received response equal to expected response*)
  if res_sn = xres_sn then
    (*At this point, SN authenticated MS*)
    event endSN(imsi_hn_sn, kc_sn);
    (*SN decide whether to encrypt messages;
        based on received capabilities of MS*)
    (* let enableEnc_sn: bool = cap_sn in *)
    event begMS(imsi_hn_sn, kc_sn);
      (*Send out cipher mode command [Msg 7]*)
      (* out(pubChannel, (CMC, enableEnc_sn)); *)
      out(pubChannel, (CMC, cap_sn));
      out(pubChannel, sencrypt(secretKc, kc_sn));
      (* if enableEnc_sn = false then *)
    if cap_sn = false then
      event disableEnc;
      out(pubChannel, (MSG, s))
    else
        out(pubChannel, (MSG, sencrypt(s, kc_sn))).


let processHN =
  (*Receive authentication vector request [Msg 3]*)
  in(secureChannel, (=AV_REQ, imsi_hn: ident));
  (*Generate a fresh random number*)
  new rand_hn: nonce;
  (*Computes expected response and Kc*)
  get keys(=imsi_hn, ki_hn) in
  let xres_hn: resp = a3(rand_hn, ki_hn) in
  let kc_hn: sessKey = a8(rand_hn, ki_hn) in
  (*Send out authentication vector [Msg 4]*)
  out(secureChannel, (AV, imsi_hn, rand_hn, xres_hn, kc_hn));
  out(pubChannel, sencrypt(secretKc, kc_hn)).

process
  ((!processMS) | processSN | processHN)
```

S2. UMTS I – IV

```
(* Public channel between the MS and the SN *)
free pubChannel: channel.
(* Secure channel between the SN and the HN *)
free secureChannel: channel [private].

(* types *)
type key.
type ident.
type nonce.
```

```
type msgHdr.
type resp.
type cipherKey.
type integKey.
type mac.
type msgMac.

(* constant message headers *)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const SMC: msgHdr.
const MSG: msgHdr.

(* Functions *)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun f9(bitstring, integKey): msgMac.

fun sencrypt(bitstring, cipherKey): bitstring.

reduc forall m: bitstring, k: cipherKey;
    sdecrypt(sencrypt(m, k), k) = m.

reduc   encCapability() = true;
    encCapability() = false.

(* To test secrecy of the integrity key, *)
(* use them as session keys to encrypt a free private name *)
fun sencryptInteg(bitstring, integKey): bitstring.
reduc forall m: bitstring, k: integKey;
    sdecryptInteg(sencryptInteg(m, k), k) = m.

(* the table ident/keys
   The key table consists of pairs
   (ident, key) shared between MS and HN
   Table is not accessible by the attacker *)
table keys(ident, key).

free s: bitstring [private].
query attacker(s).

(* The standard secrecy queries of ProVerif only *)
(* deal with the secrecy of private free names*)
(* secretCk is secret if and only if all cks are secret*)
free secretCk: bitstring [private].
query attacker(secretCk).

(* secretIk is secret if and only if all iks are secret*)
free secretIk: bitstring [private].
query attacker(secretIk).

not attacker(new ki).

(* Authentication queries *)
event begSN(ident, cipherKey, integKey).
event endSN(ident, cipherKey, integKey).
event begMS(ident, cipherKey, integKey, bool).
event endMS(ident, cipherKey, integKey, bool).

query x1: ident, x2: cipherKey, x3: integKey;
    event(endSN(x1, x2, x3)) ⇝ event(begSN(x1, x2, x3)).
query x1: ident, x2: cipherKey, x3: integKey, x4: bool;
    event(endMS(x1, x2, x3, x4)) ⇝
event(begMS(x1, x2, x3, x4)).


event disableEnc.
query attacker(s) ⇝ event(disableEnc).

let processMS =
  (* The ident and pre-shared key of the MS *)
  new imsi_ms: ident;
  new ki: key;
  insert keys(imsi_ms, ki);
  (*MS non-deterministically
      choose the capability of encryption*)
  let cap_ms: bool = encCapability() in
  (*Send out cap_ms to SN[Msg 1]*)
  out(pubChannel, (CAP, cap_ms));
  (*Send out permanent ID [Msg 2]*)
  out(pubChannel, (ID, imsi_ms));
  (* Input challenge message from SN [Msg 5]*)
  in(pubChannel, (=CHALLENGE, rand_ms: nonce, mac_ms: mac));
  if f1(ki, rand_ms) = mac_ms then
    (*Compute response and encryption key*)
    let res_ms: resp = f2(ki, rand_ms) in
    let ck_ms: cipherKey = f3(ki, rand_ms) in
    let ik_ms: integKey = f4(ki, rand_ms) in
    (*MS is authenticating itself to SN*)
    event begSN(imsi_ms, ck_ms, ik_ms);
    (*Send out response to SN [Msg 6]*)
    out(pubChannel, (RES, res_ms));
    (*Receive GSM cipher mode command [Msg 7]*)
    in(pubChannel, (=SMC, enableEnc_ms: bool,
       =cap_ms, fresh_ms: nonce,
       =f9((enableEnc_ms, cap_ms, fresh_ms), ik_ms)));
    event endMS(imsi_ms, ck_ms, ik_ms, cap_ms);
    (*Receive message from SN [Msg 8]*)
    in(pubChannel, (=MSG, msg: bitstring, fresh_msg_ms: nonce,
       =f9((msg, fresh_msg_ms), ik_ms)));
    out(pubChannel, sencrypt(secretCk, ck_ms));
    out(pubChannel, sencryptInteg(secretIk, ik_ms));
    if enableEnc_ms = true then
      let msgcontent: bitstring = sdecrypt(msg, ck_ms) in
      0.

let processSN =
  (*Receive MS's capability [Msg 1]*)
  in(pubChannel, (=CAP, cap_sn: bool));
  (*Receive permanent ID [Msg 2]*)
  in(pubChannel, (=ID, imsi_sn: ident));
  (*Send out authentication vector request [Msg 3]*)
  out(secureChannel, (AV_REQ, imsi_sn));
  (*Receive authentication vector [Msg 4]*)
  in(secureChannel, (=AV, imsi_hn_sn: ident, rand_sn: nonce,
     xres_sn: resp, ck_sn: cipherKey, ik_sn: integKey,
     mac_sn: mac));
  (*Send authentication challenge to MS [Msg 5]*)
  out(pubChannel, (CHALLENGE, rand_sn, mac_sn));
  (*Receive response [Msg 6]*)
  in(pubChannel, (=RES, res_sn: resp));
  (*Check whether received response equal to expected response*)
  if res_sn = xres_sn then
    (*At this point, SN authenticated MS*)
    event endSN(imsi_hn_sn, ck_sn, ik_sn);
    new fresh_sn: nonce;
    (*SN decide whether to encrypt messages *)
    (*base on the received capabilities of MS*)
    (* let enableEnc_sn: bool = cap_sn in *)
    event begMS(imsi_hn_sn, ck_sn, ik_sn, cap_sn);
    (*Send out cipher mode command [Msg 7]*)
    out(pubChannel, (SMC, cap_sn, cap_sn, fresh_sn,
          f9((cap_sn, cap_sn, fresh_sn), ik_sn)));
    out(pubChannel, sencrypt(secretCk, ck_sn));
    out(pubChannel, sencryptInteg(secretIk, ik_sn));
    new fresh_msg_sn: nonce;
    (*Send out one message [Msg 8]*)
    (* if enableEnc_sn = false then *)
    if cap_sn = false then
      event disableEnc;
      out(pubChannel, (MSG, s, fresh_msg_sn,
         f9((s, fresh_msg_sn), ik_sn)))
    else
      out(pubChannel, (MSG, sencrypt(s, ck_sn), fresh_msg_sn,
          f9((sencrypt(s, ck_sn), fresh_msg_sn), ik_sn))).


let processHN =
  (*Receive authentication vector request [Msg 3]*)
  in(secureChannel, (=AV_REQ, imsi_hn: ident));
  (*Generate a fresh random number*)
  new rand_hn: nonce;
  (*Computes expected response and Kc*)
  get keys(=imsi_hn, ki_hn) in
  let mac_hn: mac = f1(ki_hn, rand_hn) in
  let xres_hn: resp = f2(ki_hn, rand_hn) in
  let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
  let ik_hn: integKey = f4(ki_hn, rand_hn) in
  (*Send out authentication vector [Msg 4]*)
  out(secureChannel, (AV, imsi_hn, rand_hn, xres_hn,
     ck_hn, ik_hn, mac_hn));
  out(pubChannel, sencrypt(secretCk, ck_hn));
```

```
  out(pubChannel, sencryptInteg(secretIk, ik_hn)).

process
  ((!processMS) | processSN | processHN )
```

S3. LTE I – V

```
(*Public channel between the MS and the SN*)
free pubChannel: channel.
(*Secure channel between the SN and the HN*)
free secureChannel: channel [private].
(*Secure channel between MME and BS*)
free sChannelSnBts: channel [private].

(*types*)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type cipherKey.
type integKey.
type mac.
type msgMac.
type asmeKey.
type nasEncKey.
type nasIntKey.
type enbKey.
type asEncKey.
type asIntKey.
type upEncKey.

(*constant message headers*)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const NASSMC: msgHdr.
const NASSMComplete: msgHdr.
const ASSMC: msgHdr.
const ASSMComplete: msgHdr.
const MSG: msgHdr.

(*Functions*)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun kdf_asme(cipherKey, integKey, ident): asmeKey.
fun kdf_nas_enc(asmeKey) : nasEncKey.
fun kdf_nas_int(asmeKey): nasIntKey.
fun finteg_nas(bitstring, nasIntKey): msgMac.
fun kdf_enb(asmeKey): enbKey.
fun kdf_as_enc(enbKey): asEncKey.
fun kdf_as_int(enbKey): asIntKey.
fun kdf_up_enc(enbKey): upEncKey.
fun finteg_as(bitstring, asIntKey): msgMac.

fun sencrypt_nas(bitstring, nasEncKey): bitstring.
reduc forall m: bitstring, k: nasEncKey;
    sdecrypt_nas(sencrypt_nas(m, k), k) = m.

fun sencrypt_as(bitstring, asEncKey): bitstring.
reduc forall m: bitstring, k: asEncKey;
    sdecrypt_as(sencrypt_as(m, k), k) = m.

(*Type Converter*)
fun bool2bitstring(bool): bitstring [data, typeConverter].

reduc   encCapability() = true;
        encCapability() = false.

(*the table ident/keys
   The key table consists of pairs
   (ident, key) shared between MS and HN
   Table is not accessible by the attacker*)
table keys(ident, key).

(*SMC command msg*)
free nas_smcomplete_msg: bitstring.
free as_smcomplete_msg: bitstring.

free payload: bitstring [private].
event disableEnc.
(*When the attacker knows s, the event
    disableEnc has been executed.*)
query attacker(payload) ⇝ event(disableEnc).
query attacker(payload).

free secret: bitstring [private].
query attacker(secret).
fun senc_int_nas(bitstring, nasIntKey): bitstring.
reduc forall m: bitstring, k: nasIntKey;
    sdec_in_nas(senc_int_nas(m, k), k) = m.
fun senc_int_as(bitstring, asIntKey): bitstring.
reduc forall m: bitstring, k: asIntKey;
    sdec_in_as(senc_int_as(m, k), k) = m.
fun senc_up(bitstring, upEncKey): bitstring.
reduc forall m: bitstring, k: upEncKey;
    sdec_up(senc_up(m, k), k) = m.


not attacker(new ki).

(*Authentication queries*)
event begSN(ident, ident, asmeKey).
event endSN(ident, ident, asmeKey).
event begMS(ident, ident, asmeKey, bool).
event endMS(ident, ident, asmeKey, bool).
event begENB(ident, enbKey).
event endENB(ident, enbKey).
event begMS_ENB(ident, enbKey, bool).
event endMS_ENB(ident, enbKey, bool).


query x1: ident, x2: ident, x3: asmeKey;
    event(endSN(x1, x2, x3)) ⇝  event(begSN(x1, x2, x3)).
query x1: ident, x2: ident, x3: asmeKey, x4: bool;
    event(endMS(x1, x2, x3, x4)) ⇝
event(begMS(x1, x2, x3, x4)).
query x1: ident, x2: enbKey;
    event(endENB(x1, x2)) ⇝ event(begENB(x1, x2)).
query x1: ident, x2: enbKey, x3: bool;
    event(endMS_ENB(x1, x2, x3)) ⇝
event(begMS_ENB(x1, x2, x3)).


(*AS SMC procedure in process MS*)
let pMSAS(kasme_ms: asmeKey, imsi_ms: ident, cap_ms: bool) =
    let kenb_ms: enbKey = kdf_enb(kasme_ms) in
    let kasenc_ms: asEncKey = kdf_as_enc(kenb_ms) in
    let kasint_ms: asIntKey = kdf_as_int(kenb_ms) in
    let kupenc_ms: upEncKey = kdf_up_enc(kenb_ms) in
    in(pubChannel, (=ASSMC, enableEnc_as_ms: bool,
        =finteg_as(bool2bitstring(enableEnc_as_ms), kasint_ms)));
    event begENB(imsi_ms, kenb_ms);
    out(pubChannel, (ASSMComplete, as_smcomplete_msg,
        finteg_as(as_smcomplete_msg, kasint_ms))); (*[Msg 11]*)
    event endMS_ENB(imsi_ms, kenb_ms, cap_ms);
    in(pubChannel, (=MSG, datamsg: bitstring,
        =finteg_as(datamsg, kasint_ms))); (*[Msg 12]*)
    out(pubChannel, sencrypt_as(secret, kasenc_ms));
    out(pubChannel, senc_int_as(secret, kasint_ms));
    out(pubChannel, senc_up(secret, kupenc_ms));

    if enableEnc_as_ms = true then
        let msgcontent: bitstring = sdecrypt_as(datamsg, kasenc_ms)
        in 0.

(*process respresenting MS*)
let processMS =
    (*The identity of the MS*)
    new imsi_ms: ident;
    (*Pre-shared key*)
    new ki: key;
    (*Insert id/pre-shared key pair into the private table*)
    insert keys(imsi_ms, ki);
    (*MS non-deterministically choose the capability of encryption*)
    let cap_ms: bool = encCapability() in
    out(pubChannel, (CAP, cap_ms));
    out(pubChannel, (ID, imsi_ms));
    in(pubChannel, (=CHALLENGE, rand_ms: nonce,
        =f1(ki, rand_ms), snid_ms: ident));
    let res_ms: resp = f2(ki, rand_ms) in
    let ck_ms: cipherKey = f3(ki, rand_ms) in
```

```
    let ik_ms: integKey = f4(ki, rand_ms) in
    let kasme_ms: asmeKey = kdf_asme(ck_ms, ik_ms, snid_ms) in
    out(pubChannel, (RES, res_ms));
    (*NAS SMC procedure*)
    let knasenc_ms: nasEncKey = kdf_nas_enc(kasme_ms) in
    let knasint_ms: nasIntKey = kdf_nas_int(kasme_ms) in
    in(pubChannel, (=NASSMC, enableEnc_nas_ms: bool, =cap_ms,
        =finteg_nas((enableEnc_nas_ms, cap_ms), knasint_ms)));
    event endMS(imsi_ms, snid_ms, kasme_ms, cap_ms);
    (*NAS key secrecy*)
    out(pubChannel, sencrypt_nas(secret, knasenc_ms));
    out(pubChannel, senc_int_nas(secret, knasint_ms));
    event begSN(imsi_ms, snid_ms, kasme_ms);
    if enableEnc_nas_ms = false then
        out(pubChannel, (NASSMComplete, nas_smcomplete_msg,
            finteg_nas(nas_smcomplete_msg, knasint_ms)));
        pMSAS(kasme_ms, imsi_ms, cap_ms)
    else
        out(pubChannel, (NASSMComplete,
            sencrypt_nas(nas_smcomplete_msg, knasenc_ms),
            finteg_nas(sencrypt_nas(nas_smcomplete_msg,
            knasenc_ms), knasint_ms)));
        pMSAS(kasme_ms, imsi_ms, cap_ms).



(*process representing e-nodeB*)
let processENB =
    in(sChannelSnBts, (kasme_enb: asmeKey,
        imsi_enb: ident, cap_enb: bool));
    let kenb_enb: enbKey = kdf_enb(kasme_enb) in
    let kasenc_enb: asEncKey = kdf_as_enc(kenb_enb) in
    let kasint_enb: asIntKey = kdf_as_int(kenb_enb) in
    let kupenc_enb: upEncKey = kdf_up_enc(kenb_enb) in
    event begMS_ENB(imsi_enb, kenb_enb, cap_enb);
    out(pubChannel, (ASSMC, cap_enb,
        finteg_as(bool2bitstring(cap_enb), kasint_enb)));
    in(pubChannel, (=ASSMComplete, =as_smcomplete_msg,
        =finteg_as(as_smcomplete_msg, kasint_enb)));
    event endENB(imsi_enb, kenb_enb);
    if cap_enb = false then
      event disableEnc;
      out(pubChannel, (MSG, payload,
          finteg_as(payload, kasint_enb)))
    else
      out(pubChannel, (MSG, sencrypt_as(payload, kasenc_enb),
          finteg_as(sencrypt_as(payload, kasenc_enb),
              kasint_enb))).

(*process representing MME*)
let processMME =
    in(pubChannel, (=CAP, cap_sn: bool));
    in(pubChannel, (=ID, imsi_sn: ident));
    new snid_sn: ident;
    out(secureChannel, (AV_REQ, imsi_sn, snid_sn));
    in(secureChannel, (=AV, =imsi_sn, =snid_sn, rand_sn: nonce,
        xres_sn: resp, mac_sn: mac, kasme_sn: asmeKey));
    out(pubChannel, (CHALLENGE, rand_sn, mac_sn, snid_sn));
    in(pubChannel, (=RES, =xres_sn));
    event begMS(imsi_sn, snid_sn, kasme_sn, cap_sn);
    (*NAS SMC procedure*)
    let knasenc_sn: nasEncKey = kdf_nas_enc(kasme_sn) in
    let knasint_sn: nasIntKey = kdf_nas_int(kasme_sn) in
    out(pubChannel, (NASSMC, cap_sn, cap_sn,
        finteg_nas((cap_sn, cap_sn), knasint_sn)));
    in(pubChannel, (=NASSMComplete, msg_nas:bitstring,
        =finteg_nas(msg_nas, knasint_sn)));
    if cap_sn = true then
        if sdecrypt_nas(msg_nas, knasenc_sn) =
            nas_smcomplete_msg then
                event endSN(imsi_sn, snid_sn, kasme_sn);
                out(sChannelSnBts, (kasme_sn, imsi_sn, cap_sn))
        else 0
    else
        if cap_sn = false then
            if msg_nas = nas_smcomplete_msg then
                event endSN(imsi_sn, snid_sn, kasme_sn);
                out(sChannelSnBts, (kasme_sn, imsi_sn, cap_sn))
            else 0
        else 0.

(*process representing HN*)
let processHN =
    in(secureChannel, (=AV_REQ, imsi_hn: ident,
        snid_hn: ident));
    (*Generate athenication vectors*)
    new rand_hn: nonce;
    get keys(=imsi_hn, ki_hn) in
    let mac_hn: mac = f1(ki_hn, rand_hn) in
    let xres_hn: resp = f2(ki_hn, rand_hn) in
    let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
    let ik_hn: integKey = f4(ki_hn, rand_hn) in
    let kasme_hn: asmeKey = kdf_asme(ck_hn, ik_hn, snid_hn) in
    out(secureChannel, (AV, imsi_hn, snid_hn, rand_hn,
        xres_hn, mac_hn, kasme_hn)).


process
    ((!processMS) | processMME | processENB | processHN)
```

S4. GSM I – IV, convert(3G AV → 2G AV)

```
(* Public channel between the MS and the SN *)
free pubChannel: channel.
(* Secure channel between the SN and the HN *)
free secureChannel: channel [private].

(* types *)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type cipherKey.
type integKey.
type mac.
type msgMac.
type sessKey.


(* constant message headers *)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const CMC: msgHdr.
const MSG: msgHdr.


(* Functions *)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun f9(bitstring, integKey): msgMac.
fun c2(resp):resp.
fun c3(cipherKey, integKey): sessKey.
fun sencrypt(bitstring, sessKey): bitstring.

reduc forall m: bitstring, k: sessKey;
    sdecrypt(sencrypt(m, k), k) = m.

(* To test secrecy of the cipher key, *)
(* use them as session keys to encrypt a free private name *)
fun sencryptCipher(bitstring, cipherKey): bitstring.
reduc forall m: bitstring, k: cipherKey;
    sdecryptCipher(sencryptCipher(m, k), k) = m.

reduc   encCapability() = true;
    encCapability() = false.

(* the table ident/keys
   The key table consists of pairs
   (ident, key) shared between the MS and the HN.
   Table is not accessible by the attacker *)
table keys(ident, key).

free s: bitstring [private].
query attacker(s).

(* The standard secrecy queries of ProVerif only *)
(* deal with the secrecy of private free names*)
(* secretKc is secret if and only if all kcs is secret*)
free secretKc: bitstring [private].
query attacker(secretKc).
```

```
(* secretCk is secret if and only if all cks are secret*)
free secretCk: bitstring [private].
query attacker(secretCk).

(* If KC and CK are secret, then IK is secret *)


not attacker(new ki).

(* Authentication queries *)
event begSN(ident, sessKey).
event endSN(ident, sessKey).
event begMS(ident, sessKey).
event endMS(ident, sessKey).

query x1: ident, x2: sessKey; event(endSN(x1, x2)) ⇝
event(begSN(x1, x2)).
query x1: ident, x2: sessKey; event(endMS(x1, x2)) ⇝
event(begMS(x1, x2)).

event disableEnc.
(*When the attacker knows s, the event
    disableEnc has been executed.*)
query attacker(s) ⇝ event(disableEnc).

(*Process respresenting MS*)
let processMS =
  (* The ident and pre-shared key of the MS *)
  new imsi_ms: ident;
  new ki: key;
  insert keys(imsi_ms, ki);
  (*MS non-deterministically choose
      the capability of encryption*)
  let cap_ms: bool = encCapability() in
  (*Send out cap_ms to SN[Message 1]*)
  out(pubChannel, (CAP, cap_ms));
  (*Send out permanent ID [Message 2]*)
  out(pubChannel, (ID, imsi_ms));
  (*Input challenge message from SN [Message 5]*)
  in(pubChannel, (=CHALLENGE, rand_ms: nonce));
  (*Compute response and encryption key*)
  let res_ms_u: resp = f2(ki, rand_ms) in
  let ck_ms: cipherKey = f3(ki, rand_ms) in
  let ik_ms: integKey = f4(ki, rand_ms) in
  let res_ms_g: resp = c2(res_ms_u) in
  let kc_ms: sessKey = c3(ck_ms, ik_ms) in
  (*MS is authenticating itself to SN*)
  event begSN(imsi_ms, kc_ms);
  (*Send out response to SN [Message 6]*)
  out(pubChannel, (RES, res_ms_g));
  (*Receive GSM cipher mode command [Message 7]*)
  in(pubChannel, (=CMC, enableEnc_ms: bool));
  event endMS(imsi_ms, kc_ms);
  (*Receive message from SN [Message 8]*)
  in(pubChannel, (=MSG, msg: bitstring));
  out(pubChannel, sencrypt(secretKc, kc_ms));
  out(pubChannel, sencryptCipher(secretCk, ck_ms));
  if enableEnc_ms = true then
    let msgcontent: bitstring = sdecrypt(msg, kc_ms) in
      0.

(*Process respresenting SN*)
let processSN =
  (*Receive MS's capability [Message 1]*)
  in(pubChannel, (=CAP, cap_sn: bool));
  (*Receive permanent ID [Message 2]*)
  in(pubChannel, (=ID, imsi_sn: ident));
  (*Send out authentication vector request [Message 3]*)
  out(secureChannel, (AV_REQ, imsi_sn));
  (*Receive authentication vector [Message 4]*)
  in(secureChannel, (=AV, imsi_hn_sn: ident, rand_sn: nonce,
      xres_sn: resp, kc_sn: sessKey));
  (*Send authentication challenge to MS [Message 5]*)
  out(pubChannel, (CHALLENGE, rand_sn));
  (*Receive response [Message 6]*)
  in(pubChannel, (=RES, res_sn: resp));
  (*Check whether received response matches expected response*)
  if res_sn = xres_sn then
    (*At this point, SN authenticated MS*)
    event endSN(imsi_hn_sn, kc_sn);
    (* SN decide whether to encrypt messages *)
    (* base on the received capabilities of MS*)
    (* let enableEnc_sn: bool = cap_sn in *)
    event begMS(imsi_hn_sn, kc_sn);
    (* Send out cipher mode command [Message 7]*)
    (* out(pubChannel, (CMC, enableEnc_sn)); *)
    out(pubChannel, (CMC, cap_sn));
      out(pubChannel, sencrypt(secretKc, kc_sn));
      (* if enableEnc_sn = false then *)
    if cap_sn = false then
      event disableEnc;
      out(pubChannel, (MSG, s))
    else
        out(pubChannel, (MSG, sencrypt(s, kc_sn))).


(*Process representing HN*)
let processHN =
  (*Receive authentication vector request [Message 3]*)
  in(secureChannel, (=AV_REQ, imsi_hn: ident));
  (*Generate a fresh random number*)
  new rand_hn: nonce;
  (*Computes expected response and Kc*)
  get keys(=imsi_hn, ki_hn) in
  let mac_hn: mac = f1(ki_hn, rand_hn) in
  let xres_hn_u: resp = f2(ki_hn, rand_hn) in
  let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
  let ik_hn: integKey = f4(ki_hn, rand_hn) in
  let xres_hn_g: resp = c2(xres_hn_u) in
  let kc_hn: sessKey = c3(ck_hn, ik_hn) in
  (*Send out authentication vector [Message 4]*)
  out(secureChannel, (AV, imsi_hn, rand_hn, xres_hn_g, kc_hn));
  out(pubChannel, sencryptCipher(secretCk, ck_hn));
  out(pubChannel, sencrypt(secretKc, kc_hn)).

process
  ((!processMS) | processSN | processHN )
```

S5. GSM I–III || UMTS IV, conv($Kc \rightarrow CK\ IK$, VLR/SGSN)

```
(*param verboseClauses = explained.*)

(* Public channel between the MS and the SN *)
free pubChannel: channel.
(* Secure channel between the SN and the HN *)
free secureChannel: channel [private].

(* types *)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type sessKey.
type cipherKey.
type integKey.
type mac.
type msgMac.

(* constant message headers *)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const SMC: msgHdr.
const MSG: msgHdr.

(* Functions *)
fun a3(nonce, key) : resp.
fun a8(nonce, key): sessKey.
fun c4(sessKey): cipherKey.
fun c5(sessKey): integKey.
fun f9(bitstring, integKey): msgMac.
fun sencrypt(bitstring, cipherKey): bitstring.

reduc forall m: bitstring, k: cipherKey;
    sdecrypt(sencrypt(m, k), k) = m.

reduc   encCapability() = true;
    encCapability() = false.

(* To test secrecy of the integrity key, *)
(* use them as session keys to encrypt a free private name *)
fun sencryptInteg(bitstring, integKey): bitstring.
```

```
reduc forall m: bitstring, k: integKey;
    sdecryptInteg(sencryptInteg(m, k), k) = m.

(* the table ident/keys
   The key table consists of pairs
   (ident, key) shared between the MS and the HN.
   Table is not accessible by the attacker *)
table keys(ident, key).

free s: bitstring [private].
query attacker(s).

(* The standard secrecy queries of ProVerif only *)
(* deal with the secrecy of private free names*)
(* secretCk is secret if and only if all cks are secret*)
free secretCk: bitstring [private].
query attacker(secretCk).

(* secretIk is secret if and only if all iks are secret*)
free secretIk: bitstring [private].
query attacker(secretIk).

(* If IK and CK are secret, then KC is secret. *)
(* Because CK and IK are computed from KC by public functions *)

not attacker(new ki).

(* Authentication queries *)
event begSN(ident, sessKey).
event endSN(ident, sessKey).
event begMS(ident, cipherKey, integKey, bool).
event endMS(ident, cipherKey, integKey, bool).

query x1: ident, x2: sessKey;
    event(endSN(x1, x2)) ⇝ event(begSN(x1, x2)).
query x1: ident, x2: cipherKey, x3: integKey, x4: bool;
    event(endMS(x1, x2, x3, x4)) ⇝
event(begMS(x1, x2, x3, x4)).

event disableEnc.
(*When the attacker knows s, the event
    disableEnc has been executed.*)
query attacker(s) ⇝ event(disableEnc).

(*Process respresenting MS*)
let processMS =
  (* The ident and pre-shared key of the MS *)
  new imsi_ms: ident;
  new ki: key;
  insert keys(imsi_ms, ki);
  (*MS non-deterministically choose the
      capability of encryption*)
  let cap_ms: bool = encCapability() in
  (*Send out cap_ms to SN[Msg 1]*)
  out(pubChannel, (CAP, cap_ms));
  (*Send out permanent ID [Msg 2]*)
  out(pubChannel, (ID, imsi_ms));
  (*Input challenge message from SN [Msg 5]*)
  in(pubChannel, (=CHALLENGE, rand_ms: nonce));
  (*Compute response and encryption key*)
  let res_ms: resp = a3(rand_ms, ki) in
  let kc_ms: sessKey = a8(rand_ms, ki) in
  (*MS is authenticating itself to SN*)
  event begSN(imsi_ms, kc_ms);
  (*Send out response to SN [Msg 6]*)
  out(pubChannel, (RES, res_ms));
  (*Convert Kc into UMTS keys*)
  let ck_ms:cipherKey = c4(kc_ms) in
  let ik_ms:integKey = c5(kc_ms) in
  (*Receive GSM cipher mode command [Msg 7]*)
  in(pubChannel, (=SMC, enableEnc_ms: bool,
      =cap_ms, fresh_ms: nonce,
      =f9((enableEnc_ms, cap_ms, fresh_ms), ik_ms)));
  event endMS(imsi_ms, ck_ms, ik_ms, cap_ms);
  (*Receive message from SN [Msg 8]*)
  in(pubChannel, (=MSG, msg: bitstring, fresh_msg_ms: nonce,
      =f9((msg, fresh_msg_ms), ik_ms)));
    out(pubChannel, sencrypt(secretCk, ck_ms));
        out(pubChannel, sencryptInteg(secretIk, ik_ms));
  if enableEnc_ms = true then
    let msgcontent: bitstring = sdecrypt(msg, ck_ms) in
    0.

(*Process respresenting SN*)
let processSN =
  (*Receive MS's capability [Msg 1]*)
  in(pubChannel, (=CAP, cap_sn: bool));
  (*Receive permanent ID [Msg 2]*)
  in(pubChannel, (=ID, imsi_sn: ident));
  (*Send out authentication vector request [Msg 3]*)
  out(secureChannel, (AV_REQ, imsi_sn));
  (*Receive authentication vector [Msg 4]*)
  in(secureChannel, (=AV, imsi_hn_sn: ident, rand_sn: nonce,
      xres_sn: resp, kc_sn: sessKey));
  (*Send authentication challenge to MS [Msg 5]*)
  out(pubChannel, (CHALLENGE, rand_sn));
  (*Receive response [Msg 6]*)
  in(pubChannel, (=RES, res_sn: resp));
  (*Check whether received response equal to expected response*)
  if res_sn = xres_sn then
    (*At this point, SN authenticated MS*)
    event endSN(imsi_hn_sn, kc_sn);
    (*Convert Kc into UMTS keys*)
    let ck_sn:cipherKey = c4(kc_sn) in
    let ik_sn:integKey = c5(kc_sn) in
    (*SN decide whether to encrypt messages *)
    (* base on the received capabilities of MS*)
    (* let enableEnc_sn: bool = cap_sn in *)
    new fresh_sn: nonce;
    event begMS(imsi_hn_sn, ck_sn, ik_sn, cap_sn);
    (*Send out cipher mode command [Msg 7]*)
    (* out(pubChannel, (SMC, enableEnc_sn, cap_sn, fresh_sn,
        f9((enableEnc_sn, cap_sn, fresh_sn), ik_sn))); *)
    out(pubChannel, (SMC, cap_sn, cap_sn, fresh_sn,
        f9((cap_sn, cap_sn, fresh_sn), ik_sn)));
    out(pubChannel, sencrypt(secretCk, ck_sn));
    out(pubChannel, sencryptInteg(secretIk, ik_sn));
    new fresh_msg_sn: nonce;
    (*Send out one message [Msg 8]*)
    (* if enableEnc_sn = false then *)
    if cap_sn = false then
      event disableEnc;
      out(pubChannel, (MSG, s, fresh_msg_sn,
          f9((s, fresh_msg_sn), ik_sn)))
    else
        out(pubChannel, (MSG, sencrypt(s, ck_sn), fresh_msg_sn,
            f9((sencrypt(s, ck_sn), fresh_msg_sn), ik_sn))).


(*Process representing HN*)
let processHN =
  (*Receive authentication vector request [Msg 3]*)
  in(secureChannel, (=AV_REQ, imsi_hn: ident));
  (*Generate a fresh random number*)
  new rand_hn: nonce;
  (*Computes expected response and Kc*)
  get keys(=imsi_hn, ki_hn) in
  let xres_hn: resp = a3(rand_hn, ki_hn) in
  let kc_hn: sessKey = a8(rand_hn, ki_hn) in
  (*Send out authentication vector [Msg 4]*)
  out(secureChannel, (AV, imsi_hn, rand_hn, xres_hn, kc_hn)).


process
  ((!processMS) | processSN | processHN )
```

S6. UMTS I–III || GSM IV, conv(CK IK → Kc, VLR/SGSN)

```
(* Public channel between the MS and the SN *)
free pubChannel: channel.
(* Secure channel between the MS and the HN *)
free secureChannel: channel [private].

(* types *)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type cipherKey.
type integKey.
type sessKey.
type mac.
type msgMac.

(* constant message headers *)
const CAP: msgHdr.
```

```
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const CMC: msgHdr.
const MSG: msgHdr.

(* Functions *)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun f9(bitstring, integKey): msgMac.
fun c3(cipherKey, integKey): sessKey.
fun sencrypt(bitstring, sessKey): bitstring.

reduc forall m: bitstring, k: sessKey;
    sdecrypt(sencrypt(m, k), k) = m.

(* To test secrecy of the cipher key, *)
(* use them as session keys to encrypt a free private name *)
fun sencryptCipher(bitstring, cipherKey): bitstring.
reduc forall m: bitstring, k: cipherKey;
    sdecryptCipher(sencryptCipher(m, k), k) = m.

reduc   encCapability() = true;
    encCapability() = false.

(* the table ident/keys
   The key table consists of pairs
   (ident, key) shared between the MS and the HN.
   Table is not accessible by the attacker *)
table keys(ident, key).

free s: bitstring [private].
query attacker(s).

(* The standard secrecy queries of ProVerif only *)
(* deal with the secrecy of private free names*)
(* secretKc is secret if and only if all kcs is secret*)
free secretKc: bitstring [private].
query attacker(secretKc).

(* secretCk is secret if and only if all cks are secret*)
free secretCk: bitstring [private].
query attacker(secretCk).

(* If KC and CK are secret, then IK is secret *)

not attacker(new ki).

(* Authentication queries *)
event begSN(ident, cipherKey, integKey).
event endSN(ident, cipherKey, integKey).
event begMS(ident, sessKey).
event endMS(ident, sessKey).

query x1: ident, x2: cipherKey, x3: integKey;
    event(endSN(x1, x2, x3)) ⇝ event(begSN(x1, x2, x3)).
query x1: ident, x2: sessKey;
    event(endMS(x1, x2)) ⇝ event(begMS(x1, x2)).

event disableEnc.
(*When the attacker knows s, the event
    disableEnc has been executed.*)
query attacker(s) ⇝ event(disableEnc).

(*Process respresenting MS*)
let processMS =
  (* The ident and pre-shared key of the mobile station *)
  new imsi_ms: ident;
  new ki: key;
  insert keys(imsi_ms, ki);
  (*MS non-deterministically choose
      the capability of encryption*)
  let cap_ms: bool = encCapability() in
  (*Send out cap_ms to SN[Msg 1]*)
  out(pubChannel, (CAP, cap_ms));
  (*Send out permanent ID [Msg 2]*)
  out(pubChannel, (ID, imsi_ms));
  (*Input challenge message from SN [Msg 5]*)
  in(pubChannel, (=CHALLENGE, rand_ms: nonce, mac_ms: mac));
  if f1(ki, rand_ms) = mac_ms then
    (*Compute response and encryption key*)
    let res_ms: resp = f2(ki, rand_ms) in
    let ck_ms: cipherKey = f3(ki, rand_ms) in
    let ik_ms: integKey = f4(ki, rand_ms) in
    (*MS is authenticating itself to SN*)
    event begSN(imsi_ms, ck_ms, ik_ms);
    (*Send out response to SN [Msg 6]*)
    out(pubChannel, (RES, res_ms));
    let kc_ms: sessKey = c3(ck_ms, ik_ms) in
    (*Receive GSM cipher mode command [Msg 7]*)
    in(pubChannel, (=CMC, enableEnc_ms: bool));
    event endMS(imsi_ms, kc_ms);
    (*Receive message from SN [Msg 8]*)
    in(pubChannel, (=MSG, msg: bitstring));
        out(pubChannel, sencrypt(secretKc, kc_ms));
      out(pubChannel, sencryptCipher(secretCk, ck_ms));
    if enableEnc_ms = true then
    let msgcontent: bitstring = sdecrypt(msg, kc_ms) in
      0.

(*Process respresenting SN*)
let processSN =
  (*Receive MS's capability [Msg 1]*)
  in(pubChannel, (=CAP, cap_sn: bool));
  (*Receive permanent ID [Msg 2]*)
  in(pubChannel, (=ID, imsi_sn: ident));
  (*Send out authentication vector request [Msg 3]*)
  out(secureChannel, (AV_REQ, imsi_sn));
  (*Receive authentication vector [Msg 4]*)
  in(secureChannel, (=AV, imsi_hn_sn: ident, rand_sn: nonce,
      xres_sn: resp, ck_sn: cipherKey, ik_sn: integKey, mac_sn: mac));
  (*Send authentication challenge to MS [Msg 5]*)
  out(pubChannel, (CHALLENGE, rand_sn, mac_sn));
  (*Receive response [Msg 6]*)
  in(pubChannel, (=RES, res_sn: resp));
  (*Check whether received response equal to expected response*)
  if res_sn = xres_sn then
    (*At this point, SN authenticated MS*)
    event endSN(imsi_hn_sn, ck_sn, ik_sn);
    let kc_sn: sessKey = c3(ck_sn, ik_sn) in
    (*SN decide whether to encrypt messages *)
    (* base on the received capabilities of MS*)
    (* let enableEnc_sn: bool = cap_sn in *)
    event begMS(imsi_hn_sn, kc_sn);
    (*Send out cipher mode command [Msg 7]*)
    (* out(pubChannel, (CMC, enableEnc_sn)); *)
    out(pubChannel, (CMC, cap_sn));
        out(pubChannel, sencrypt(secretKc, kc_sn));
      out(pubChannel, sencryptCipher(secretCk, ck_sn));
      (* if enableEnc_sn = false then *)
    if cap_sn = false then
      event disableEnc;
      out(pubChannel, (MSG, s)) (*[Msg 8]*)
    else
        out(pubChannel, (MSG, sencrypt(s, kc_sn))). (*[Msg 8]*)


(*Process representing HN*)
let processHN =
  (*Receive authentication vector request [Msg 3]*)
  in(secureChannel, (=AV_REQ, imsi_hn: ident));
  (*Generate a fresh random number*)
  new rand_hn: nonce;
  (*Computes expected response and Kc*)
  get keys(=imsi_hn, ki_hn) in
  let mac_hn: mac = f1(ki_hn, rand_hn) in
  let xres_hn: resp = f2(ki_hn, rand_hn) in
  let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
  let ik_hn: integKey = f4(ki_hn, rand_hn) in
  (*Send out authentication vector [Msg 4]*)
  out(secureChannel, (AV, imsi_hn, rand_hn, xres_hn,
      ck_hn, ik_hn, mac_hn));
    out(pubChannel, sencryptCipher(secretCk, ck_hn)).


process
  ((!processMS) | processSN | processHN )
```

S7. LTE I || UMTS II–III || LTE V, conv(CK IK → K_{ASME}, MME)

```
(* Public channel between the MS and the SN *)
free pubChannel: channel.
```

```
(* Secure channel between the SN and the HN *)
free secureChannel: channel [private].

(* types *)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type cipherKey.
type integKey.
type asmeKey.
type enbKey.
type asEncKey.
type asIntKey.
type upEncKey.
type mac.
type msgMac.

(* constant message headers *)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const ASSMC: msgHdr.
const ASSMComplete: msgHdr.
const MSG: msgHdr.

(* Functions *)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun kdf_asme(cipherKey, integKey): asmeKey.
fun kdf_enb(asmeKey): enbKey.
fun kdf_as_enc(enbKey): asEncKey.
fun kdf_as_int(enbKey): asIntKey.
fun kdf_up_enc(enbKey): upEncKey.
fun finteg_as(bitstring, asIntKey): msgMac.

fun sencrypt(bitstring, cipherKey): bitstring.
reduc forall m: bitstring, k: cipherKey;
    sdecrypt(sencrypt(m, k), k) = m.

(* To test secrecy of the integrity key, *)
(* use them as session keys to encrypt a free private name *)
fun sencryptInteg(bitstring, integKey): bitstring.
reduc forall m: bitstring, k: integKey;
    sdecryptInteg(sencryptInteg(m, k), k) = m.

fun sencrypt_as(bitstring, asEncKey): bitstring.
reduc forall m: bitstring, k: asEncKey;
    sdecrypt_as(sencrypt_as(m, k), k) = m.

(*Type Converter*)
fun bool2bitstring(bool): bitstring [data, typeConverter].

reduc   encCapability() = true;
    encCapability() = false.

(* the table ident/keys
   The key table consists of pairs
   (ident, key) shared between MS and HN
   Table is not accessible by the attacker *)
table keys(ident, key).

free as_smcomplete_msg: bitstring.

free payload: bitstring [private].
event disableEnc.
(*When the attacker knows s, the event
    disableEnc has been executed.*)
query attacker(payload) ⇝ event(disableEnc).
query attacker(payload).

(* The standard secrecy queries of ProVerif only *)
(* deal with the secrecy of private free names*)
free secret: bitstring [private].
query attacker(secret).

not attacker(new ki).
```

```
(* Authentication queries *)
event begSN(ident, cipherKey, integKey).
event endSN(ident, cipherKey, integKey).
event begMS(ident, enbKey, bool).
event endMS(ident, enbKey, bool).

query x1: ident, x2: cipherKey, x3: integKey;
   event(endSN(x1, x2, x3)) ⇝  event(begSN(x1, x2, x3)).
query x1: ident, x2: enbKey, x3: bool;
   event(endMS(x1, x2, x3)) ⇝ event(begMS(x1, x2, x3)).

let processMS =
  (* The ident and pre-shared key of the mobile station *)
  new imsi_ms: ident;
  new ki: key;
  insert keys(imsi_ms, ki);
  (*MS non-deterministically choose
      the capability of encryption*)
  let cap_ms: bool = encCapability() in
  (*Send out cap_ms to SN *)
  out(pubChannel, (CAP, cap_ms));
  (*Send out permanent ID *)
  out(pubChannel, (ID, imsi_ms));
  (*Input challenge message from SN *)
  in(pubChannel, (=CHALLENGE, rand_ms: nonce, mac_ms: mac));
  if f1(ki, rand_ms) = mac_ms then
    (*Compute response and encryption key*)
    let res_ms: resp = f2(ki, rand_ms) in
    let ck_ms: cipherKey = f3(ki, rand_ms) in
    let ik_ms: integKey = f4(ki, rand_ms) in
    (*MS is authenticating itself to SN*)
    event begSN(imsi_ms, ck_ms, ik_ms);
    (*Send out response to SN *)
    out(pubChannel, (RES, res_ms));
    let kasme_ms = kdf_asme(ck_ms, ik_ms) in
    let kenb_ms: enbKey = kdf_enb(kasme_ms) in
        let kasenc_ms: asEncKey = kdf_as_enc(kenb_ms) in
        let kasint_ms: asIntKey = kdf_as_int(kenb_ms) in
        let kupenc_ms: upEncKey = kdf_up_enc(kenb_ms) in
    (*Receive GSM cipher mode command *)
    in(pubChannel, (=ASSMC, enableEnc_as_ms: bool,
        =finteg_as(bool2bitstring(enableEnc_as_ms),
            kasint_ms)));
        out(pubChannel, (ASSMComplete, as_smcomplete_msg,
            finteg_as(as_smcomplete_msg, kasint_ms)));
    event endMS(imsi_ms, kenb_ms, cap_ms);
    in(pubChannel, (=MSG, datamsg: bitstring,
        =finteg_as(datamsg, kasint_ms)));
        out(pubChannel, sencrypt(secret, ck_ms));
        out(pubChannel, sencryptInteg(secret, ik_ms));
    if enableEnc_as_ms = true then
        let msgcontent: bitstring = sdecrypt_as(datamsg,
            kasenc_ms) in 0.

let processSN =
  (*Receive MS's capability *)
  in(pubChannel, (=CAP, cap_sn: bool));
  (*Receive permanent ID *)
  in(pubChannel, (=ID, imsi_sn: ident));
  (*Send out authentication vector request *)
  out(secureChannel, (AV_REQ, imsi_sn));
  (*Receive authentication vector *)
  in(secureChannel, (=AV, =imsi_sn, rand_sn: nonce,
      xres_sn: resp, ck_sn: cipherKey, ik_sn: integKey,
      mac_sn: mac));
  (*Send authentication challenge to MS *)
  out(pubChannel, (CHALLENGE, rand_sn, mac_sn));
  (*Receive response *)
  in(pubChannel, (=RES, res_sn: resp));
  (*Check whether received response equal to XRES*)
  if res_sn = xres_sn then
    (*At this point, SN authenticated MS*)
    event endSN(imsi_sn, ck_sn, ik_sn);
    let kasme_sn = kdf_asme(ck_sn, ik_sn) in
    let kenb_sn: enbKey = kdf_enb(kasme_sn) in
    let kasenc_sn: asEncKey = kdf_as_enc(kenb_sn) in
        let kasint_sn: asIntKey = kdf_as_int(kenb_sn) in
        let kupenc_sn: upEncKey = kdf_up_enc(kenb_sn) in
        event begMS(imsi_sn, kenb_sn, cap_sn);
        out(pubChannel, (ASSMC, cap_sn,
            finteg_as(bool2bitstring(cap_sn), kasint_sn)));
        in(pubChannel, (=ASSMComplete, =as_smcomplete_msg,
            =finteg_as(as_smcomplete_msg, kasint_sn)));
        if cap_sn = false then
```

```
            event disableEnc;
            out(pubChannel, (MSG, payload,
                finteg_as(payload, kasint_sn)))
        else
            out(pubChannel, (MSG, sencrypt_as(payload,
                kasenc_sn), finteg_as(sencrypt_as(payload,
                kasenc_sn), kasint_sn))).

let processHN =
  (* Receive authentication vector request *)
  in(secureChannel, (=AV_REQ, imsi_hn: ident));
  (* Generate a fresh random number*)
  new rand_hn: nonce;
  (* Computes expected response and Kc*)
  get keys(=imsi_hn, ki_hn) in
  let mac_hn: mac = f1(ki_hn, rand_hn) in
  let xres_hn: resp = f2(ki_hn, rand_hn) in
  let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
  let ik_hn: integKey = f4(ki_hn, rand_hn) in
  (* Send out authentication vector *)
  out(secureChannel, (AV, imsi_hn, rand_hn,
      xres_hn, ck_hn, ik_hn, mac_hn)).

process
  ((!processMS) | processSN | processHN)
```

S7+. LTE I || UMTS II–III || LTE IV–V, conv(CK IK → K_{ASME}, MME)

```
(* Public channel between the MS and the SN*)
free pubChannel: channel.

(*types*)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type cipherKey.
type integKey.
type mac.
type msgMac.
type asmeKey.
type nasEncKey.
type nasIntKey.
type enbKey.
type asEncKey.
type asIntKey.
type upEncKey.

(* constant message headers*)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const NASSMC: msgHdr.
const NASSMComplete: msgHdr.
const ASSMC: msgHdr.
const ASSMComplete: msgHdr.
const MSG: msgHdr.

(* Functions*)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun kdf_asme(cipherKey, integKey): asmeKey.
fun kdf_nas_enc(asmeKey) : nasEncKey.
fun kdf_nas_int(asmeKey): nasIntKey.
fun finteg_nas(bitstring, nasIntKey): msgMac.
fun kdf_enb(asmeKey): enbKey.
fun kdf_as_enc(enbKey): asEncKey.
fun kdf_as_int(enbKey): asIntKey.
fun kdf_up_enc(enbKey): upEncKey.
fun finteg_as(bitstring, asIntKey): msgMac.

fun sencrypt_nas(bitstring, nasEncKey): bitstring.
reduc forall m: bitstring, k: nasEncKey;
    sdecrypt_nas(sencrypt_nas(m, k), k) = m.

fun sencrypt_as(bitstring, asEncKey): bitstring.
reduc forall m: bitstring, k: asEncKey;
    sdecrypt_as(sencrypt_as(m, k), k) = m.

(*Type Converter*)
fun bool2bitstring(bool): bitstring [data, typeConverter].

reduc   encCapability() = true;
        encCapability() = false.

(*the table ident/keys
   The key table consists of pairs
   (ident, key) shared between MS and HN
   Table is not accessible by the attacker*)
table keys(ident, key).

(*SMC command msg*)
free nas_smcomplete_msg: bitstring.
free as_smcomplete_msg: bitstring.

free payload: bitstring [private].
event disableEnc.
(*When the attacker knows s, the event
    disableEnc has been executed.*)
query attacker(payload) ~> event(disableEnc).
query attacker(payload).

free secret: bitstring [private].
query attacker(secret).
fun senc_int_nas(bitstring, nasIntKey): bitstring.
reduc forall m: bitstring, k: nasIntKey;
    sdec_in_nas(senc_int_nas(m, k), k) = m.
fun senc_int_as(bitstring, asIntKey): bitstring.
reduc forall m: bitstring, k: asIntKey;
    sdec_in_as(senc_int_as(m, k), k) = m.
fun senc_up(bitstring, upEncKey): bitstring.
reduc forall m: bitstring, k: upEncKey;
    sdec_up(senc_up(m, k), k) = m.


not attacker(new ki).

(*Authentication queries*)
event begSN(ident, cipherKey, integKey).
event endSN(ident, cipherKey, integKey).
event begMS(ident, asmeKey, bool).
event endMS(ident, asmeKey, bool).
event begMS_ENB(ident, enbKey, bool).
event endMS_ENB(ident, enbKey, bool).


query x1: ident, x2:cipherKey, x3: integKey;
    event(endSN(x1, x2, x3)) ~> event(begSN(x1, x2, x3)).
query x1: ident,  x2: asmeKey, x3: bool;
    event(endMS(x1, x2, x3)) ~> event(begMS(x1, x2, x3)).
query x1: ident, x2: enbKey, x3: bool;
    event(endMS_ENB(x1, x2, x3)) ~>
event(begMS_ENB(x1, x2, x3)).


(*AS SMC procedure in process MS*)
let pMSAS(kasme_ms: asmeKey, imsi_ms: ident, cap_ms: bool) =
    let kenb_ms: enbKey = kdf_enb(kasme_ms) in
    let kasenc_ms: asEncKey = kdf_as_enc(kenb_ms) in
    let kasint_ms: asIntKey = kdf_as_int(kenb_ms) in
    let kupenc_ms: upEncKey = kdf_up_enc(kenb_ms) in
    in(pubChannel, (=ASSMC, enableEnc_as_ms: bool,
        =finteg_as(bool2bitstring(enableEnc_as_ms), kasint_ms)));
    event endMS_ENB(imsi_ms, kenb_ms, cap_ms);
    out(pubChannel, (ASSMComplete, as_smcomplete_msg,
        finteg_as(as_smcomplete_msg, kasint_ms)));
    in(pubChannel, (=MSG, datamsg: bitstring,
        =finteg_as(datamsg, kasint_ms)));
    out(pubChannel, sencrypt_as(secret, kasenc_ms));
    out(pubChannel, senc_int_as(secret, kasint_ms));
    out(pubChannel, senc_up(secret, kupenc_ms));

    if enableEnc_as_ms = true then
        let msgcontent: bitstring =
            sdecrypt_as(datamsg, kasenc_ms) in 0.

(*process respresenting MS*)
let processMS =
    (*The identity of the MS*)
    new imsi_ms: ident;
```

```
    (* Pre-shared key*)
    new ki: key;
    (* Insert id/pre-shared key pair into the private table*)
    insert keys(imsi_ms, ki);
    (*MS non-deterministically choose
        the capability of encryption*)
    let cap_ms: bool = encCapability() in
    out(pubChannel, (CAP, cap_ms));
    out(pubChannel, (ID, imsi_ms));
    in(pubChannel, (=CHALLENGE, rand_ms: nonce,
        =f1(ki, rand_ms)));
    (* Compute response and encryption key*)
  let res_ms: resp = f2(ki, rand_ms) in
  let ck_ms: cipherKey = f3(ki, rand_ms) in
  let ik_ms: integKey = f4(ki, rand_ms) in
  (*MS is authenticating itself to SN*)
    event begSN(imsi_ms, ck_ms, ik_ms);
  (* Send out response to SN *)
  out(pubChannel, (RES, res_ms));
  let kasme_ms:asmeKey = kdf_asme(ck_ms, ik_ms) in
    (*NAS SMC procedure*)
    let knasenc_ms: nasEncKey = kdf_nas_enc(kasme_ms) in
    let knasint_ms: nasIntKey = kdf_nas_int(kasme_ms) in
    in(pubChannel, (=NASSMC, enableEnc_nas_ms: bool,
        =cap_ms, =finteg_nas((enableEnc_nas_ms, cap_ms),
            knasint_ms)));
    event endMS(imsi_ms, kasme_ms, cap_ms);
    (*NAS key secrecy*)
    out(pubChannel, sencrypt_nas(secret, knasenc_ms));
    out(pubChannel, senc_int_nas(secret, knasint_ms));
    if enableEnc_nas_ms = false then
        out(pubChannel, (NASSMComplete, nas_smcomplete_msg,
            finteg_nas(nas_smcomplete_msg, knasint_ms)));
        pMSAS(kasme_ms, imsi_ms, cap_ms)
    else
        out(pubChannel, (NASSMComplete,
          sencrypt_nas(nas_smcomplete_msg, knasenc_ms),
            finteg_nas(sencrypt_nas(nas_smcomplete_msg,
            knasenc_ms), knasint_ms)));
        pMSAS(kasme_ms, imsi_ms, cap_ms).



(* process representing e-nodeB*)
let pENB(kasme_enb: asmeKey, imsi_enb: ident, cap_enb: bool) =
    let kenb_enb: enbKey = kdf_enb(kasme_enb) in
    let kasenc_enb: asEncKey = kdf_as_enc(kenb_enb) in
    let kasint_enb: asIntKey = kdf_as_int(kenb_enb) in
    let kupenc_enb: upEncKey = kdf_up_enc(kenb_enb) in
    event begMS_ENB(imsi_enb, kenb_enb, cap_enb);
    out(pubChannel, (ASSMC, cap_enb,
      finteg_as(bool2bitstring(cap_enb), kasint_enb)));
    in(pubChannel, (=ASSMComplete, =as_smcomplete_msg,
        =finteg_as(as_smcomplete_msg, kasint_enb)));
    if cap_enb = false then
      event disableEnc;
      out(pubChannel, (MSG, payload,
          finteg_as(payload, kasint_enb)))
    else
      out(pubChannel, (MSG, sencrypt_as(payload,
          kasenc_enb), finteg_as(sencrypt_as(payload,
          kasenc_enb), kasint_enb))).

(* process representing MME*)
let processMME =
    in(pubChannel, (=CAP, cap_sn: bool));
    in(pubChannel, (=ID, imsi_sn: ident));

  new rand_sn: nonce;
  (* Computes expected response and Kc*)
  get keys(=imsi_sn, ki_sn) in
  let mac_sn: mac = f1(ki_sn, rand_sn) in
  let xres_sn: resp = f2(ki_sn, rand_sn) in
  let ck_sn: cipherKey = f3(ki_sn, rand_sn) in
  let ik_sn: integKey = f4(ki_sn, rand_sn) in

  out(pubChannel, (CHALLENGE, rand_sn, mac_sn));
  (* Receive response *)
  in(pubChannel, (=RES, =xres_sn));
    event endSN(imsi_sn, ck_sn, ik_sn);
    let kasme_sn: asmeKey = kdf_asme(ck_sn, ik_sn) in
    (*NAS SMC procedure*)
    let knasenc_sn: nasEncKey = kdf_nas_enc(kasme_sn) in
    let knasint_sn: nasIntKey = kdf_nas_int(kasme_sn) in
    event begMS(imsi_sn, kasme_sn, cap_sn);
    out(pubChannel, (NASSMC, cap_sn, cap_sn,
        finteg_nas((cap_sn, cap_sn), knasint_sn)));
    in(pubChannel, (=NASSMComplete, msg_nas: bitstring,
        =finteg_nas(msg_nas, knasint_sn)));
    if cap_sn = true then
        if sdecrypt_nas(msg_nas, knasenc_sn)
            = nas_smcomplete_msg then
                pENB(kasme_sn, imsi_sn, cap_sn)
        else 0
    else
        if msg_nas = nas_smcomplete_msg then
            pENB(kasme_sn, imsi_sn, cap_sn)
        else 0.

process
    ((!processMS) | processMME)
```

S8. LTE I' ∥ UMTS II–III ∥ LTE IV–V, conv(CK IK nonces → K_{ASME}, MME)

```
(* Public channel between the MS and the SN*)
free pubChannel: channel.
(* Secure channel between the SN and the HN*)
free secureChannel: channel [private].
free sChannelSnBts: channel [private].

(* types*)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type cipherKey.
type integKey.
type mac.
type msgMac.
type asmeKey.
type nasEncKey.
type nasIntKey.
type enbKey.
type asEncKey.
type asIntKey.
type upEncKey.

(* constant message headers*)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const NASSMC: msgHdr.
const NASSMComplete: msgHdr.
const ASSMC: msgHdr.
const ASSMComplete: msgHdr.
const MSG: msgHdr.
const NONCE_TAU: msgHdr.

(* Functions*)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun kdf_asme(cipherKey, integKey, nonce, nonce): asmeKey.
fun kdf_nas_enc(asmeKey) : nasEncKey.
fun kdf_nas_int(asmeKey): nasIntKey.
fun finteg_nas(bitstring, nasIntKey): msgMac.
fun kdf_enb(asmeKey): enbKey.
fun kdf_as_enc(enbKey): asEncKey.
fun kdf_as_int(enbKey): asIntKey.
fun kdf_up_enc(enbKey): upEncKey.
fun finteg_as(bitstring, asIntKey): msgMac.

fun sencrypt_nas(bitstring, nasEncKey): bitstring.
reduc forall m: bitstring, k: nasEncKey;
    sdecrypt_nas(sencrypt_nas(m, k), k) = m.

fun sencrypt_as(bitstring, asEncKey): bitstring.
reduc forall m: bitstring, k: asEncKey;
    sdecrypt_as(sencrypt_as(m, k), k) = m.

(* Type Converter*)
```

```
fun bool2bitstring(bool): bitstring [data, typeConverter].

reduc     encCapability() = true;
        encCapability() = false.

(*the table ident/keys
   The key table consists of pairs
   (ident, key) shared between MS and HN
   Table is not accessible by the attacker*)
table keys(ident, key).

(*SMC command msg*)
free nas_smcomplete_msg: bitstring.
free as_smcomplete_msg: bitstring.

free payload: bitstring [private].
event disableEnc.
(*When the attacker knows s, the event
    disableEnc has been executed.*)
query attacker(payload) ~> event(disableEnc).
query attacker(payload).

free secret: bitstring [private].
query attacker(secret).
fun senc_int_nas(bitstring, nasIntKey): bitstring.
reduc forall m: bitstring, k: nasIntKey;
    sdec_in_nas(senc_int_nas(m, k), k) = m.
fun senc_int_as(bitstring, asIntKey): bitstring.
reduc forall m: bitstring, k: asIntKey;
    sdec_in_as(senc_int_as(m, k), k) = m.
fun senc_up(bitstring, upEncKey): bitstring.
reduc forall m: bitstring, k: upEncKey;
    sdec_up(senc_up(m, k), k) = m.

not attacker(new ki).

(*Authentication queries*)
event begSN(ident, cipherKey, integKey).
event endSN(ident, cipherKey, integKey).
event begMS(ident, cipherKey, integKey, bool).
event endMS(ident, cipherKey, integKey, bool).
event begMS_ENB(ident, enbKey, bool).
event endMS_ENB(ident, enbKey, bool).

query x1: ident, x2: cipherKey, x3: integKey;
    event(endSN(x1, x2, x3)) ~> event(begSN(x1, x2, x3)).
query x1: ident, x2: cipherKey, x3: integKey, x4: bool;
    event(endMS(x1, x2, x3, x4)) ~>
event(begMS(x1, x2, x3, x4)).
query x1: ident, x2: enbKey, x3: bool;
    event(endMS_ENB(x1, x2, x3)) ~>
event(begMS_ENB(x1, x2, x3)).

(*AS SMC procedure in process MS*)
let pMSAS(kasme_ms: asmeKey, imsi_ms: ident, cap_ms: bool) =
    let kenb_ms: enbKey = kdf_enb(kasme_ms) in
    let kasenc_ms: asEncKey = kdf_as_enc(kenb_ms) in
    let kasint_ms: asIntKey = kdf_as_int(kenb_ms) in
    let kupenc_ms: upEncKey = kdf_up_enc(kenb_ms) in
    in(pubChannel, (=ASSMC, enableEnc_as_ms: bool,
        =finteg_as(bool2bitstring(enableEnc_as_ms),
            kasint_ms)));
    out(pubChannel, (ASSMComplete, as_smcomplete_msg,
        finteg_as(as_smcomplete_msg, kasint_ms)));
    event endMS_ENB(imsi_ms, kenb_ms, cap_ms);
    in(pubChannel, (=MSG, datamsg: bitstring,
        =finteg_as(datamsg, kasint_ms)));
    out(pubChannel, sencrypt_as(secret, kasenc_ms));
    out(pubChannel, senc_int_as(secret, kasint_ms));
    out(pubChannel, senc_up(secret, kupenc_ms));
    if enableEnc_as_ms = true then
        let msgcontent: bitstring =
            sdecrypt_as(datamsg, kasenc_ms) in 0.

(*process respresenting MS*)
let processMS =
    (*The identity of the MS*)
    new imsi_ms: ident;
    (*Pre-shared key*)
    new ki: key;
    (*Insert id/pre-shared key pair into the private table*)
    insert keys(imsi_ms, ki);
    (*MS non-deterministically choose the capability of encryption*)
    let cap_ms: bool = encCapability() in
    out(pubChannel, (CAP, cap_ms));
    out(pubChannel, (ID, imsi_ms));
    new nonce_ms: nonce;
    out(pubChannel, (NONCE_TAU, nonce_ms));
    in(pubChannel, (=CHALLENGE, rand_ms: nonce, =f1(ki, rand_ms)));
    let res_ms: resp = f2(ki, rand_ms) in
    let ck_ms: cipherKey = f3(ki, rand_ms) in
    let ik_ms: integKey = f4(ki, rand_ms) in
    event begSN(imsi_ms, ck_ms, ik_ms);
    out(pubChannel, (RES, res_ms));
    (*NAS SMC procedure*)
    in(pubChannel, (=NASSMC, enableEnc_nas_ms: bool, =cap_ms,
        =nonce_ms, nonce_mme_ms: nonce, nas_mac: msgMac));
    let kasme_ms: asmeKey = kdf_asme(ck_ms, ik_ms,
        nonce_ms, nonce_mme_ms) in
    let knasenc_ms: nasEncKey = kdf_nas_enc(kasme_ms) in
    let knasint_ms: nasIntKey = kdf_nas_int(kasme_ms) in
    if(nas_mac = finteg_nas((enableEnc_nas_ms, cap_ms,
        nonce_ms, nonce_mme_ms), knasint_ms)) then
        event endMS(imsi_ms, ck_ms, ik_ms, cap_ms);
        (*NAS key secrecy*)
        out(pubChannel, sencrypt_nas(secret, knasenc_ms));
        out(pubChannel, senc_int_nas(secret, knasint_ms));
        if enableEnc_nas_ms = false then
            out(pubChannel, (NASSMComplete, nas_smcomplete_msg,
                finteg_nas(nas_smcomplete_msg, knasint_ms)));
            pMSAS(kasme_ms, imsi_ms, cap_ms)
        else
            out(pubChannel, (NASSMComplete,
                sencrypt_nas(nas_smcomplete_msg, knasenc_ms),
                finteg_nas(sencrypt_nas(nas_smcomplete_msg,
                knasenc_ms), knasint_ms)));
            pMSAS(kasme_ms, imsi_ms, cap_ms).

(*process representing e-nodeB*)
let processENB =
    in(sChannelSnBts, (kasme_enb: asmeKey, imsi_enb: ident,
        cap_enb: bool));
    let kenb_enb: enbKey = kdf_enb(kasme_enb) in
    let kasenc_enb: asEncKey = kdf_as_enc(kenb_enb) in
    let kasint_enb: asIntKey = kdf_as_int(kenb_enb) in
    let kupenc_enb: upEncKey = kdf_up_enc(kenb_enb) in
    event begMS_ENB(imsi_enb, kenb_enb, cap_enb);
    out(pubChannel, (ASSMC, cap_enb,
        finteg_as(bool2bitstring(cap_enb), kasint_enb)));
    in(pubChannel, (=ASSMComplete, =as_smcomplete_msg,
        =finteg_as(as_smcomplete_msg, kasint_enb)));
    if cap_enb = false then
        event disableEnc;
        out(pubChannel, (MSG, payload,
            finteg_as(payload, kasint_enb)))
    else
        out(pubChannel, (MSG, sencrypt_as(payload, kasenc_enb),
            finteg_as(sencrypt_as(payload, kasenc_enb),
            kasint_enb))).

(*process representing MME*)
let processMME =
    in(pubChannel, (=CAP, cap_sn: bool));
    in(pubChannel, (=ID, imsi_sn: ident));
    in(pubChannel, (=NONCE_TAU, nonce_ms_sn:nonce));
    out(secureChannel, (AV_REQ, imsi_sn));
    in(secureChannel, (=AV, =imsi_sn, rand_sn: nonce,
        xres_sn: resp, ck_sn: cipherKey,
        ik_sn: integKey, mac_sn: mac));
    out(pubChannel, (CHALLENGE, rand_sn, mac_sn));
    in(pubChannel, (=RES, =xres_sn));
    event endSN(imsi_sn, ck_sn, ik_sn);
    new nonce_mme: nonce;
    (*NAS SMC procedure*)
    let kasme_sn: asmeKey = kdf_asme(ck_sn, ik_sn,
        nonce_ms_sn, nonce_mme) in
    let knasenc_sn: nasEncKey = kdf_nas_enc(kasme_sn) in
    let knasint_sn: nasIntKey = kdf_nas_int(kasme_sn) in
    event begMS(imsi_sn, ck_sn, ik_sn, cap_sn);
    out(pubChannel, (NASSMC, cap_sn, cap_sn, nonce_ms_sn,
        nonce_mme, finteg_nas((cap_sn, cap_sn,
        nonce_ms_sn, nonce_mme), knasint_sn)));
    in(pubChannel, (=NASSMComplete, msg_nas:bitstring,
        =finteg_nas(msg_nas, knasint_sn)));
    if cap_sn = true then
        if sdecrypt_nas(msg_nas, knasenc_sn)
            = nas_smcomplete_msg then
                out(sChannelSnBts, (kasme_sn,
```

```
                    imsi_sn, cap_sn))
        else 0
    else
        if cap_sn = false then
            if msg_nas = nas_smcomplete_msg then
                out(sChannelSnBts, (kasme_sn,
                    imsi_sn, cap_sn))
            else 0
        else 0.

(*process representing HN*)
let processHN =
  (*Receive authentication vector request *)
  in(secureChannel, (=AV_REQ, imsi_hn: ident));
  (*Generate a fresh random number*)
  new rand_hn: nonce;
  (*Computes expected response and Kc*)
  get keys(=imsi_hn, ki_hn) in
  let mac_hn: mac = f1(ki_hn, rand_hn) in
  let xres_hn: resp = f2(ki_hn, rand_hn) in
  let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
  let ik_hn: integKey = f4(ki_hn, rand_hn) in
  (*Send out authentication vector *)
  out(secureChannel, (AV, imsi_hn, rand_hn,
     xres_hn, ck_hn, ik_hn, mac_hn)).

process
    ((!processMS) | (processMME) | (processENB) | (processHN))
```

S9. GSM I || LTE II–III || GSM IV, conv(CK IK → Kc, MME), AV = 4G AV + CK + IK

```
(*Public channel between the MS and the SN*)
free pubChannel: channel.
(*Secure channel between the SN and the HN*)
free secureChannel: channel [private].

(*types*)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type cipherKey.
type integKey.
type gsmKey.
type mac.
type msgMac.
type asmeKey.
type nasEncKey.
type nasIntKey.

(*constant message headers*)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const NASSMC: msgHdr.
const NASSMComplete: msgHdr.
const ASSMC: msgHdr.
const ASSMComplete: msgHdr.
const MSG: msgHdr.
const CMComplete: msgHdr.

(*Functions*)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun f9(bitstring, integKey): bitstring.
fun kdf_asme(cipherKey, integKey, ident): asmeKey.
fun kdf_nas_enc(asmeKey) : nasEncKey.
fun kdf_nas_int(asmeKey): nasIntKey.
fun finteg_nas(bitstring, nasIntKey): msgMac.
fun c3(cipherKey, integKey): gsmKey.

fun sencrypt_nas(bitstring, nasEncKey): bitstring.
reduc forall m: bitstring, k: nasEncKey;
    sdecrypt_nas(sencrypt_nas(m, k), k) = m.

fun sencrypt_as(bitstring, gsmKey): bitstring.
reduc forall m: bitstring, k: gsmKey;
    sdecrypt_as(sencrypt_as(m, k), k) = m.

(*Type Converter*)
fun bool2bitstring(bool): bitstring [data, typeConverter].

reduc     encCapability() = true;
        encCapability() = false.

(*the table ident/keys
   The key table consists of pairs
   (ident, key) shared between MS and HN
   Table is not accessible by the attacker*)
table keys(ident, key).

(*SMC command msg*)
free nas_smcomplete_msg: bitstring.
free as_smcomplete_msg: bitstring.

free payload: bitstring [private].
event disableEnc.
(*When the attacker knows s, the event
    disableEnc has been executed.*)
query attacker(payload) ~> event(disableEnc).
query attacker(payload).

free secret: bitstring [private].
query attacker(secret).
fun senc_int_nas(bitstring, nasIntKey): bitstring.
reduc forall m: bitstring, k: nasIntKey;
    sdec_in_nas(senc_int_nas(m, k), k) = m.
fun senc_int_as(bitstring, integKey): bitstring.
reduc forall m: bitstring, k: integKey;
    sdec_in_as(senc_int_as(m, k), k) = m.


not attacker(new ki).

(*Authentication queries*)
event begSN(ident, cipherKey, integKey).
event endSN(ident, cipherKey, integKey).
event begMS_AS(ident, gsmKey, bool).
event endMS_AS(ident, gsmKey, bool).

query x1: ident, x2: cipherKey, x3: integKey;
    event(endSN(x1, x2, x3)) ~>  event(begSN(x1, x2, x3)).
query x1: ident, x2: gsmKey, x3: bool;
    event(endMS_AS(x1, x2, x3)) ~>
event(begMS_AS(x1, x2, x3)).


(*process respresenting MS*)
let processMS =
    (*The identity of the MS*)
    new imsi_ms: ident;
    (*Pre-shared key*)
    new ki: key;
    (*Insert id/pre-shared key pair into the private table*)
    insert keys(imsi_ms, ki);
    (*MS non-deterministically choose
        the capability of encryption*)
    let cap_ms: bool = encCapability() in
    out(pubChannel, (CAP, cap_ms));
    out(pubChannel, (ID, imsi_ms));
    in(pubChannel, (=CHALLENGE, rand_ms: nonce,
        =f1(ki, rand_ms), snid_ms: ident));
    let res_ms: resp = f2(ki, rand_ms) in
    let ck_ms: cipherKey = f3(ki, rand_ms) in
    let ik_ms: integKey = f4(ki, rand_ms) in
    let kasme_ms: asmeKey = kdf_asme(ck_ms, ik_ms, snid_ms) in
    event begSN(imsi_ms, ck_ms, ik_ms);
    out(pubChannel, (RES, res_ms));
    let kc_ms:gsmKey = c3(ck_ms, ik_ms) in
    in(pubChannel, (=ASSMC, enableEnc_as_ms: bool));
    event endMS_AS(imsi_ms, kc_ms, cap_ms);
    out(pubChannel, CMComplete);
    in(pubChannel, (=MSG, datamsg: bitstring));
    out(pubChannel, sencrypt_as(secret, kc_ms));
    if enableEnc_as_ms = true then
        let msgcontent: bitstring =
            sdecrypt_as(datamsg, kc_ms) in 0.
```

```
(*process representing MME*)
let processSN =
    in(pubChannel, (=CAP, cap_sn: bool));
    in(pubChannel, (=ID, imsi_sn: ident));
    new snid_sn: ident;
    out(secureChannel, (AV_REQ, imsi_sn, snid_sn));
    in(secureChannel, (=AV, =imsi_sn, snid_hn_sn: ident,
        rand_sn: nonce, xres_sn: resp, mac_sn: mac,
        kasme_sn: asmeKey,
    ck_sn: cipherKey, ik_sn: integKey));
    out(pubChannel, (CHALLENGE, rand_sn, mac_sn, snid_sn));
    in(pubChannel, (=RES, =xres_sn));
    event endSN(imsi_sn, ck_sn, ik_sn);
    let kc_sn: gsmKey = c3(ck_sn, ik_sn) in
    event begMS_AS(imsi_sn, kc_sn, cap_sn);
    out(pubChannel, (ASSMC, cap_sn));
    in(pubChannel, =CMComplete);
    if cap_sn = false then
        event disableEnc;
        out(pubChannel, (MSG, payload))
    else
        out(pubChannel, (MSG, sencrypt_as(payload, kc_sn))).

(*process representing HN*)
let processHN =
    in(secureChannel, (=AV_REQ, imsi_hn: ident,
        snid_hn: ident));
    (*Generate athenication vectors*)
    new rand_hn: nonce;
    get keys(=imsi_hn, ki_hn) in
    let mac_hn: mac = f1(ki_hn, rand_hn) in
    let xres_hn: resp = f2(ki_hn, rand_hn) in
    let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
    let ik_hn: integKey = f4(ki_hn, rand_hn) in
    let kasme_hn: asmeKey = kdf_asme(ck_hn, ik_hn, snid_hn) in
    out(secureChannel, (AV, imsi_hn, snid_hn, rand_hn,
        xres_hn, mac_hn, kasme_hn, ck_hn, ik_hn)).


process
    ((!processMS) | processSN | processHN)
```

S9+. GSM I || LTE II–IV || GSM IV, conv(CK IK → Kc, MME), AV = 4G AV + CK + IK

```
(*Public channel between the MS and the SN*)
free pubChannel: channel.
(*Secure channel between the SN and the HN*)
free secureChannel: channel [private].
free sChannelSnBts: channel [private].

(*types*)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type cipherKey.
type integKey.
type gsmKey.
type mac.
type msgMac.
type asmeKey.
type nasEncKey.
type nasIntKey.

(*constant message headers*)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const NASSMC: msgHdr.
const NASSMComplete: msgHdr.
const ASSMC: msgHdr.
const ASSMComplete: msgHdr.
const MSG: msgHdr.
const CMComplete: msgHdr.

(*Functions*)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun f9(bitstring, integKey): bitstring.
fun kdf_asme(cipherKey, integKey, ident): asmeKey.
fun kdf_nas_enc(asmeKey) : nasEncKey.
fun kdf_nas_int(asmeKey): nasIntKey.
fun finteg_nas(bitstring, nasIntKey): msgMac.
fun c3(cipherKey, integKey): gsmKey.

fun sencrypt_nas(bitstring, nasEncKey): bitstring.
reduc forall m: bitstring, k: nasEncKey;
    sdecrypt_nas(sencrypt_nas(m, k), k) = m.

fun sencrypt_as(bitstring, gsmKey): bitstring.
reduc forall m: bitstring, k: gsmKey;
    sdecrypt_as(sencrypt_as(m, k), k) = m.

(*Type Converter*)
fun bool2bitstring(bool): bitstring [data, typeConverter].

reduc      encCapability() = true;
        encCapability() = false.

(*the table ident/keys
   The key table consists of pairs
   (ident, key) shared between MS and HN
   Table is not accessible by the attacker*)
table keys(ident, key).

(*SMC command msg*)
free nas_smcomplete_msg: bitstring.
free as_smcomplete_msg: bitstring.

free payload: bitstring [private].
event disableEnc.
(*When the attacker knows s, the event
    disableEnc has been executed.*)
query attacker(payload) ⇝ event(disableEnc).
query attacker(payload).

free secret: bitstring [private].
query attacker(secret).
fun senc_int_nas(bitstring, nasIntKey): bitstring.
reduc forall m: bitstring, k: nasIntKey;
    sdec_in_nas(senc_int_nas(m, k), k) = m.
fun senc_int_as(bitstring, integKey): bitstring.
reduc forall m: bitstring, k: integKey;
    sdec_in_as(senc_int_as(m, k), k) = m.


not attacker(new ki).

(*Authentication queries*)
event begSN(ident, ident, asmeKey).
event endSN(ident, ident, asmeKey).
event begMS(ident, ident, asmeKey, bool).
event endMS(ident, ident, asmeKey, bool).
event begMS_AS(ident, gsmKey, bool).
event endMS_AS(ident, gsmKey, bool).

query x1: ident, x2: ident, x3: asmeKey;
    event(endSN(x1, x2, x3)) ⇝  event(begSN(x1, x2, x3)).
query x1: ident, x2: ident, x3: asmeKey, x4: bool;
    event(endMS(x1, x2, x3, x4)) ⇝
event(begMS(x1, x2, x3, x4)).
query x1: ident, x2: gsmKey, x3: bool;
    event(endMS_AS(x1, x2, x3)) ⇝
event(begMS_AS(x1, x2, x3)).


(*AS SMC procedure in process MS*)
let pMSAS(kc_ms:gsmKey, imsi_ms: ident, cap_ms: bool) =
    in(pubChannel, (=ASSMC, enableEnc_as_ms: bool));
    event endMS_AS(imsi_ms, kc_ms, cap_ms);
    out(pubChannel, CMComplete);
    in(pubChannel, (=MSG, datamsg: bitstring));
    out(pubChannel, sencrypt_as(secret, kc_ms));
    if enableEnc_as_ms = true then
        let msgcontent: bitstring = sdecrypt_as(datamsg, kc_ms)
        in 0.

(*process respresenting MS*)
let processMS =
```

```
    (*The identity of the MS*)
    new imsi_ms: ident;
    (*Pre-shared key*)
    new ki: key;
    (*Insert id/pre-shared key pair into the private table*)
    insert keys(imsi_ms, ki);
    (*MS non-deterministically choose
        the capability of encryption*)
    let cap_ms: bool = encCapability() in
    out(pubChannel, (CAP, cap_ms));
    out(pubChannel, (ID, imsi_ms));
    in(pubChannel, (=CHALLENGE, rand_ms: nonce,
        =f1(ki, rand_ms), snid_ms: ident));
    let res_ms: resp = f2(ki, rand_ms) in
    let ck_ms: cipherKey = f3(ki, rand_ms) in
    let ik_ms: integKey = f4(ki, rand_ms) in
    let kasme_ms: asmeKey = kdf_asme(ck_ms, ik_ms, snid_ms) in
    event begSN(imsi_ms, snid_ms, kasme_ms);
    out(pubChannel, (RES, res_ms));
    (*NAS SMC procedure*)
    let knasenc_ms: nasEncKey = kdf_nas_enc(kasme_ms) in
    let knasint_ms: nasIntKey = kdf_nas_int(kasme_ms) in
    in(pubChannel, (=NASSMC, enableEnc_nas_ms: bool, =cap_ms,
        =finteg_nas((enableEnc_nas_ms, cap_ms), knasint_ms)));
    event endMS(imsi_ms, snid_ms, kasme_ms, cap_ms);
    (*NAS key secrecy*)
    out(pubChannel, sencrypt_nas(secret, knasenc_ms));
    out(pubChannel, senc_int_nas(secret, knasint_ms));
    let kc_ms:gsmKey = c3(ck_ms, ik_ms) in
    if enableEnc_nas_ms = false then
        out(pubChannel, (NASSMComplete, nas_smcomplete_msg,
            finteg_nas(nas_smcomplete_msg, knasint_ms)));
        pMSAS(kc_ms, imsi_ms, cap_ms)
    else
        out(pubChannel, (NASSMComplete,
            sencrypt_nas(nas_smcomplete_msg, knasenc_ms),
            finteg_nas(sencrypt_nas(nas_smcomplete_msg,
            knasenc_ms), knasint_ms)));
        pMSAS(kc_ms, imsi_ms, cap_ms).

(*process representing e-nodeB*)
let processBS =
    in(sChannelSnBts, (kc_bs: gsmKey,
        imsi_bs: ident, cap_bs: bool));
    event begMS_AS(imsi_bs, kc_bs, cap_bs);
    out(pubChannel, (ASSMC, cap_bs));
    in(pubChannel, =CMComplete);
    if cap_bs = false then
        event disableEnc;
        out(pubChannel, (MSG, payload))
    else
        out(pubChannel, (MSG, sencrypt_as(payload, kc_bs))).

(*process representing MME*)
let processSN =
    in(pubChannel, (=CAP, cap_sn: bool));
    in(pubChannel, (=ID, imsi_sn: ident));
    new snid_sn: ident;
    out(secureChannel, (AV_REQ, imsi_sn, snid_sn));
    in(secureChannel, (=AV, imsi_hn_sn: ident,
        snid_hn_sn: ident, rand_sn: nonce,
        xres_sn: resp, mac_sn: mac, kasme_sn: asmeKey,
        ck_sn: cipherKey, ik_sn: integKey));
    out(pubChannel, (CHALLENGE, rand_sn, mac_sn, snid_sn));
    in(pubChannel, (=RES, =xres_sn));
    event begMS(imsi_hn_sn, snid_hn_sn, kasme_sn, cap_sn);
    (*NAS SMC procedure*)
    let knasenc_sn: nasEncKey = kdf_nas_enc(kasme_sn) in
    let knasint_sn: nasIntKey = kdf_nas_int(kasme_sn) in
    out(pubChannel, (NASSMC, cap_sn, cap_sn,
        finteg_nas((cap_sn, cap_sn), knasint_sn)));
    in(pubChannel, (=NASSMComplete, msg_nas:bitstring,
        =finteg_nas(msg_nas, knasint_sn)));
    let kc_sn: gsmKey = c3(ck_sn, ik_sn) in
    if cap_sn = true then
        if sdecrypt_nas(msg_nas, knasenc_sn) =
            nas_smcomplete_msg then
            event endSN(imsi_hn_sn, snid_hn_sn, kasme_sn);
            out(sChannelSnBts, (kc_sn, imsi_hn_sn, cap_sn))
        else 0
    else
        if cap_sn = false then
            if msg_nas = nas_smcomplete_msg then
                event endSN(imsi_hn_sn, snid_hn_sn, kasme_sn);
                out(sChannelSnBts, (kc_sn, imsi_hn_sn, cap_sn))
            else 0
        else 0.

(*process representing HN*)
let processHN =
    in(secureChannel, (=AV_REQ, imsi_hn: ident, snid_hn: ident));
    (*Generate athenication vectors*)
    new rand_hn: nonce;
    get keys(=imsi_hn, ki_hn) in
    let mac_hn: mac = f1(ki_hn, rand_hn) in
    let xres_hn: resp = f2(ki_hn, rand_hn) in
    let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
    let ik_hn: integKey = f4(ki_hn, rand_hn) in
    let kasme_hn: asmeKey = kdf_asme(ck_hn, ik_hn, snid_hn) in
    out(secureChannel, (AV, imsi_hn, snid_hn, rand_hn,
        xres_hn, mac_hn, kasme_hn, ck_hn, ik_hn)).


process
    ((!processMS) | processSN | processBS | processHN)
```

S10. UMTS I || LTE II–III || UMTS IV, AV = 4G AV + CK + IK

```
(*Public channel between the MS and the SN*)
free pubChannel: channel.
(*Secure channel between the SN and the HN*)
free secureChannel: channel [private].

(*types*)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type cipherKey.
type integKey.
type mac.
type msgMac.
type asmeKey.

(*constant message headers*)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const ASSMC: msgHdr.
const ASSMComplete: msgHdr.
const MSG: msgHdr.

(*Functions*)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun f9(bitstring, integKey): bitstring.
fun kdf_asme(cipherKey, integKey, ident): asmeKey.

fun sencrypt_as(bitstring, cipherKey): bitstring.
reduc forall m: bitstring, k: cipherKey;
    sdecrypt_as(sencrypt_as(m, k), k) = m.

(*Type Converter*)
fun bool2bitstring(bool): bitstring [data, typeConverter].

reduc      encCapability() = true;
        encCapability() = false.

(*the table ident/keys
   The key table consists of pairs
   (ident, key) shared between MS and HN
   Table is not accessible by the attacker*)
table keys(ident, key).

(*SMC command msg*)
free as_smcomplete_msg: bitstring.

free payload: bitstring [private].
event disableEnc.
(*When the attacker knows s, *)
```

```
(* the event disableEnc has been executed.*)
query attacker(payload) ~> event(disableEnc).
query attacker(payload).

free secret: bitstring [private].
query attacker(secret).
fun senc_int_as(bitstring, integKey): bitstring.
reduc forall m: bitstring, k: integKey;
sdec_in_as(senc_int_as(m, k), k) = m.


not attacker(new ki).

(*Authentication queries*)
event begSN(ident, cipherKey, integKey).
event endSN(ident, cipherKey, integKey).
event begMS(ident, cipherKey, integKey, bool).
event endMS(ident, cipherKey, integKey, bool).


query x1: ident, x2: cipherKey, x3: integKey;
    event(endSN(x1, x2, x3)) ~> event(begSN(x1, x2, x3)).

query x1: ident, x2: cipherKey, x3: integKey, x4: bool;
    event(endMS(x1, x2, x3, x4)) ~>
event(begMS(x1, x2, x3, x4)).




(*process respresenting MS*)
let processMS =
    (*The identity of the MS*)
    new imsi_ms: ident;
    (*Pre-shared key*)
    new ki: key;
    (*Insert id/pre-shared key pair into the private table*)
    insert keys(imsi_ms, ki);
    (*MS non-deterministically choose
        the capability of encryption*)
    let cap_ms: bool = encCapability() in
    out(pubChannel, (CAP, cap_ms));
    out(pubChannel, (ID, imsi_ms));
    in(pubChannel, (=CHALLENGE, rand_ms: nonce,
        =f1(ki, rand_ms), snid_ms: ident));
    let res_ms: resp = f2(ki, rand_ms) in
    let ck_ms: cipherKey = f3(ki, rand_ms) in
    let ik_ms: integKey = f4(ki, rand_ms) in
    let kasme_ms: asmeKey = kdf_asme(ck_ms, ik_ms, snid_ms) in
    event begSN(imsi_ms, ck_ms, ik_ms);
    out(pubChannel, (RES, res_ms));
    in(pubChannel, (=ASSMC, =cap_ms, enableEnc_as_ms: bool,
        fresh_ms:nonce, =f9((cap_ms, enableEnc_as_ms,
        fresh_ms), ik_ms)));
    out(pubChannel, (ASSMComplete, as_smcomplete_msg,
        f9(as_smcomplete_msg, ik_ms)));
    event endMS(imsi_ms, ck_ms, ik_ms, cap_ms);
    in(pubChannel, (=MSG, datamsg: bitstring,
        =f9(datamsg, ik_ms)));
    out(pubChannel, sencrypt_as(secret, ck_ms));
    out(pubChannel, senc_int_as(secret, ik_ms));
    if enableEnc_as_ms = true then
        let msgcontent: bitstring = sdecrypt_as(datamsg, ck_ms)
        in 0.


(*process representing MME*)
let processSN =
    in(pubChannel, (=CAP, cap_sn: bool));
    in(pubChannel, (=ID, imsi_sn: ident));
    new snid_sn: ident;
    out(secureChannel, (AV_REQ, imsi_sn, snid_sn));
    in(secureChannel, (=AV, =imsi_sn, =snid_sn,
        rand_sn: nonce, xres_sn: resp, mac_sn: mac,
        kasme_sn: asmeKey, ck_sn: cipherKey, ik_sn: integKey));
    out(pubChannel, (CHALLENGE, rand_sn, mac_sn, snid_sn));
    in(pubChannel, (=RES, =xres_sn));
    event endSN(imsi_sn, ck_sn, ik_sn);
    new fresh_sn: nonce;
    event begMS(imsi_sn, ck_sn, ik_sn, cap_sn);
    out(pubChannel, (ASSMC, cap_sn, cap_sn, fresh_sn,
        f9((cap_sn, cap_sn, fresh_sn), ik_sn)));
    in(pubChannel, (=ASSMComplete, =as_smcomplete_msg,
        =f9(as_smcomplete_msg, ik_sn)));
    if cap_sn = false then
        event disableEnc;
        out(pubChannel, (MSG, payload, f9(payload, ik_sn)))
    else
        out(pubChannel, (MSG, sencrypt_as(payload, ck_sn),
            f9(sencrypt_as(payload, ck_sn), ik_sn))).

(*process representing HN*)
let processHN =
    in(secureChannel, (=AV_REQ, imsi_hn: ident, snid_hn: ident));
    (*Generate athenication vectors*)
    new rand_hn: nonce;
    get keys(=imsi_hn, ki_hn) in
    let mac_hn: mac = f1(ki_hn, rand_hn) in
    let xres_hn: resp = f2(ki_hn, rand_hn) in
    let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
    let ik_hn: integKey = f4(ki_hn, rand_hn) in
    let kasme_hn: asmeKey = kdf_asme(ck_hn, ik_hn, snid_hn) in
    out(secureChannel, (AV, imsi_hn, snid_hn, rand_hn, xres_hn,
        mac_hn, kasme_hn, ck_hn, ik_hn)).

process
    ((!processMS) | processSN | processHN)
```

S10+. UMTS I || LTE II–IV || UMTS IV, AV = 4G AV + CK + IK

```
(*Public channel between the MS and the SN*)
free pubChannel: channel.
(*Secure channel between the SN and the HN*)
free secureChannel: channel [private].
free sChannelSnBts: channel [private].

(*types*)
type key.
type ident.
type nonce.
type msgHdr.
type resp.
type cipherKey.
type integKey.
type mac.
type msgMac.
type asmeKey.
type nasEncKey.
type nasIntKey.

(*constant message headers*)
const CAP: msgHdr.
const ID: msgHdr.
const AV_REQ: msgHdr.
const AV: msgHdr.
const CHALLENGE: msgHdr.
const RES: msgHdr.
const NASSMC: msgHdr.
const NASSMComplete: msgHdr.
const ASSMC: msgHdr.
const ASSMComplete: msgHdr.
const MSG: msgHdr.

(*Functions*)
fun f1(key, nonce): mac.
fun f2(key, nonce): resp.
fun f3(key, nonce): cipherKey.
fun f4(key, nonce): integKey.
fun f9(bitstring, integKey): bitstring.
fun kdf_asme(cipherKey, integKey, ident): asmeKey.
fun kdf_nas_enc(asmeKey) : nasEncKey.
fun kdf_nas_int(asmeKey): nasIntKey.
fun finteg_nas(bitstring, nasIntKey): msgMac.

fun sencrypt_nas(bitstring, nasEncKey): bitstring.
reduc forall m: bitstring, k: nasEncKey;
    sdecrypt_nas(sencrypt_nas(m, k), k) = m.

fun sencrypt_as(bitstring, cipherKey): bitstring.
reduc forall m: bitstring, k: cipherKey;
    sdecrypt_as(sencrypt_as(m, k), k) = m.

(*Type Converter*)
fun bool2bitstring(bool): bitstring [data, typeConverter].

reduc       encCapability() = true;
```

```
    encCapability() = false.

(*the table ident/keys
  The key table consists of pairs
  (ident, key) shared between MS and HN
  Table is not accessible by the attacker*)
table keys(ident, key).

(*SMC command msg*)
free nas_smcomplete_msg: bitstring.
free as_smcomplete_msg: bitstring.

free payload: bitstring [private].
event disableEnc.
(*When the attacker knows s, *)
(* the event disableEnc has been executed.*)
query attacker(payload) ~> event(disableEnc).
query attacker(payload).

free secret: bitstring [private].
query attacker(secret).
fun senc_int_nas(bitstring, nasIntKey): bitstring.
reduc forall m: bitstring, k: nasIntKey;
sdec_in_nas(senc_int_nas(m, k), k) = m.
fun senc_int_as(bitstring, integKey): bitstring.
reduc forall m: bitstring, k: integKey;
sdec_in_as(senc_int_as(m, k), k) = m.

not attacker(new ki).

(*Authentication queries*)
event begSN(ident, ident, asmeKey).
event endSN(ident, ident, asmeKey).
event begMS(ident, ident, asmeKey, bool).
event endMS(ident, ident, asmeKey, bool).
event begMS_AS(ident, cipherKey, integKey, bool).
event endMS_AS(ident, cipherKey, integKey, bool).

query x1: ident, x2: ident, x3: asmeKey;
    event(endSN(x1, x2, x3)) ~>  event(begSN(x1, x2, x3)).
query x1: ident, x2: ident, x3: asmeKey, x4: bool;
    event(endMS(x1, x2, x3, x4)) ~>
event(begMS(x1, x2, x3, x4)).
query x1: ident, x2: cipherKey, x3: integKey, x4: bool;
    event(endMS_AS(x1, x2, x3, x4)) ~>
event(begMS_AS(x1, x2, x3, x4)).

(*AS SMC procedure in process MS*)
let pMSAS(ck_ms: cipherKey, ik_ms: integKey,
imsi_ms: ident, cap_ms: bool) =
    in(pubChannel, (=ASSMC, =cap_ms, enableEnc_as_ms: bool,
        =f9((cap_ms, enableEnc_as_ms), ik_ms)));
    out(pubChannel, (ASSMComplete, as_smcomplete_msg,
        f9(as_smcomplete_msg, ik_ms)));
    event endMS_AS(imsi_ms, ck_ms, ik_ms, cap_ms);
    in(pubChannel, (=MSG, datamsg: bitstring,
        =f9(datamsg, ik_ms)));
    out(pubChannel, sencrypt_as(secret, ck_ms));
    out(pubChannel, senc_int_as(secret, ik_ms));
    if enableEnc_as_ms = true then
        let msgcontent: bitstring =
            sdecrypt_as(datamsg, ck_ms) in 0.

(*process respresenting MS*)
let processMS =
    (*The identity of the MS*)
    new imsi_ms: ident;
    (*Pre-shared key*)
    new ki: key;
    (*Insert id/pre-shared key pair into the private table*)
    insert keys(imsi_ms, ki);
    (*MS non-deterministically choose
        the capability of encryption*)
    let cap_ms: bool = encCapability() in
    out(pubChannel, (CAP, cap_ms));
    out(pubChannel, (ID, imsi_ms));
    in(pubChannel, (=CHALLENGE, rand_ms: nonce,
        =f1(ki, rand_ms), snid_ms: ident));
    let res_ms: resp = f2(ki, rand_ms) in
    let ck_ms: cipherKey = f3(ki, rand_ms) in
    let ik_ms: integKey = f4(ki, rand_ms) in
    let kasme_ms: asmeKey = kdf_asme(ck_ms, ik_ms, snid_ms) in
    event begSN(imsi_ms, snid_ms, kasme_ms);
    out(pubChannel, (RES, res_ms));
    (*NAS SMC procedure*)
    let knasenc_ms: nasEncKey = kdf_nas_enc(kasme_ms) in
    let knasint_ms: nasIntKey = kdf_nas_int(kasme_ms) in
    in(pubChannel, (=NASSMC, enableEnc_nas_ms: bool, =cap_ms,
        =finteg_nas((enableEnc_nas_ms, cap_ms), knasint_ms)));
    event endMS(imsi_ms, snid_ms, kasme_ms, cap_ms);
    (*NAS key secrecy*)
    out(pubChannel, sencrypt_nas(secret, knasenc_ms));
    out(pubChannel, senc_int_nas(secret, knasint_ms));
    if enableEnc_nas_ms = false then
        out(pubChannel, (NASSMComplete, nas_smcomplete_msg,
            finteg_nas(nas_smcomplete_msg, knasint_ms)));
        pMSAS(ck_ms, ik_ms, imsi_ms, cap_ms)
    else
        out(pubChannel, (NASSMComplete,
            sencrypt_nas(nas_smcomplete_msg, knasenc_ms),
            finteg_nas(sencrypt_nas(nas_smcomplete_msg,
            knasenc_ms), knasint_ms))); (*[Msg 8]*)
        pMSAS(ck_ms, ik_ms, imsi_ms, cap_ms).

(*process representing e-nodeB*)
let processBS =
    in(sChannelSnBts, (ck_bs: cipherKey, ik_bs: integKey,
        imsi_bs: ident, cap_bs: bool));
    event begMS_AS(imsi_bs, ck_bs, ik_bs, cap_bs);
    out(pubChannel, (ASSMC, cap_bs, cap_bs,
        f9((cap_bs, cap_bs), ik_bs)));
    in(pubChannel, (=ASSMComplete, =as_smcomplete_msg,
        =f9(as_smcomplete_msg, ik_bs)));
    if cap_bs = false then
        event disableEnc;
        out(pubChannel, (MSG, payload, f9(payload, ik_bs)))
    else
        out(pubChannel, (MSG, sencrypt_as(payload, ck_bs),
            f9(sencrypt_as(payload, ck_bs), ik_bs))).

(*process representing MME*)
let processSN =
    in(pubChannel, (=CAP, cap_sn: bool));
    in(pubChannel, (=ID, imsi_sn: ident));
    new snid_sn: ident;
    out(secureChannel, (AV_REQ, imsi_sn, snid_sn));
    in(secureChannel, (=AV, =imsi_sn, snid_hn_sn: ident,
        rand_sn: nonce, xres_sn: resp, mac_sn: mac,
        kasme_sn: asmeKey, ck_sn: cipherKey, ik_sn: integKey));
    out(pubChannel, (CHALLENGE, rand_sn, mac_sn, snid_sn));
    in(pubChannel, (=RES, =xres_sn));
    event begMS(imsi_sn, snid_hn_sn, kasme_sn, cap_sn);
    (*NAS SMC procedure*)
    let knasenc_sn: nasEncKey = kdf_nas_enc(kasme_sn) in
    let knasint_sn: nasIntKey = kdf_nas_int(kasme_sn) in
    out(pubChannel, (NASSMC, cap_sn, cap_sn,
        finteg_nas((cap_sn, cap_sn), knasint_sn)));
    in(pubChannel, (=NASSMComplete, msg_nas:bitstring,
        =finteg_nas(msg_nas, knasint_sn)));
    if cap_sn = true then
        if sdecrypt_nas(msg_nas, knasenc_sn)
        = nas_smcomplete_msg then
            event endSN(imsi_sn, snid_hn_sn, kasme_sn);
            out(sChannelSnBts, (ck_sn, ik_sn, imsi_sn, cap_sn))
        else 0
    else
        if cap_sn = false then
            if msg_nas = nas_smcomplete_msg then
                event endSN(imsi_sn, snid_hn_sn, kasme_sn);
                out(sChannelSnBts, (ck_sn, ik_sn,
                    imsi_sn, cap_sn))
            else 0
        else 0.

(*process representing HN*)
let processHN =
    in(secureChannel, (=AV_REQ, imsi_hn: ident, snid_hn: ident));
    (*Generate athenication vectors*)
    new rand_hn: nonce;
    get keys(=imsi_hn, ki_hn) in
    let mac_hn: mac = f1(ki_hn, rand_hn) in
    let xres_hn: resp = f2(ki_hn, rand_hn) in
    let ck_hn: cipherKey = f3(ki_hn, rand_hn) in
    let ik_hn: integKey = f4(ki_hn, rand_hn) in
    let kasme_hn: asmeKey = kdf_asme(ck_hn, ik_hn, snid_hn) in
    out(secureChannel, (AV, imsi_hn, snid_hn, rand_hn,
        xres_hn, mac_hn, kasme_hn, ck_hn, ik_hn)).
```

process

((! processMS) | processSN | processBS | processHN)