3GPP TSG SA WG3 Security — S3#13 **S3-000315**
May 24-26, 2000
Yokohama, Japan

Source: **TR-45 AHAG**

To: **TSG SA WG3**

Title: **Use of SHA-1 for AKA f0-f5**

Cc:

Contact Person: Frank Quick, Vice-Chair TIA TR-45 Ad Hoc Authentication Group (AHAG)
E-mail: fquick@qualcomm.com
Tel: +1-858-658-3608

At the joint meeting of S3 and TR-45 in Stockholm, Sweden, April 12, 2000, it was agreed that AHAG would send S3 the latest proposal on the use of SHA-1 as a hash function in AKA. The attached contribution provides this information.

Please note that this contribution represents work in progress that has not been formally approved or adopted by TR-45. Although the document is very close to completion, some details are still subject to change. We invite your comments and suggestions for improvement.

Please note also that this document is based on the latest draft of 3GPP document 33-105 available to us (v.3.3.0). If 33-105 undergoes further revision, changes to the SHA-1 proposal may accordingly be necessary.

May 19,2000

TITLE:

SHA based functions for Authenticated Key Agreement

SOURCE:

Lucent Technologies

Sarvar Patel	Zulfikar Ramzan	Ganesh Sundaram	Marcus Wong
(973)386-6558	(617)253-2345	(973)739-4489	(973)739-1258
sarvar@lucent.com	zulfikar@mit.edu	ganeshs@lucent.com	mw888mw@lucent.com

ABSTRACT:

This contribution presents an adaptation of the SHA-1 algorithm for functions f0, f1, f1*, f2, f3, f4, and f5 needed in the AKA scheme.

RECOMMENDATIONS:

Review and accept the proposed adaptation of SHA-1 for default functions in 3GPP security architecture.

May 19,2000

1. Introduction

The various functions f0, f1, f1*, f2, f3, f4, f5 in the AKA protocol require different cryptographic primitives:

1. ***MAC***: A message authentication code (MAC) is used to produce a "tag" of a message, and typical applications include, challenge response and message integrity. The user and serving system share a secret key and the MAC function is expected to have the following property: Given several message tag pairs, (x_1,t_1), (x_2,t_2), …., (x_q,t_q) an adversary who does not know the secret key cannot create a *new* message tag pair in any reasonable amount of time.
2. ***PRG***: A pseudo random generator (PRG) is used to produce bit strings which "appear random". Typical applications include challenge response and other authentication applications. A PRG is expected to have the following property: Given a secret seed, a PRG outputs bits with the property that any adversary who does not know the secret key, cannot predict *any* of the bits (given a past history) in any reasonable amount of time.
3. ***PRF***: A pseudo random function (PRF) is used to produce random looking bit strings given a secret key and any known input. Typical applications include, for example, session key generation. A PRF is expected to have the property that: given any sequence of outputs, an adversary who does not know the secret key is unable to differentiate between the output sequence and a sequence of truly random bits, in any reasonable amount of time.

Note that the PRF and PRG properties are stronger requirements compared to the MAC property. In particular, a function that is believed to be a secure MAC may not be a secure PRG since one or more bits of the output may be predictable. But every PRF can be used as a PRG, and again every PRF can be used as a MAC. So it seems attractive to use a PRF for all functions, but because of the security requirements a PRF can be a lot more complex to compute when compared to a MAC.

Below we list the primitives required by various AKA functions:

- *f0*: This function is used to generate the RAND, and so is expected to behave like a PRG.
- *f1, f1*, f2* : These functions are used to generate the MAC as a part of AUTN, the MACS as a part of re-synchronization procedure, and RES, respectively, and are expected to behave like a MAC.
- *f3, f4*: These functions are used to generate the Ciphering Key CK and Integrity Key IK, and are expected to behave like a PRF.
- *f5*: This function is used to generate the Anonymity Key AK. Ideally this function is expected to behave like a PRF. But in AKA, the role of f5 is limited to masking the relatively short sequence number SQN. Hence it may be secure enough to use a MAC function.

In our proposal the secure hash algorithm (SHA) is used as a MAC function. SHA was designed by the National Institute of Standards and Technology (NIST) along with the National Security Agency (NSA) to be used as the core hashing algorithm for the Digital Signature Standard (DSS). The algorithm was closely modeled after the MD family of message digest algorithms developed by Rivest. SHA is also used for message authentication as described in [RFC2404]. Over the years, SHA has been crypt-analyzed and scrutinized by both the academic and industrial scientific communities and no weaknesses on the MAC property have been reported. In addition, using the MAC property one can create a PRF as well PRG out of SHA. Note that SHA was initially proposed by NIST as a collision resistant function, however, its use as a MAC function was exploited early and it has withstood the test of time. SHA is also believed to possess

some pseudo random properties [DSS]. Since pseudo random generation and key generation are vital to the security of the entire system, we want to make only the conservative assumption that SHA possesses the MAC properties and build the other primitives (PRG and PRF) on top of it. To summarize:

- SHA's MAC property is widely believed.
- SHA has been part of the Internet standard HMAC which uses the MAC property of the SHA compression function.
- We can create the other PRG and PRF functions from the MAC function.
- It is openly and widely available.

2. Details of Functions

The proposal is based on the simple basic premise:

- The same ***SHA-1*** algorithm is used for all functions.
- Every bit of the output of these functions is ensured to be ***cryptographically secure*** based on widely believed properties of SHA-1.
- Whenever the expected length of the function output does not exceed 64 bits, the ***MAC property*** of SHA-1 is claimed, and the algorithm is used directly.
- Whenever the expected length of the function exceeds 64 bits, the SHA-1 is "***whitened***" by following it with a linear congruential hashing procedure over a finite field of order 2^{160}.
- The "White SHA" procedure is than ***executed multiple times*** as necessary, producing 64 cryptographically secure bits with each execution.
- For each execution all inputs to the SHA-1 remain the same except for the value of ***Internal Counter***, which is updated for each function call. The counter mode operation allows additional flexibility. For example, if the 100^{th} output is needed, the counter mode takes only one function call into the procedure to generate required output
- The constants for the "whitening" operation are fixed across the board for all functions. These values are chosen at ***random*** and can be ***publicly known***.
- With a view towards maintaining operator identity, a ***Family Key*** is used as an input to the functions.
- The ***Function Type*** input parameter differentiates between functions.

In the rest of this section we present the algorithms and pseudo code for the various functions. The sample ANSI-C code including Test Vectors can be provided on request.

2.1 RAND Generation Procedure *f0*

Procedure name:

f0

Inputs from calling process:

Random Secret Seed (K)	128 bits
Type identifier (f0)	8 bits
Family Key (Fmk)	32 bits

Inputs from internal stored data:

Counter (C)	64 bits.

Outputs to calling process:

None.

Outputs to internal stored data:

Buffer	64 bits

The function f0 is a pseudo random generator algorithm that is provably as hard as SHA-1. The function is presented with the random secret Seed, which is chosen by the operator, as well as a Counter parameter, which is initialized to 0 at the Authentication Center (AC) once, and is incremented every time the function is called. The procedure returns 64 pseudo random bits every time it is invoked; the calling process is responsible for incrementing the counter and repeatedly calling the procedure in order to generate a required number of pseudo random bits. The generator is specified in Exhibit 2.1-1.

Exhibit 2.1-1. Pseudocode of f0

SHA-based PRG:

Input:

Random Secret Seed	K: 128 bits
Type identifier	f0: 8 bits
Counter C	C: 64 bits
Family Key	Fmk: 32 bits.

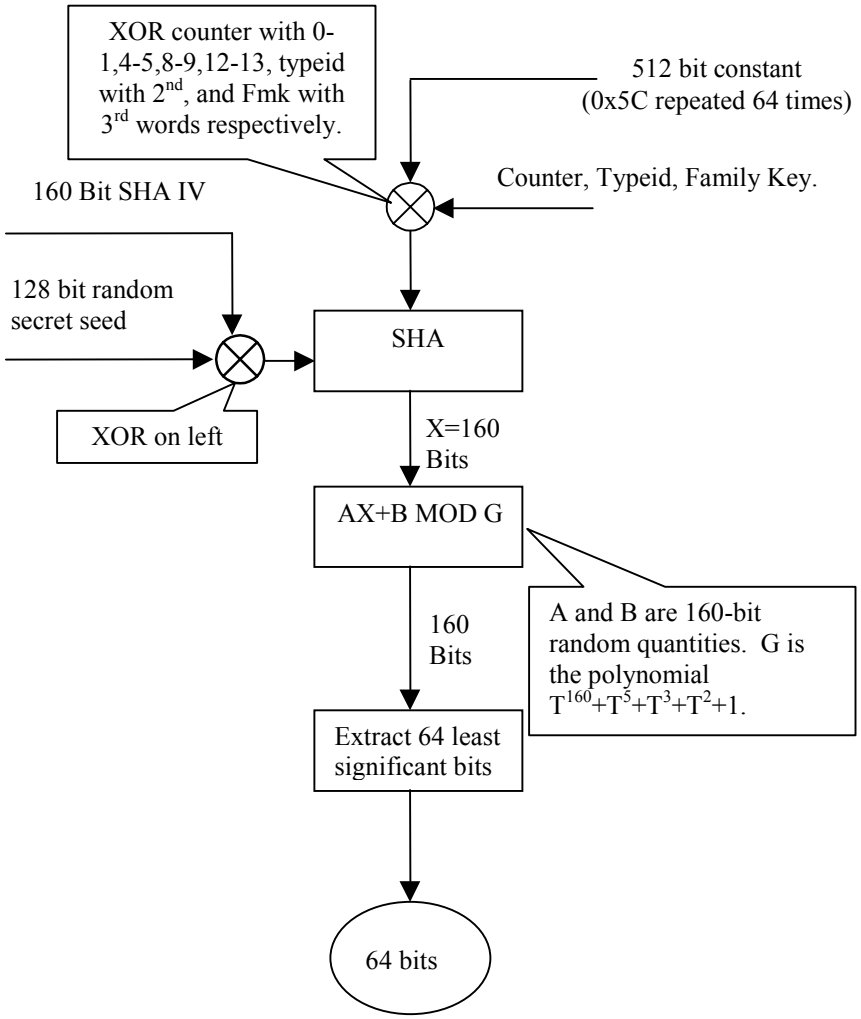
Output:

64 bits of key material stored in buffer per call.

Procedure:

1. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times
2. Load seed and counter values as follows:
   XOR seed into the leftmost (most significant) 128 bits of IV. The 64 bit counter value is XORed into the (0^{th}, 1^{st}) words, (4^{th}, 5^{th}) words, (8^{th}, 9^{th}) words, and (12^{th}, 13^{th}) words. 0^{th} is the least significant word and 15^{th} is the most significant. Next, a "type constant" (unique to function f0) is XORed into the 2^{nd} word, and the "family key" is XORed into the 3^{rd} word.
3. Run SHA to produce a 160-bit output.
4. The polynomial (AX + B mod G) is calculated, where:
   A and B are predetermined 160-bit random numbers (treated as binary polynomials in the variable T) and remain constant for the life of the system, and are equal across system.
   X is the 160-bit output from the SHA operation, treated as a binary polynomial in the variable T.
   G is the polynomial T^160+T^5+T^3+T^2+1.
5. The Least Significant 64 bits of the result are returned and stored in buffer.
6. The next time the function is invoked, increment the counter value by 1 and repeat steps 1 through 5.
7. Repeat procedure as many times as needed.

Exhibit 2.1-2 Pseudo Random Generator.



2.2 Ciphering Key (CK) and Integrity Key (IK) Generation Procedures *f3,f4*.

Procedure name:	
f3,f4	
Inputs from calling process:	
Subscriber Authentication Key (K)	128 bits
Type Identifier (f3 or f4)	8 bits
Random Number (RAND)	128 bits
Family Key (Fmk)	32 bits
Inputs from internal stored data:	
None.	
Outputs to calling process:	
None.	
Outputs to internal stored data:	
Key_buffer	128 bits

The functions f3,f4 are pseudo random functions which are provably as hard as SHA-1. The function is presented with a secret key, a family key, a type identifier, and a random number. The procedure is ran twice returning 64 pseudo random bits with each iteration; an internal index is incremented each time the procedure is run; as the result, the function generates 128 of pseudo random bits. The algorithm is specified in Exhibit 2.2-1.

Exhibit 2.2-1. Pseudocode of f3,f4

SHA-based PRF:

Input:

Subscriber Authentication Key	K: 128 bits
Random Number	RAND: 128 bits
Family Key	Fmk: 32 bits
Type identifier	fi: 8 bits

Output:

128 bits of key material stored in CK or IK.

Procedure:

1. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times

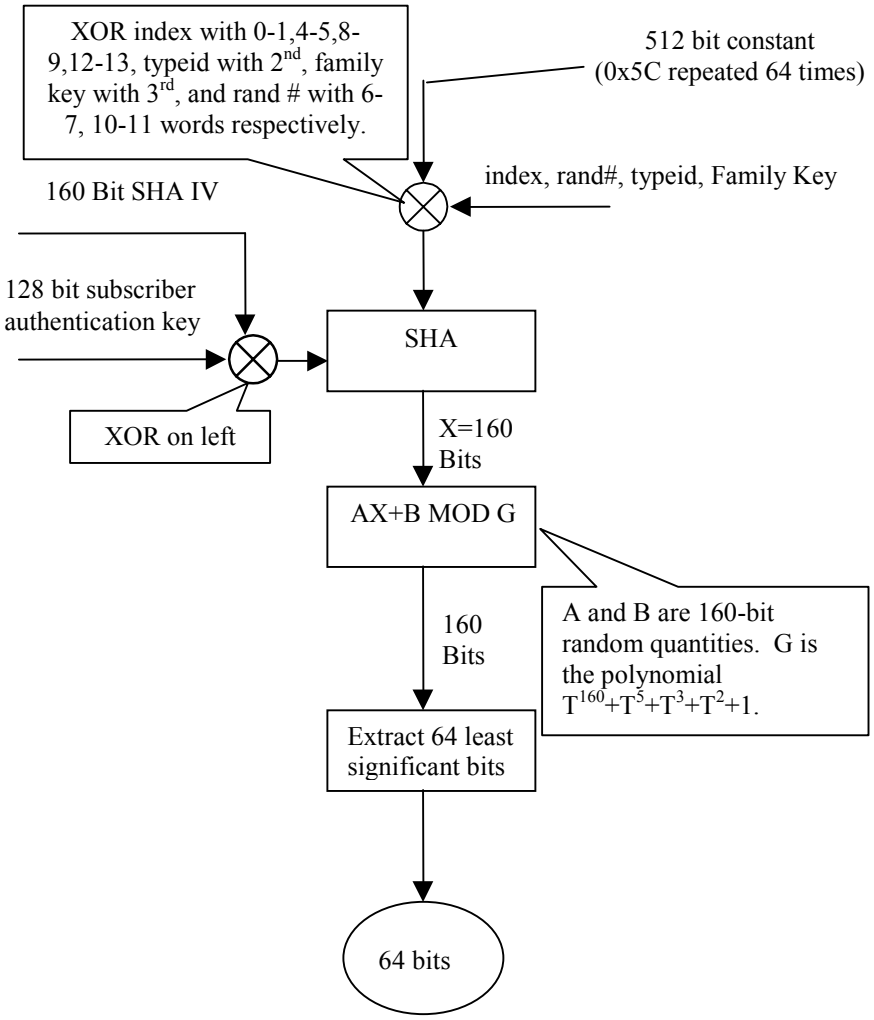
2. Load the subscriber authentication key and index value as follows:
   XOR the subscriber authentication key into the leftmost (most significant) 128 bits of IV. The 64 bit index value, initialized to 0, is XORed into the (0^{th}, 1^{st}) words, (4^{th}, 5^{th}) words, (8^{th}, 9^{th}) words, and (12^{th}, 13^{th}) words. 0^{th} is the least significant word and 15^{th} is the most significant. Next, a "type constant" (unique to function f3, f4) is XORed into the 2^{nd} word, and the family key is XORed with the 3^{rd} word. The 128 bit random number is split into two parts (64 bits each). The least significant 64 bits are XORed with the 6^{th} and 7^{th} words, and the most significant 64 bits are XORed with the 10^{th} and 11^{th} words.

3. Run SHA to produce the 160-bit output.

4. The polynomial AX + B mod G is calculated, where:
   A and B are predetermined 160-bit random numbers (treated as binary polynomials in the variable T) and remain constant for the life of the system,
   X is the 160-bit output from the SHA operation, treated as a binary polynomial in the variable T.
   G is the polynomial T^160+T^5+T^3+T^2+1. Extract the least significant 64 bits and store it in the key buffer.

5. Steps 1 through 4 are repeated 2 times, with the index incremented between iterations. This gives a total of 128 bits. Store these 128 bits in CK or IK (accordingly).

Exhibit 2.2-3 Key Scheduler.



2.3 Message Authentication Code & Authentication Signature Generation Procedures, *f1, f1*, f2, f5*

Procedure name:	
f1, f1*, f2, f5	
Inputs from calling process:	
Subscriber Authentication Key (K)	128 bits
Random Number (RAND)	128 bits
Family Key (Fmk)	32 bits
Type Identifier (f1,f1*, f2, or f5)	8 bits
SQN (set to 0 for f2 and f5)	48 bits
AMF(set to 0 for f2 and f5)	16 bits
Inputs from internal stored data:	
None.	
Outputs to calling process:	
None.	
Outputs to internal stored data:	
Buffer	160 bits

The function f1, f1*, f2, and f5 require the MAC property, hence the SHA-1 compression function is used as follows: the input arguments are loaded in the SHA payload and the resulting 160-bit digest is truncated to the required output length.

Exhibit 2.1-4. Pseudocode of f1, f1*, f2, f5

SHA-based MAC:

Input:

Subscriber Authentication Key	K: 128 bits
Type identifier	f1,f1*,f2,f5: 8 bits
Family Key	Fmk: 32 bits
Random Number	RAND: 128 bits
SQN	SQN: 48 bits
AMF	AMF: 16 bits

Output:

160 bits truncated to the required f1, f1*, f2, f5 output length.

Procedure:

1. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times
2. Load the subscriber authentication key and counter value as follows:
   XOR the subscriber authentication key into the leftmost (most significant) 128 bits of IV. A "type constant" (unique to function f1, f1*, f2) is XORed into the 2nd word. The 32 bit family key is XORed into the 3rd word, 128 bit RAND is XORed into words 4-7th. For f1 and f1* the 32 bit SQN is XORed into the 8th and 9th words, and the AMF is XORed into the 10th word.
3. Run SHA to produce a 160-bit output.
4. Only the required least significant bits are used as output for each of the f1, f1*, f2, f5 functions.