Improved Attack on the Cellular Authentication and Voice Encryption Algorithm

Praveen S.S Gauravaram and William L. Millan

Information Security Research Centre,
Queensland University of Technology,
GPO BOX 2434, Brisbane, QLD, 4001, Australia
praveen@isrc.qut.edu.au,millan@isrc.qut.edu.au

Abstract. We present new cryptanalysis of the Telecommunications hash algorithm known as Cellular Authentication and Voice Encryption Algorithm (CAVE). The previous guess-and-determine style reconstruction attack requires 2^{91} (resp. 2^{93}) evaluations of CAVE-4 (resp. CAVE-8) to find a single valid pre-image (one which satisfies the input redundancy). Here we present a new attack that finds *all* valid pre-images with effort equivalent to around 2^{72} evaluations of the algorithm for both CAVE-4 and CAVE-8.

1 Introduction

CAVE is a cryptographic primitive approved by the Telecommunication International Association (TIA) to be used for authentication, data protection and anonymity of the second generation Code Division Multiple Access (CDMA) networks [7]. It is also used to provide security for North American IS-41C mobile phones [2] and IS-54 dual mode cellular systems [1]. It had been known in the telecommunications industry for some time that CAVE has weaknesses [4]. The first known attack on the CAVE algorithm in the open literature was presented by Millan [3] in 1998 detailing a reconstruction attack that demonstrated that CAVE can not be considered a secure hash function. The other CDMA encryption algorithms broken during that period were ORYX [5], a stream cipher used for the protection of cellular data transmissions and CMEA [6] a block cipher which protects a user's confidential keypad data during a telephone call.

The CAVE algorithm authenticates a legitimate subscriber to the CDMA network[1] and prevents the network and customers of mobile phones from the cloning fraud [7]. CAVE is designed to deter radio access

[1] CAVE is used in a similar fashion on other wireless communication networks

to the 32-bit Electronic Serial Number(ESN), Mobile Identification Number(MIN) and the 64-bit Authentication key (A-Key) of a CDMA mobile phone. CAVE uses ESN, A-Key and a random number(RANDSSD) generated by the Home Location Register(HLR), an integral component of a CDMA network which permanently stores subscriber information, to generate a 128-bit intermediate key called "shared secret data", SSD-A and SSD-B. The 64-bit SSD-A is used for signature authentication and SSD-B is used for cryptographic key generation.

CAVE uses SSD-A and a broadcast random number(RAND) generated at the Mobile Switch Center(MSC) to produce an 18-bit random authentication signature (AUTH_SIGNATURE). Base station verifies this signature allowing the legitimate subscriber to access the network. CAVE uses the 64-bit SSD-B data to generate a private long code mask which is used for voice scrambling for data privacy over the CDMA interface of the mobile phone. SSD-B is also used to generate keys for other encryption algorithms like ORYX(32 bits) and CMEA(64 bits). CAVE is also used to verify A-Key by truncating the 128-bit hash output to 18 bits and comparing this value with the A-key checksum. The initial loading of CAVE for A-key verification and SSD generation is shown in Table 1. Figure 1 shows different applications of CAVE and Figure 2 shows how CAVE is used in reality in IS-41C and IS-54 communication systems.

Our new attack on 4-Round CAVE uses pre-computed look-up tables (LUTs) to exploit the additional weaknesses (discussed in Section 4) to obtain the set of *all* valid pre-images for any given output. This is in contrast to the previous attack on CAVE [3] which finds a single pre-image with expected effort equivalent to evaluating 2^{11} instances of 4-Round CAVE. That method must be repeated around 2^{80} times in order to generate just one example of input data that has redundancy consistent with the input processing stage of the specific CAVE applications (for example A-key verification and SSD generation as shown in Table 1). The total complexity for the attack [3] on a 4-Round CAVE is $2^{11} * 2^{80} = 2^{91}$ to find a single valid input for a given 128-bit hash result, where the unit of effort is an evaluation of 4-Round CAVE. In comparison, the complexity of our new attack is less effort than computing 2^{72} evaluations of 4-Round CAVE to obtain a list of *all* valid pre-images, including those which satisfy the linear input redundancy required by the various applications. Against 8-Round CAVE, the method of [3] requires effort equivalent to the evaluation of $2^{13} * 2^{80} = 2^{93}$ instances of 8-Round CAVE, just to find a single pre-image that satisfies the application specific redundancy (which is eight times the absolute effort required to fully break the 4-Round version). In comparison, the new attack applied to 8-Round CAVE requires an effort equivalent to less than 2^{72} evaluations of that algorithm, or twice the overall effort to break CAVE-4. As the number of rounds of CAVE is increased, the relative advantage of our attack over the previous method also increases. The significant reduction in effort makes this new attack more threatening for CAVE in practice.

This paper is organised as follows: In Section 2 we review the structure and operation of the CAVE algorithm. In Section 3 we discusses the previous attack on CAVE and point out some relevant properties of that approach. Our new attack is described in detail in Section 4. Section 5 analyses the expected complexity of the new attack. Finally we make some concluding remarks in Section 6.

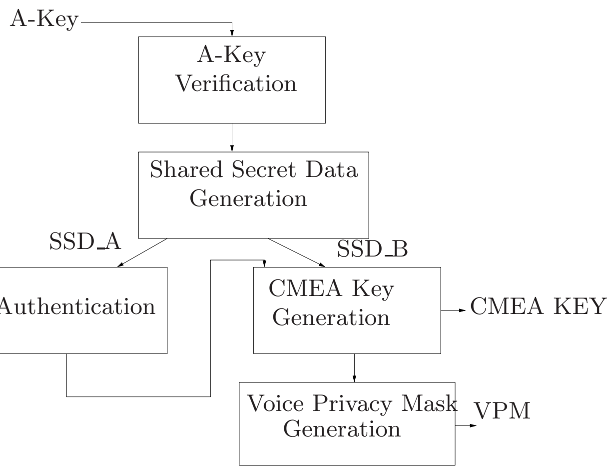

Fig. 1. Many roles of CAVE

Table 1. Initial loading of CAVE

CAVE Component	A-key Verification	SSD Generation
LFSR	32 MSBs of A-key	32 LSBs of RANDSSD
$sreg[0, 1, \ldots, 7]$	A-key	A-key
$sreg[8]$	Algorithm version	Algorithm version
$sreg[9, 10, 11]$	24 LSBs of A-key	24 MSBs of RANDSSD
$sreg[12, \ldots, 15]$	ESN	ESN
offset_1	128	128
offset_2	128	128

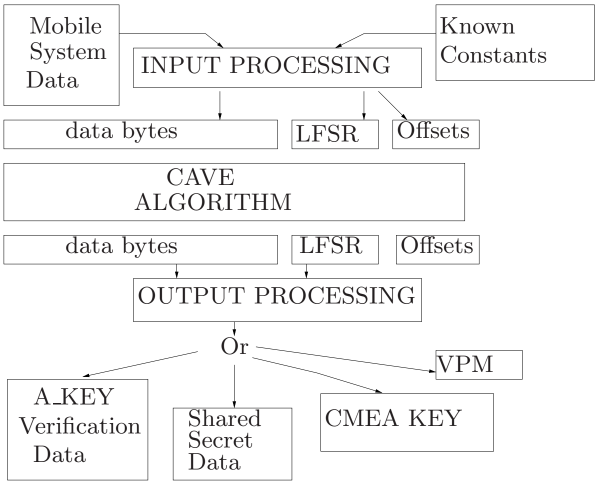

Fig. 2. Usage of CAVE in IS-41C and IS-54 phone systems

2 CAVE Algorithm

A report describing the CAVE algorithm and its various applications is available at [1]. In this paper we follow the notation used in [3].

The main components of the algorithm are sixteen 8-bit data *registers*, two 8-bit offsets *offset_1* and *offset_2* and a 32-bit Linear Feedback Shift Register (LFSR). CAVE operates in four or eight rounds as per the requirements of a specific application with each round having 16 register update *phases*. The 32-bit LFSR contains four separate register bytes $LFSR_A$, $LFSR_B$, $LFSR_C$ and $LFSR_D$ with a primitive feedback polynomial whose feedback function is defined as:

$$L_{t+32} = L_t \oplus L_{t+1} \oplus L_{t+2} \oplus L_{t+22}$$

For each phase, CAVE uses bytes from the LFSR, the offsets and two 8*4 LUTs or SBoxes[2] to modify one of the registers. The offsets *offset_1* and *offset_2* act as pointers into the low and high CAVE tables which are represented as $CT\_low[\cdot]$ and $CT\_high[\cdot]$. The steps that take place in the low segment of CAVE are expressed as

$$\mathit{offset\_1} = \mathit{offset\_1}_{prev} + (LFSR_A \oplus sreg[i]) \bmod 256 \tag{1}$$

$$\mathit{temp\_low} = CT\_low[\mathit{offset\_1}] \tag{2}$$

[2] each table has 256 nibble values

and the steps for high segment are similar. This segment operation is shown in Figure 3.

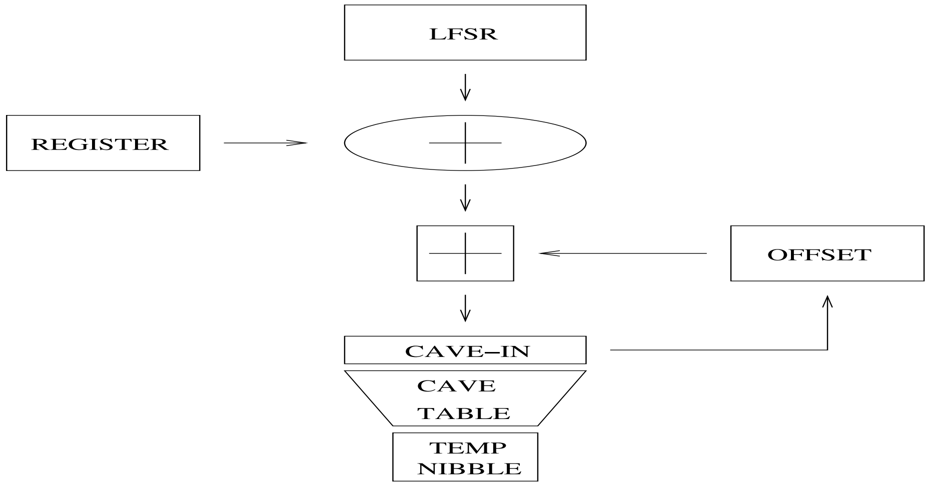

Fig. 3. Segment operation in CAVE

The byte $offset\_1_{prev}$ represents the previous value of the offset byte initialized as a constant (See Table 1). CAVE cycles the LFSR linearly to the right when the nibbles become equal to the corresponding $low/high$ order bits of $sreg[i]$, where i is a particular phase in a round. When they become unequal, CAVE computes *temp* byte by concatenating the nibbles *temp_low* and *temp_high* and moving to the next phase of a round. If the compared values get equal, there would be an extra cycle of the LFSR and the above calculation is repeated with the latest LFSR byte and offset values. In the very rare event that the count of these extra cycles reaches thirty-two, then byte $LFSR_D$ is incremented modulo 256. After the completion of a phase, the LFSR cycles once resulting in a minimum of sixteen LFSR shifts in each round of CAVE. Between rounds, bits in the registers are shuffled by using the low CAVE table to define a byte permutation followed by a 1-bit rotation on the 128-bit register block as a whole.

2.1 Previous attack on CAVE

The previous attack on CAVE [3] shows that CAVE is not a secure one-way hash function. For a given 128-bit hash value, it was shown that an input (pre-image) to the algorithm can be found with effort equivalent to 2^{13}

executions of an 8-Round CAVE or 2^{11} executions of a 4-Round CAVE. The attack works by first guessing a 32-bit LFSR value and generating a sufficient number of LFSR cycles required for the attack using the known primitive feedback polynomial. The last round (Round 0) is reconstructed forward by guessing the two offset bytes and $sreg[0]$ and validating the guesses using the "sanity check" equation $ereg[15] \oplus temp[15] = sreg[0]$ where $sreg[0]$ is the value of the data register 0 at the start of the last round and $ereg[15]$ is the final value of the data register 15 at the end of that round. Once the final values of the offsets are established by the forward reconstruction of Round 0, the previous rounds (Rounds > 0) are reconstructed in the reverse direction of the algorithm by guessing first the $sreg[15]$ value of Round 1 (4-Round CAVE operates from Round 3 to Round 0). The validity of this guess is checked by using the sanity check equation $temp[15] \oplus ereg[15] = sreg[0]$ and once this is valid it means that particular round has been reconstructed correctly. In this fashion, the other rounds are reconstructed in the backward direction. This algorithm results in a single data set that is a CAVE input producing the given 128-bit hash value.

3 Weakness of CAVE

This section discusses some of the weak properties of CAVE that we use in the improved attack, which we present in the following section. These properties of CAVE were not exploited by the previous attack [3]. By searching the maximum likelihood data first would have improved the efficiency of the previous attack.

Imbalances in the CAVE tables

The rows of the CAVE table are permutations but the columns are not. So, for a given nibble output of the CAVE table, the low order input offset bits to the CAVE tables are not uniformly distributed. We call this imbalance in the low order input offsets as "nibble imbalance". Given some input data or a small guess, this property of CAVE assists in determining unknown values with a higher probability than just guessing. The nibble imbalances in the low and high CAVE tables are represented in Tables 2 and 3. Our attack (Section 4) directly uses this property.

Table 2 shows the frequencies of low nibble input bits to the low CAVE table giving a particular *temp_low* nibble output. Similarly, Table 3 shows the frequencies for the high CAVE table. On average, there are six low

Table 2. Imbalances in the LOW CAVE table

	Frequency of low nibble inputs giving specified low nibble output						
low nibble output	0	1	2	3	4	5	6
0	0,3,6,8,A,B	1,4,5,7,D,E	2,9,F	-	C	-	-
1	1,2,4,7,9,A,B,E,F	5,6	8,D	0,C	3	-	-
2	1,2,5,6,C,D	0,3,4,A,E,F	B,7	8,9	-	-	-
3	0,3,8,9,B,F	1,4,7,A,C,D	5,E	2,6	-	-	-
4	4,8,A,C,F	0,1,2,3,7,9,D,E	5	6,B	-	-	-
5	1,2,5,6,C,F	0,3,4,9,A,D,E	7	B	8	-	-
6	4,6,7,8,9,A,B,C,F	0,2,E	3,5	D	-	-	1
7	7,9,B,D,E,F	0,2,6,8,A,C	1,3	4,5	-	-	-
8	3,6,7,8,9,C,F	0,1,4	2,5,A,D,E	B	-	-	-
9	2,4,9,A,D	1,3,5,7,8,B,C,E	F	0,6	-	-	-
A	5,6,8,A,B	1,4,9,D,E,F	0,2,3,7,C	-	-	-	-
B	1,3,5,7,B,C	0,2,4,6,A,E	D,F	8,9	-	-	-
C	0,1,2,5,8,C	3,4,7,9,B,D,E	6,F	-	-	A	-
D	0,4,5,8,C,D	1,2,3,6,B	7,9,A,F	E	-	-	-
E	1,3,9,A,D,E	0,2,4,5,6,8,B	7	C	F	-	-
F	0,3,6,D,E,F	1,2,5,7,8,B	9,A,C	-	4	-	-

order input bits to the low CAVE table, with five being the minimum and nine being the maximum, which do not give a particluar *temp_low* nibble output. Similarly, on average, there are six low order inputs to the high CAVE table, with four being the minimum and eight being the maximum not giving a *temp_high* nibble output. These observations indicate a strongly non-uniform probability distribution of data in the CAVE tables when worked backwards. Our attack uses this information to check the most likely candidate values before less likely ones.

Correlations between LFSR bytes

The bits of $LFSR_A$ ($\{L_0, L_1, \ldots, L_7\}$) before the start of a phase used again in $LFSR_B$ ($\{L_8, L_9, \ldots, L_{15}\}$) after 8 LFSR cycles as L_7 jumps into the MSB position of $LFSR_B$ for every cycle. The LFSR bytes also do not depend on offsets and registers. So given the set $\{L_0, L_1, \ldots, L_7\}$ at time t, the set $\{L_8, L_9, \ldots, L_{15}\}$ is completely specified after time $t + 8$. These relations are expressed as follows.
For $\Delta t = 1$, $L_n(t) = L_{n-1}(t - 1)$
For $\Delta t = 4$, $L_n(t) = L_{n-4}(t - 4)$
For $\Delta t = 8$, $L_n(t) = L_{n-8}(t - 8)$
Our attack uses the known values of LFSR from the pre-computed look-up tables and test the above equations for every cycle of the LFSR.

Table 3. Imbalances in the HIGH CAVE table

	Frequency of high nibble inputs giving specified high nibble output				
high nibble output	0	1	2	3	4	5
0	1,4,6,9	3,5,7,8,A,B,C,D,F	0,2	E	-	-
1	0,1,2,A,F	5,6,7,8,9,B,D	3,C,E	4	-	-
2	0,3,7,9,A,E	1,2,4,5,B,D	6,C	8,F	-	-
3	0,2,3,6,7,D,F	4,5,A,E	8,9,B,C	-	1	-
4	2,5,8,A,B,C,D,E	3,4,9	0,1,6	F	7	-
5	4,7,9,D,E,F	1,2,5,8,C	0,3,6,B	A	-	-
6	0,2,3,8,9,D,F	1,A,C	4,5,6,B,E	7	-	-
7	0,4,7,C,E,F	2,3,5,6,8,A,B	9	1	D	-
8	4,C,D,F	0,1,2,6,8,9,A,B,E	3,5	7	-	-
9	8,A,B,D,F	1,2,3,6,7,9,C	0,5,E	4	-	-
A	0,1,3,4,7,A,C,E	6,8,B,D	5	2,9	F	-
B	3,5,6,8,A,B,E	1,2,4,7,9,F	C	0	-	D
C	0,1,2,5,F	3,4,8,9,B,D,E	7,A,C	6	-	-
D	1,4,6,7,B,C	0,2,5,8,9,A	D,F	E,3	-	-
E	1,4,6,7,9,D,E	3,5,8,C	0,2,B,F	-	A	-
F	5,6,7,D,F	0,1,3,A,B,C,E	2,8,9	4	-	-

4 An Improved Attack on CAVE

In this section we present our new approach to attacking CAVE[3]. We first use a precomputation to establish look-up-tables (LUTs) that define the operation of a segment in CAVE. Then, given a 128-bit hash output (the final values of the register bytes), these tables are used to guide a process which maintains lists of *all* data that is self-consistent. We generate these lists across consecutive segments within a phase, then consecutive phases within a round. Our experiments reveal that the resulting data sets after only two phases can specify about half of the unknown LFSR bits. Similarly, the process may be extended across more phases back to the start of the algorithm. Considering the big picture, the CAVE algorithm hashes the fixed input of 176 bits down to 128 bits. So it is expected that each output to have 2^{48} preimages. We make a 24-bit guess each time, so we expect to have 2^{24} elements in the list on each occasion. To decrease practical running times, we first compute LUTs representing the set of the most frequently repeated operations in CAVE. Firstly we explain the generation of these LUTs, then we present the attack algorithm.

[3] Note we do not consider any case where there were thirty-two continuous "extra" cycles in the segment operation of CAVE, since the probability of this event happening is around 2^{-128}.

4.1 Precomputing the CAVE Tables Backwards

Each segment (see Figure 3) of CAVE takes 24 input bits: it carries out an exclusive-OR operation on a byte from the input data register and an LFSR byte followed by a mod 256 addition of this result with the respective low/high offset byte. This gives the new offset byte which is the input to the low/high CAVE table giving low/high temp nibble.

The LUTs (1 and 2) can be constructed using exhaustive computation on the essential operations in the segments of CAVE. The offset, LFSR byte and the input data register add up to 24 bits giving 2^{24} different possible values. This computation on these 2^{24} possible input values results in an offset byte in each segment which is input to the CAVE table. The output of the CAVE table is a 4-bit *candidate temp_low* nibble or *temp_high* nibble and an "extra cycle" counter which acts as a flag. A flag value of one indicates the values of input to the CAVE table, for which the *low*/*high* temp nibble gets equal to the *low*/*high* order bits of $sreg[i]$ where i represents a particular phase of a round in CAVE. The *temp* nibble value is a *candidate* since, if an extra cycle is indicated, then CAVE cycles the LFSR once, thus changing the respective LFSR byte used in the above calculation by losing the LSB bit, gaining the MSB bit and shifting the other 7 bits by one bit. When the extra cycle count is zero, then the current *temp* nibble is used. Thus these two LUTs consist of 24-bit entries, an output offset byte, a *temp* nibble and an extra cycle counter.

4.2 The Attack

The overall attack algorithm could be described as follows.

Pre-computation Calculate the high and low LUTs.

- **Init**: Repeat, for all 2^{24} values of $sreg[15]$ and the pair of offset bytes:
- **Step 1**: Use the LUTs to find lists of valid inputs to both segments in two consecutive phases.
- **Step 2**: For each phase, combine the two segment data lists into a list of valid data for that phase.
- **Step 3**: Combine the adjacent phase lists into a single list for the pair of phases.

Final: Combine the remaining lists, filtering for consistency, to determine the list of all possible valid inputs.

Let's look at each of these operations in detail.

Init This step involves choosing a 24-bit value that makes up the two offsets and the start register byte. For a 4-Round CAVE, using the known 128-bit hash value, the values of end registers $ereg[0, 1, 2 \ldots 15]$ of Round 0 are found using the reverse round byte permutation on the 128-bit hash value followed by a left circular shift of the mixing registers. By guessing the value of the input data register of phase 15 (sreg[15]) of Round 0 to CAVE, the $temp_{14}$ output byte of phase 14 can be determined using the following expression:

$$ereg[14] \oplus sreg[15] = temp_{14}$$

The value $temp_{14}$ is the concatenation of $low/high\ temp_{14}$ nibble outputs of low and high CAVE tables. These nibbles are represented as $temp\_low_{14}$ and $temp\_high_{14}$ respectively. Since these nibbles are the final outputs of the low and high segments of phase 14 of Round 0 for the CAVE algorithm, the offsets $\mathit{offset\_1}_{14}$ and $\mathit{offset\_2}_{14}$ that have given these nibbles could not have produced extra cycles.

It means that

$$temp\_low_{14} \neq sreg[14]\&0x0F$$

and

$$temp\_high_{14} \neq sreg[14]\&0xF0.$$

There are 16 different possible values of $\mathit{offset\_1}_{14}$ and $\mathit{offset\_2}_{14}$ that can give these nibbles because of the *row permutations* of the CAVE tables. Our attack involves testing every value of these offset bytes.

Step 1. The first step in the main analysis process is the evaluation of different possible 24-bit values accessed from the LUTs 1 and 2 for each value of the data choice in the segments satisfying Equation (1). This test is performed simultaneously on both the low and high segments. The considered 24-bit values in the low and high segments should have the same byte of $sreg[14]$. This key condition on the selection of 24-bit values results in a shorter list of 24-bit vectors evaluating Equation (1) acquiring the offsets $\mathit{offset\_1}_{14}$ and $\mathit{offset\_2}_{14}$. Our experiments show that this equality condition on the register bytes results in around a list of 2^{24} values performing the equation (1) which is a 50% reduction from the original set of $2 * 2^{24}$ values (considering two segments of CAVE). The previous offset bytes $\mathit{offset\_1}_{14prev}$ and $\mathit{offset\_2}_{14prev}$ that resulted in the chosen offsets $\mathit{offset\_1}_{14}$ and $\mathit{offset\_2}_{14}$ can be extracted from the assumed 24-bit values of low and high CAVE segments.

Step 2. The previous offsets are used to calculate the corresponding temp nibbles as follows:
$CT\_low[offset\_1_{14prev}] = temp\_low_{prev}$
$CT\_high[offset\_1_{14prev}] = temp\_high_{prev}$
The *temp_prev* is calculated by concatenating these temp nibbles. The validity of our guess on the 24-bit data obtained from the LUTs is checked using the "sanity check" equation:

$$temp\_13 = ereg[13] \oplus sreg[14].$$

The sanity check equation, in general is represented as:

$$temp\_i = ereg[i] \oplus sreg[i+1].$$

Our experiments show that during this step the lists get reduced to about 2^{16} values which is a significant 99.8 % reduction from the original pre-computed list of $2*2^{24}$ values.

Step 3 The above steps are repeated for the last two adjacent phases of the last round to get the reduced lists of each phase. The lists are checked for compatibility using the property of correlation between LFSR bytes as described in Section 3 and also the use of final offsets of one phase as the starting offset values in the following phase. Our experiments show that backward reconstruction of the four segments of two phases is enough to establish the values of about *half* the bits of the LFSR and the two offsets used in those particular phases! The attack then proceeds backwards on other phases with much more known information which reduces the complexity for these subsequent iterations. In this process the lists will be reduced until finally ending up with a set of valid data used at the start of Round 3.

5 Complexity Analysis

To assess the complexity of this attack in a way that can be compared with the previous attack, we calculate the theoretical complexity of each step using units equivalent to (or less than) evaluating a complete phase of CAVE, and recalling that there are 16 phases in every round of CAVE. Since list processing with pre-computed LUTs is less complex than executing a phase of CAVE, we may develop an upper bound for the complexity of our attack using the phase-equivalent complexity as the fundamental unit. Step 1 has complexity less than 2^{24} of these units, for each of the 2 phases in each of 2 adjacent segments making a total effort of 2^{26}. Step 2 requires around 2^{25} effort for each of the 2 phases, so that it makes 2^{26} effort as well, for a running total of 2^{27} phase-equivalent units. For the first time only, Step 3 must consider all pairings from two segment lists each of size 2^{16} elements, for a total complexity of 2^{32} operations. This dominates the complexity from the first two steps, so we may safely upper bound the complexity of finding all data consistent across two consecutive phases as being clearly less than 2^{33} phase-units. Lists become size of 2^{24}, so combining them costs 2^{48} effort. We use this as an upper bound on complexity for each phase in this attack (64 phases in CAVE-4). As all these calculations must be performed 2^{24} times (with different initial choices for the pair of offset bytes and the start registers in the Init stage), so we expect the effort to find all valid data for 4-Round CAVE to be less than $2^{48} * 2^6 * 2^{24} = 2^{78}$ phase-units (which is about 2^{72} calculations of 4-Round CAVE which has 64 phases). This compares favourably with the 2^{91} effort required by the previous attack [3]. The effort to extend this attack to 8-Round CAVE is minor: only another $2^{48} * 2^6$ effort for each of the 2^{24} trials is an extra 2^{78} phase-units or double the effort above what was needed to break 4-Round CAVE. To compare, the previous attack requires eight times the effort.

We summarize the advantages of the improved attack over the previous attack:

1. Efficient pre-computation analyses the S-boxes or look-up tables backwards and surrounding operations creating lists of possible datasets. Consequently using many look-up tables is much faster than progressively calculating the data.
2. The new attack exploits more information/weaknesses than the previous attack did.
3. The new attack manages an efficient time/memory trade off as it collects lists of all possible data.
4. The complexity of the current attack is much less than that of the previous attack in the task of finding all possible valid inputs.

6 Conclusion

Our improved attack on CAVE may threaten the security of real CAVE implementations. Authenticating a legitimate subscriber is the main application of CAVE (Section 1). If the different input values that hash to a given digest are found, it is possible to illegally program ESN and MIN into the mobile phone thereby providing a fraudulent customer with an access to the wireless network. When the authentication fails, subscriber calls to the network would not be protected even by voice encryption.

The decision to replace CAVE with Authenticated Key Agreement (AKA) was made in 1999 [4]. The slow standardization process, added to that slower adoption by the operators is delaying its replacement. Considering the threats we strongly recommend that where CAVE is still in use, it should be replaced with AKA as soon as possible.

References

1. Telecommunications Industry Association. Appendix A to IS-54 Rev. B Dual-Mode Cellular System: Authentication, Message Encryption, Voice Privacy Mask Generation, Shared Secret Data Generation, A-Key Verification and Test Data, February 1992. This document may be found at `http://www.tcs.hut.fi/~helger/crypto/link/practice/mobile.html`.
2. D.Park, M.N.Oh, and M.Looi. A fraud detection method using IS41C protocols and its applications to the third generation wireless systems. In *IEEE Globecom 98*, volume 4, pages 1984–1989, 1998.
3. William Millan. Cryptanalysis of the alleged CAVE algorithm. In *The 1st International Conference on Information Security and Cryptology*, volume 1, pages 107–119. Korea Institute of Information Security and Cryptology (KIISC), 18-19 December 1998.
4. Greg Rose. Personal Communication, June 2004.
5. Wagner, Simpson, Dawson, Kelsey, Millan, and Schneier. Cryptanalysis of ORYX. In *SAC: Annual International Workshop on Selected Areas in Cryptography*. Lecture Notes in Computer Science, 1998.
6. David Wagner, Bruce Schneier, and John Kelsey. Cryptanalysis of the Cellular Message Encryption Algorithm. *Lecture Notes in Computer Science*, 1294:526–537, 1997.
7. Christopher Wingert and Mullaguru Naidu. CDMA 1XRTT SECURITY OVERVIEW, August 2002. This report is available at `http://www.telecom.co.nz/binarys/cdma_security_overview.pdf`.