Extended version of the paper that will appear under the same title in the proceedings of SAC'03,
Lecture Notes in Computer Science, Springer-Verlag, 2003.

Cryptanalysis of the Alleged SecurID Hash Function (extended version)*

Alex Biryukov**, Joseph Lano*** and Bart Preneel

Katholieke Universiteit Leuven, Dept. Elect. Eng.-ESAT/SCD-COSIC,
Kasteelpark Arenberg 10,
B-3001 Leuven-Heverlee, Belgium
{alex.biryukov,joseph.lano,bart.preneel}@esat.kuleuven.ac.be

Abstract. The SecurID hash function is used for authenticating users to a corporate computer infrastructure. We analyse an alleged implementation of this hash function. The block cipher at the heart of the function can be broken in few milliseconds on a PC with 70 adaptively chosen plaintexts. The 64-bit secret key of 10% of the cards can be discovered given two months of token outputs and 2^{48} analysis steps. A larger fraction of cards can be covered given more observation time.
Keywords: Alleged SecurID Hash Function, Differential Cryptanalysis, Internal Collision, Vanishing Differential.

1 Introduction

The SecurID authentication infrastructure was developed by SDTI (now RSA Security). A SecurID token is a hand-held hardware device issued to authorized users of a company. Every minute (or every 30 seconds), the device generates a new pseudo-random token code. By combining this code with his PIN, the user can gain access to the computer infrastructure of his company. Software-based tokens also exist, which can be used on a PC, a mobile phone or a PDA.

More than 12 million employees in more than 8000 companies worldwide use SecurID tokens. Institutions that use SecurID include the White House, the U.S. Senate, NSA, CIA, The U.S. Department of Defense, and a majority of the Fortune 100 companies.

The core of the authenticator is the proprietary SecurID hash function, developed by John Brainard in 1985. This function takes as an input the 64-bit

* The work described in this paper has been supported by the Concerted Research Action (GOA) Mefisto.
** F.W.O. researcher, sponsored by the Fund for Scientific Research – Flanders
*** Research financed by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

secret key, unique to one single authenticator, and the current time (expressed in seconds since 1986). It generates an output by computing a pseudo-random function on these two input values.

SDTI/RSA has never made this function public. However, according to V. McLellan [5], the author of the SecurID FAQ, SDTI/RSA has always stated that the secrecy of the SecurID hash function is not critical to its security. The hash function has undergone cryptanalysis by academics and customers under a non-disclosure agreement, and has been added to the NIST "evaluated products"-list, which implies that it is approved for U.S. government use and recommended as cryptographically sound to industry.

The source code for an alleged SecurID token emulator was posted on the Internet in December 2000 [10]. This code was obtained by reverse engineering, and the correctness of this code was confirmed in some reactions on this post. From now on, this implementation will be called the "Alleged SecurID Hash Function" (ASHF). Although the source code of the ASHF has been available for more than two years, no article discussing this function has appeared in the cryptographic literature, except for a brief note [6]. This may be explained by a huge number of subrounds (256) and by the fact that the function looses information and thus standard techniques for analysis of block-ciphers may not be applicable.

In this paper, we will analyse the ASHF and show that the block-cipher at the core of this hash function is very weak. In particular this function is non-bijective and allows for an easy construction of *vanishing* differentials (*i.e.* collisions) which leak information on the secret key. The most powerful attacks that we show are differential [1] adaptive chosen plaintext attacks which result in a complete secret key recovery in just a few milliseconds on a PC (See Table 1 for a summary of attacks). For the real token, vanishing differentials do occur for 10% of all keys given only two months of token output and for 35% of all keys given a year of token output. We describe an algorithm that recovers the secret key given one single vanishing differential. The complexity of this algorithm is 2^{48} SecurID encryptions. These attacks motivate a replacement of ASHF by a more conventional hash function, for example one based on the AES.

In Sect. 2, we give a detailed description of the ASHF. In Sect. 3, we show attacks on the block cipher-like structure, the heart of the ASHF. In Sect. 4, we show how to use these weaknesses for the cryptanalysis of the full ASHF construction. Sect. 5 presents the conclusions.

2 Description of the Alleged SecurID Hash Function

The alleged SecurID hash function (ASHF) outputs a 6 to 8-digit word every minute (or 30 seconds). All complexity estimates further in this paper, if not specified otherwise, will correspond to 60-second token described in [10] though we expect that our results apply to 30-second tokens with twice shorter observation period. The design of ASHF is shown in Fig. 1. The secret information contained in ASHF is a 64-bit secret key. This key is stored securely and remains

Table 1. The attacks presented in this paper.

Function attacked	Type of Attack	Data Compl. (#texts**)	Time Compl.**	Result	Sect.
Block-cipher (256 subrounds)	Adaptively chosen plaintext	70	2^{10}	full key recovery	3.3
Full ASHF	Adaptively chosen plaintext	2^{17}	2^{17}	few key bits	4.1
Full ASHF	Chosen + Adaptively chosen plaintext	$2^{24} + 2^{13}$	2^{24}	full key recovery	4.1
Full ASHF+ time duplication	Known plaintext⋆	$2^{16} - 2^{18}$	2^{48}	full key recovery, for $10 - 35\%$ of the keys	4.2
Full ASHF+ time duplication	Known plaintext* + Network monitoring	2^{18}	2^{18}	Possible network intrusion	4.2

⋆ A full output of the 60-second token for 2-12 months.
* A full output of the 60-second token for one year.
** Data complexity is measured in full hash outputs which would correspond to two consecutive outputs of the token.
** Time complexity is measured in equivalent SecurID encryptions.

the same during the lifetime of the authenticator. Every 30 or 60 seconds, the time is read from the clock. This time is then manipulated by the expansion function (which is not key-dependent), to yield a 64-bit time.

This 64-bit time is the input to the key-dependent transformation, which consists of an initial permutation, four rounds of a block cipher-like function, a final permutation and a conversion to decimal. After every round the key is changed by XORing the key with the output of that round, hence providing a simple key scheduling algorithm. The different building blocks of the hash function will now be discussed in more detail.

In the following, we will denote a 64-bit word by b, consisting of bytes $B_0, B_1, \ldots B_7$, of nibbles $\mathrm{B}_0\mathrm{B}_1 \ldots \mathrm{B}_{15}$ and of bits $b_0 b_1 \ldots b_{63}$. B_0 is the Most Significant Byte (MSB) of the word, b_0 is the most significant bit (msb).

2.1 The Expansion Function

We consider the case where the token code changes every 60 seconds. The time is read from the clock into a 32-bit number, and converted first into the number of minutes since January 1, 1986, 0.00 GMT. Then this number is shifted to the left one position and the two least significant bits are set to zero. We now have a 32-bit number $B_0B_1B_2B_3$. This is converted into the 64-bit number by repeating the three least significant bytes several times. The final time t is then of the form $B_1B_2B_3B_3B_1B_2B_3B_3$. As the two least significant bits of B_3 are always zero, this also applies to some bits of t. The time t will only repeat after

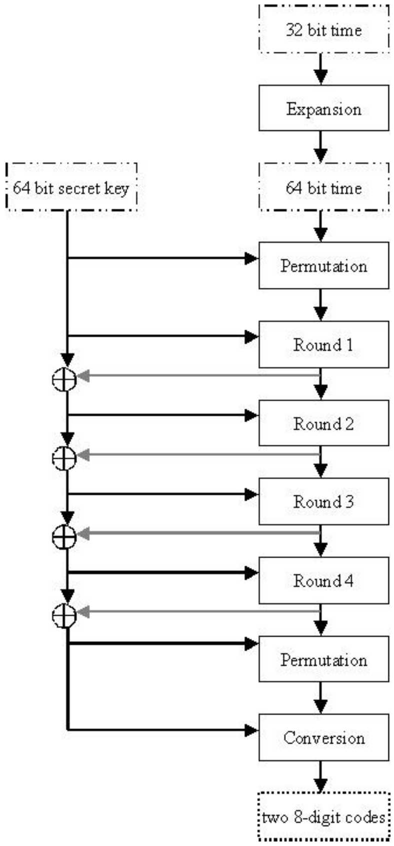

Fig. 1. Overview of the ASHF hash function

2^{23} minutes (about 16 years), which is clearly longer than the lifetime of the token.

One can notice that the time only changes every two minutes, due to the fact that the last two bits are set to zero during the conversion. In order to get a different one-time password every minute, the ASHF will at the end take the 8 (or 6) least significant digits at even minutes, and the 8 (or 6) next digits at odd minutes. This may have been done to economize on computing power, but on the other hand this also means that the hash function outputs more information. From an information-theoretic point of view, the ASHF gives 53 bits (40 bits) of information if the attacker can obtain two consecutive 8 digit (6 digit) outputs. This is quite high considering the 64-bit secret key.

2.2 The Key-Dependent Initial and Final Permutation

The key-dependent bit permutation is applied before the core rounds of the hash function and after them. It takes a 64-bit value u and a key k as input and outputs a permutation of u, called $y = P(u, k)$.

This is how the permutation works: The key is split into 16 nibbles $\mathrm{K}_0\mathrm{K}_1 \ldots \mathrm{K}_{15}$, and the 64 bits of u are put into an array. A pointer p jumps to bit u_{K_0}. The four bits strictly before p will become the output bits $y_{60}y_{61}y_{62}y_{63}$.

The bits that have already been picked are removed from the array, and the pointer is increased (modulo the size of the array) by the next key nibble K_1. Again the four bits strictly before p become the output bits $y_{56}y_{57}y_{58}y_{59}$. This process is repeated until the 16 nibbles of the key are used up and the whole output word y is determined.

An interesting observation is that the 2^{64} keys only lead to $2^{60.58}$ possible permutations. This can be easily seen: when the array is reduced to 12 values, the key nibble can only lead to 12 different outputs, and likewise for an array of length 8 or 4.[1]

This permutation is not strong: given one input-output pair (u, y), one gets a lot of information on the key. An algorithm has been written that searches all remaining keys given one pair (u, y). On average, 2^{12} keys remain, which is a significant reduction. Given two input-output pairs, the average number of remaining keys is 17.

In Sect. 3 and 4, we will express the complexity of attacks in the number of SecurID encryptions. In terms of processing time, one permutation is equivalent to 5% of a SecurID encryption.

2.3 The Key-Dependent Round

One round takes as input the 64-bit key k and a 64-bit value b^0. It outputs a 64-bit value b^{64}, and the key k is transformed to $k = k \oplus b^{64}$. This ensures that every round (and also the final permutation) gets a different key. The round consists of 64 subrounds, in each of these one bit of the key is used. The superscript ^{i} denotes the word after the i-th round. Subround i ($i = 1 \ldots 64$) transforms b^{i-1} into b^i and works as follows (see Fig. 2):

1. Check if the key bit k_{i-1} equals b_0^{i-1}.
   – if the answer is yes:

$$\begin{cases} B_0^i = ((((B_0^{i-1} \ggg 1) - 1) \ggg 1) - 1) \oplus B_4^{i-1} \\ B_j^i = B_j^{i-1} \text{ for } j = 1, 2, 3, 4, 5, 6, 7 \end{cases} \tag{1}$$

   where $\ggg i$ denotes a cyclic shift to the right by i positions and $\oplus$ denotes an exclusive or. This function will be called R.

[1] The description of the permutation given here is more transparent than the actual implementation. This equivalent representation and the observation on the number of permutations has also been found independently by Contini [2].

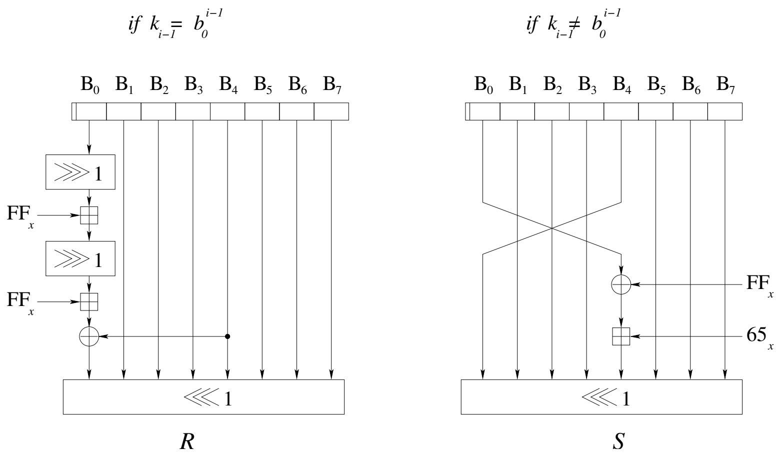

Fig. 2. Sub-round i defined by the operations R and S.

– if the answer is no:

$$\begin{cases} B_0^i = B_4^{i-1} \\ B_4^i = 100 - B_0^{i-1} \mod 256 \\ B_j^i = B_j^{i-1} \text{ for } j = 1,2,3,5,6,7 \end{cases} \tag{2}$$

This function will be called S.

2. Cyclically shift all bits one position to the left:

$$b^i = b^i \lll 1\,. \tag{3}$$

After these 64 iterations, the output b^{64} of the round is found, and the key is changed according to $k = k \oplus b^{64}$.

In terms of processing time, one round is equivalent to 0.21 SecurID encryptions.

2.4 The Key-Dependent Conversion to Decimal

After the initial permutation, the 4 rounds and the final permutation, the 16 nibbles of the data are converted into 16 decimal symbols. These will be used to form the two next 8 (or 6)-digit codes.

This is done in a very straightforward way: the nibbles $0_x, 1_x, \ldots 9_x$ are simply transformed into the digits $0, 1, \ldots 9$. A_x, C_x and E_x are transformed into $0, 2, 4, 6$ or 8, depending on the key; and B_x, D_x and F_x are transformed into $1, 3, 5, 7$ or 9, also depending on the key.

3 Cryptanalysis of the Key-Dependent Rounds

The four key-dependent rounds are the major factor for mixing the secret key and the current time. The ASHF could have used any block cipher here (for example DES), but a dedicated design has been used instead. The designers had the advantage that the rounds do not have to be reversible, as the cipher is only used to encrypt. In this section we will show some weaknesses of this dedicated design.

3.1 Avalanche of 1-Bit Differences

It is instructive to look at how a differential propagates through the rounds. As has been explained in Sect. 2.3, in every subround one bit of the key is mixed in by first performing one R or S-operation and then shifting the word one position to the left.

A standard differential trail, for an input difference in b_{63}, is shown in Table 2 in Appendix B. As long as the difference does not enter B_0 or B_4, there is no avalanche at all: the difference is just shifted one position to the left after each subround. But once the difference enters into B_0 and B_4, avalanche occurs quite rapidly. This is due to two factors: S and R are nonlinear (though very close to linear) and mainly due to a choice between the two operations S or R, conditional on the most significant bit b_0 of the text.

Another complication is that, unlike in regular block ciphers, the round output is XORed into the subkey of the following round. This leads to difference propagation into the subkeys which makes the mixing even more complex. Because of the many subrounds (256), it seems that avalanche will be more than sufficient to resist differential cryptanalysis.

3.2 Differential Characteristics

We now search for interesting differential characteristics. Two texts are encrypted and the differential trail is inspected. A difference in the lsb of B_7 will be denoted by 1_x, in the second bit of B_7 by 2_x, etc… Three scenarios can occur in a subround:

- If the difference is not in B_0 or B_4, it will undergo a linear operation: the difference will always propagate by shifting one position to the left: 1_x will become 2_x, 2_x will become 4_x, etc.
- If the difference is in B_0 and/or B_4, and the difference in the msb of B_0 equals the difference in the key, both texts will undergo the same operation (R or S). These operations are not completely linear, but a difference can only go to a few specific differences. A table has been built that shows for every differential the differentials it can propagate to together with the probability.
- If the difference is in B_0 and/or B_4, and the difference in the msb of B_0 does not equal the difference in the key, one text will undergo the R operation and the other text will undergo the S operation. This results in a uniform distribution for the output differences: each difference has probability 2^{-16}.

Our approach has been to use an input differential in the least significant bits of B_3 and B_7. This differential will then propagate "for free" for about

24 subrounds. Then it will enter B_0 and B_4. We will pay a price in the second scenario (provided the msb of the differential in B_0 is not 1) for a few subrounds, after which we want the differential to leave B_0 and B_4 and jump to B_7 or B_3. Then it can again propagate for free for 24 rounds, and so on.

That way we want to have a difference in only 1 bit after 64 subrounds. This is the only difference which will propagate into the key, and so we can hope to push the differentials through more rounds.

The best differential for one round that we have found has an input difference of 2_x in B_3 and 5_x in B_7, and gives an output difference of 1_x in B_6 with probability 2^{-15}. As an illustration, such a differential trail is shown in Table 3 in Appendix B.

It is possible that better differentials exist. One could try to push the characteristic through more rounds, but because of the difference in the keys other paths will have to be searched. For now, it is unclear whether a path through the four rounds can be found with probability larger than 2^{-64}, but it may well be feasible. In any case, this approach is far less efficient than the approach we will discuss in the following section.

3.3 Vanishing Differentials

Because of the specific application in which the ASHF is used, the rounds do not have to be bijections. A consequence of this non-bijective nature is that it is possible to choose an input differential to a round, such that the output differential is zero, *i.e.*, an internal collision. An input differential that causes an internal collision will be called a vanishing differential[2].

This can be derived as follows. Suppose that we have two words at the input of the i-th subround, b^{i-1} and b'^{i-1}. A first condition is that:

$$b_0^{i-1} \neq b_0'^{i-1}. \tag{4}$$

Then one word, assume b^{i-1}, will undergo the R operation, and the other word b'^{i-1} will undergo the S operation. We impose a second condition, namely:

$$B_j^{i-1} = B_j'^{i-1} \text{ for } j = 1, 2, 3, 5, 6, 7. \tag{5}$$

We now want to find an input differential in B_0 and B_4 that vanishes. This is easily achieved by simply setting b^i equal to b'^i after S or R:

$$\begin{cases} B_0^i = B_0'^i \Rightarrow B_4'^{i-1} = ((((B_0^{i-1} \ggg 1) - 1) \ggg 1) - 1) \oplus B_4^{i-1} \\ B_4^i = B_4'^i \Rightarrow B_0'^{i-1} = 100 - B_4^{i-1} \,, \end{cases} \tag{6}$$

As both R and S are bijections, there will be 2^{16} such tuples (B_0^{i-1}, B_4^{i-1}, $B_0'^{i-1}$, $B_4'^{i-1}$). The extra condition that the msb of B_0^{i-1} and $B_0'^{i-1}$ have to be different will filter out half of these, still leaving 2^{15} such pairs.

[2] Notice that the ASHF scheme can be viewed both as a block cipher with partially non-bijective round function or as a MAC where current time is authenticated with the secret key. The fact that internal collisions in MACs may lead to forgery or key-recovery attacks was noted and exploited against popular MACs in [7, 8, 4].

This observation leads to a very easy attack on the full block cipher. We choose two plaintexts b^0 and b'^0 that satisfy the above conditions. We encrypt these with the block cipher (the 4 rounds), and look at the difference between the outputs. Two scenarios are possible:

- b^0 undergoes the R operation and b'^0 undergoes the S operation in the first subround: this means that b^1 will be equal to b'^1. As the difference is equal to zero, this will of course propagate to the end and we will observe this at the output. This also implies that $k_0 = b^0_0$, because that is the condition for b^0 to undergo the R operation.
- b^0 undergoes the S operation and b'^0 undergoes the R operation in the first subround: this means that b^1 will be different from b'^1. This difference will propagate further (with very high probability), and we will observe a nonzero output difference. This also implies that $k_0 = b'^0_0$, because that is the condition for b'^0 to undergo the R operation.

To summarize, this gives us a very easy way to find the first bit of the secret key with two chosen plaintexts: choose b^0 and b'^0 according to (4), (5) and (6) and encrypt these. If outputs will be equal, then $k_0 = b^0_0$. If not, then $k_0 = b'^0_0$!

This approach can be easily extended to find the second, third, ... bit of the secret key by working iteratively. Suppose we have determined the first $i-1$ bits of the key. Then we can easily find a pair (b^0, b'^0), that will be transformed by the already known key bits of the first $i-1$ subrounds into a pair (b^{i-1}, b'^{i-1}) that satisfies (4), (5) and (6). The search for an adequate pair is performed offline, and once we have found a good pair offline it will always be suitable for testing on the real cipher. The procedure we do is to encrypt a random text with the key bits we have deduced so far, then we choose the corresponding second text there for a vanishing differential, and for this text we then try to find a preimage. Correct preimages can be found with high probabilities (75% for the 1st bit, 1.3% for 64th bit), hence few text pairs need to be tried before an adequate pair is found. Now we can deduce k_i in the same way we deduced k_0 above.

This algorithm has been implemented in C on a Pentium. Finding the whole secret key requires at most 128 adaptively chosen plaintexts. This number will in practice mostly be reduced, because we can often reuse texts from previous subrounds in the new rounds. Simulation shows that on average only 70 adaptively chosen plaintexts are needed. The complexity of this attack can be calculated to be 2^{10} SecurID encryptions. Finding the 64-bit secret key for the whole block cipher, with the search for the appropriate plaintexts included, requires only a few milliseconds. Of course, it is possible to perform a trade-off by only searching the first i key bits. The remaining $64-i$ key bits can then be found by doing a reduced exhaustive search.

4 Extending the Attack with Vanishing Differentials to the Whole ASHF

So far, we have shown how to break the four key-dependent rounds of the ASHF. In this paragraph, we will first try to extend the attack of Sect. 3.3 to the full

ASHF, which consists of a weak initial key-dependent permutation, the four key-dependent rounds, a weak final key-dependent permutation and the key-dependent conversion to decimal. Then we will mount the attack when taking the symmetrical time format into account.

4.1 The Attack on the Full ASHF

The attack with vanishing differentials of Sect. 3.3 is not defeated by the final permutation. Even a perfect key-dependent permutation would not help. The only thing an attacker needs to know is whether the outputs are equal, and this is not changed by the permutation. The lossy conversion to decimal also doesn't defeat the attack. There still are 53 bits (40 bits) of information, so the probability that two equal outputs are caused by chance (and thus are not caused by a vanishing differential) remains negligibly small.

However, the initial key-dependent permutation causes a problem. It prevents the attacker from choosing the inputs to the block cipher rounds. A new approach is needed to extract information from the occurrence of vanishing differentials.

The attack starts by searching a vanishing differential by randomly choosing 2 inputs that differ in 1 bit, say in bit i. Approximately 2^{17} pairs need to be tested before such a pair can be found. Information on the initial permutation (and thus on the secret key) can then be learned by flipping a bit j in the pair and checking whether a vanishing differential is still observed. If this is so, this is an indication that bit j is further away from B_0 and B_4 than bit i.

An algorithm has been implemented that searches for a vanishing pair with one bit difference in all bits i, and then flips a bit for all bits j. This requires 2^{24} chosen and 2^{13} adaptively chosen plaintexts. A $64 \cdot 64$ matrix is obtained that carries a lot of information on the initial permutation. Every element of the matrix establishes an order relationship, every row i shows which bits j can be flipped for an input differential in bit i, and every column shows how often flipping bit i will preserve the vanishing differential. This information can be used to extract the secret key. The first nibble of the key determines which bits go to positions $b_{60}, \ldots b_{63}$. By inspecting the matrix, one can see that this should be the four successive bits between bit 60 and bit 11 that have the lowest row weight and the highest column weight.

Once we have determined the first nibble, we can work recursively to determine the following nibbles of the secret key. The matrix will not always give a unique solution, so then it will be necessary to try all likely scenarios. Simulation indicates that the uncertainty seems to be lower than the work required to find the vanishing differentials.

The data complexity of this attack may be further reduced by using fewer vanishing differentials, but thereby introducing more uncertainty.

4.2 The Attack on the Full ASHF with the Time Format

The attack is further complicated by the symmetrical time format. A one-bit difference in the 32-bit time is expanded into a 2-bit difference (if the one-bit difference is in the second or third byte of the 32-bit time) or into a 4-bit difference (if the one-bit difference is in least significant byte of the 32-bit time).

A 4-bit difference caused by a one-bit difference in the 32-bit time will yield a vanishing differential with negligible probability, because the initial permutation will seldom put the differences in an ordered way that permits a vanishing differential to occur. A 2-bit difference, however, will lead to a vanishing differential with high probability: taking a random key, an attacker observes two equal outputs for one given 2-bit difference with probability 2^{-19}. This can be used to distinguish the SecurID stream from a random stream, as two equal outputs would normally only be detected with probability 2^{-53} ($2^{-26.5}$ if only one 8-digit tokencode is observed).

In the following, we assume that an attacker can observe the output of the 60-second token during some time period. This attack could simulate a "curious user" who wants to know the secret key of his own card, or a malicious internal user who has access to the SecurID cards before they are deployed. The attacker could put his token(s) in front of a PC camera equipped with OCR software, automating the process of reading the numbers from the card. The attacker then would try to detect vanishing differentials in the sampled data. Figure 3 shows the percentage of the tokens for which at least one vanishing differential is detected, as a function of the observation time.

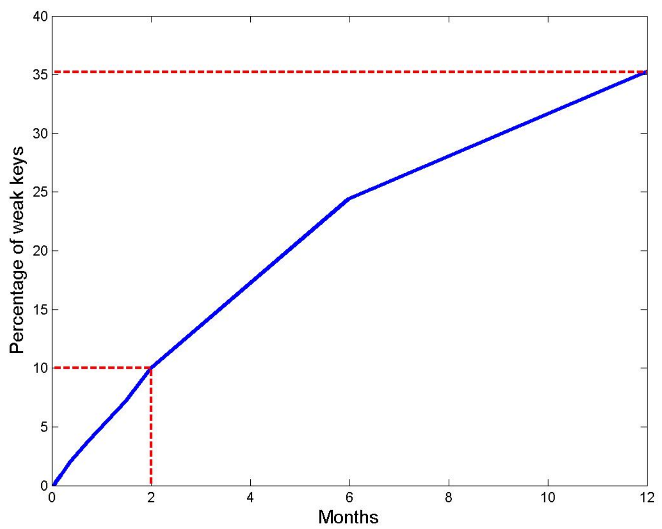

Fig. 3. Percentage of keys that have at least one VD

Two interesting points are the following: after two months, about 10% of the keys have one or more vanishing differentials; after one year, 35% of keys have at least one vanishing differential. Below, we show two ways in which an attacker can use the information obtained by the observation of the token output.

Full key recovery. In this attack, we recover the key from the observation of a single vanishing differential. In the following, it is assumed that we are given one pair of 64-bit time inputs (t, t') that leads to a vanishing differential, and that there is a 2-bit difference between the two inputs (the attack can also be mounted with other differences). We know that the known pair (t, t') will be permuted into an unknown pair (b, b'), and that the difference will vanish after an unknown number of subrounds N. The outputs of the hash, o and o', are also known and are of course equal.

The idea of the attack is to first guess the subround N in which the vanishing differential occurs, for increasing $N = 1, 2, \ldots$ For each value of N, a filtering algorithm is used that searches the set of keys that make such a vanishing differential possible. We then search this set to find the correct key. We will now describe the filtering algorithm for the case $N = 1$. For other values of N the algorithm will be similar.

When guessing $N = 1$, we assume the vanishing differential to happen in the first subround. This means that the only bits that play a role in the vanishing differential are the key bit k_0, and the bytes B_0, B_4, B'_0 and B'_4 of the two words after the permutation. Out of the 2^{33} possible values, we are only interested in those combinations that have a 2-bit difference between the texts and lead to a vanishing differential. We construct a list of these values during a precomputation phase, which has to be performed only once for all attacks. In the precomputation phase, we let a guess of the word b, called b_v, take on all values in bytes B_{v0} and B_{v4} (the other values are set to a fixed value), and let k_0 be 0 or 1. We perform one subround for all these values and store the outputs. For each value of k_0, we then search for equal outputs for which the inputs b_v, b'_v have 2-bit difference. We then have found a valid combination, which we store as a triplet (k_0, b_v, b'_v). Note that in b_v and b'_v only two bytes are of importance, we refer to such words as "partially filled". Only 30 such valid triplets are found.

The list of scenarios found in the precomputation phase will be used as the start for a number of guessing and filtering steps. In the first step, for each scenario (k_0, b_v, b'_v), we try all possible values of $k_1, \ldots k_{27}$. Having guessed the first 28 key bits, we know the permutation of the last 28 bits. We call these partially guessed values $b_p = P(t, k)$ and $b'_p = P(t', k)$. Under this scenario, we now have two partial guesses for both b and b', namely b_v, b_p and b'_v, b'_p. One can see that b_v and b_p overlap in the nibble B_9. If $\mathrm{B}_{v9} = \mathrm{B}_{p9}$ and $\mathrm{B'}_{v9} = \mathrm{B'}_{p9}$, the scenario $(k_0 \ldots k_{27}, b_v, b'_v)$ is possible and will go to the next iteration. If there's a contradiction, the scenario is impossible and can be filtered out.

In the second step, a scenario that survived the first step is taken as input and the key bits $k_{28} \ldots k_{31}$ are guessed. Filtering is done based on the overlap between the two guesses for nibble B_8. In the third step we guess key bits $k_{32} \ldots k_{59}$ and filter for the overlap in nibble B_1. Finally, in the fourth step the key bits $k_{60} \ldots k_{63}$ are guessed and the filtering is based on the overlap in nibble B_0. Every output of this fourth step is a key value that leads to a vanishing differential in the first subround for the given inputs t and t'. Each key that survives the filtering algorithm will then be checked to see if it is the correct key.

The time complexity of the attack itself is dependent on the strength of the filtering. This may vary from input to input, but simulations indicate the following averages for $N = 1$: the 30 scenarios go to 2^{27} possibilities after the first step. These are reduced to 2^{25} after the second step, go to 2^{45} after the third step and reduce to 2^{41} after the fourth step. The most expensive step in this algorithm is the third step: 28 bits of the key are guessed in 2^{25} scenarios. At every iteration, two partial permutations (28 bits in each) have to be done. This means that the total complexity in terms of SecurID encryptions of this step is:

$$2^{25} \cdot 2^{28} \cdot \frac{28}{64} \cdot 2 \cdot 0.05 \approx 2^{48.5} , \tag{7}$$

where the factor 0.05 is the relative complexity of the permutation.

The time complexity of the third (and forth) step can be further improved by using the fact that there are only $2^{60.58}$ possible permutations, as described in Sect. 2.2. By searching for the permutations instead of the key, we reduce the complexity of the attack by a factor $\frac{12}{16} \cdot \frac{8}{16}$ to 2^{47}. The memory requirements, apart from the storage of the precomputed tables, are negligible.

The total attack consists of performing the above algorithm for $N = 1, 2, \ldots$ until we have a success probability of 50%. Simulation on 10000 samples indicated that more than 50% of the vanishing differentials (observed in two-month periods) occur before the 12th subround. So it suffices to run the attack for N up to 12.

One can see that the complexity of the attack will be lower for $N \geq 2$ than for $N = 1$. Let's take the example of $N = 2$. Now the only bits that play a role in the vanishing differential are the key bits k_0 and k_1, and the bits $b_0 \ldots b_8$ and $b_{32} \ldots b_{40}$. During the precomputation phase, we construct a list for these 20 bits. Only 350 possibilities remain. The complexity of the attack is now reduced for two reasons. First, the overlap in the filtering algorithms is now 18 bits in total instead of 16 bits for $N = 1$. Our filtering will thus be stronger. Second, we can now do six filtering steps instead of four filtering steps. For instance, in the first step we will only have to guess $k_1 \ldots k_{23}$ to have a first overlap. Because of this, we will see that the complexity of the algorithm decreases with increasing N. The total complexity of the attack is the sum of the complexities of the steps for $N = 1, 2, \ldots$. It can be calculated that the time complexity is equivalent to 2^{48} *SecurID encryptions*. We expect that further refinements of the filtering algorithm are possible which reduce to 2^{45} the time complexity of the attack .[3]

Another factor in the attack is the complexity of the precomputation step. The time complexity of the precomputation step increases with N. One can see that for a vanishing differential in subround N, $2 \cdot N + 14$ bits of the text

[3] Indeed, after publication of this paper, Contini and Yin published a nice follow-up[3] which proves that this estimate is correct. They improved the time complexity of the attack to 2^{44}, and show further refinements which improve the complexity to 2^{40}.

and N bits of the key are of importance. It can be calculated that the total time complexity to precompute the tables for N up to 12 will be 2^{44} SecurID encryptions. The size of the precomputed tables is increasing with N. Our results show that the size increases with N by a factor of 6 to 8. Our conservative estimate is that about 500 GB of hard disk is required to store the tables for N up to 12.

Network intrusion. An attacker can also use the one year output without trying to recover the secret key of the token. In this attack scenario it is assumed that the attacker can monitor the traffic on the network in the following year. When an attacker monitors a token at the (even) time t, he will compare this with the value of the token 2^{19} minutes (about 1 year) earlier. If both values are equal, this implies that a vanishing differential is occurring. This vanishing differential will also hold at the next (odd) time $t+1$. The attacker can thus predict at time t the value of the token at time $t+1$ by looking it up in his database, and can use this value to log into the system.

Simulation shows that 4% of the keys [4] will have such vanishing differentials within a year, and that these keys will have 6 occurrences of vanishing differentials on average. Higher percentages of such weak keys are possible if we assume the attacker is monitoring several years.

The total probability of success of an attacker depends on the number of cards from which he could collect the output during one year, and on the number of logins of the users in the next year. It is clear that the attack will not be very practical, since the success probability for a single user with 1000 login attempts per year will be $2^{-19} \cdot 1000 \cdot 6 \cdot 0.04/2 \approx 2^{-12}$. This is however, much higher than what would be expected from the password length (*i.e.*, $2^{-26.5}$ and 2^{-20} for 8- and 6-digit tokens respectively). Note that this attack does not produce failed login attempts and requires only passive monitoring.

5 Conclusion

In this paper, we have performed a cryptanalysis of the Alleged SecurID Hash Function (ASHF). It has been shown that the core of this hash, the four key-dependent block cipher rounds, is very weak and can be broken in a few milliseconds with an adaptively chosen plaintext attack, using vanishing differentials.

The full ASHF is also vulnerable: by observing the output during two months, we have a 10% chance to find at least one vanishing differential. In our main attack, we use this vanishing differential to recover the secret key of the token with expected time complexity 2^{48}. In a follow-up on this paper, Contini and Yin [3] confirm this result and improve the complexity of the attack with a factor 16. We can also use the observed output to significantly increase our chances of network intrusion.

[4] Decreased key fraction is due to the restriction of the input difference to a specific one year difference.

The ASHF does not deliver the security that one would expect from a present-day cryptographic algorithm. See Table 1 for an overview of the attacks presented in this paper.

We thus would recommend to replace the ASHF-based tokens by tokens implementing another algorithm that has undergone extensive open review and which has 128-bit internal secret. The AES seems to be a good candidate for such a replacement and indeed, from February 2003 RSA has started phasing in AES-based tokens [9].

References

1. E. Biham, A. Shamir, *Differential Cryptanalysis of DES-like Cryptosystems,* Journal of Cryptology, volume 4, number 1, pp. 3–72, 1991.
2. S. Contini, *The Effect of a Single Vanishing Differential on ASHF,* sci.crypt post, Sept. 6, 2003.
3. S. Contini, Y.L. Yin, *Improved Cryptanalysis of SecurID,* Cryptology ePrint Archive, Report 2003/205, http://eprint.iacr.org/2003/205, September 2003.
4. Internet Security, Applications, Authentication and Cryptography, University of California, Berkeley, *GSM Cloning* http://www.isaac.cs.berkeley.edu/isaac/gsm-faq.html.
5. V. McLellan, *Re: SecurID Token Emulator*, post to BugTraq, http://cert.uni-stuttgart.de/archive/bugtraq/2001/01/msg00090.html
6. Mudge, Kingpin, *Initial Cryptanalysis of the RSA SecurID Algorithm,* www.atstake.com/ research/reports/acrobat/initial_securid_analysis.pdf
7. B. Preneel, P. van Oorschot, *MDx-MAC and Building Fast MACs From Hash Functions*, CRYPTO'95, LNCS 963, D. Coppersmith, Ed., Springer-Verlag, pp. 1–14, 1995.
8. B. Preneel, P. van Oorschot, *On the Security of Two MAC Algorithms,* Eurocrypt 96, LNCS 1070, U. Maurer, Ed., Springer-Verlag, pp. 19–32, 1996.
9. RSA security website, http://www.rsasecurity.com/company/news/releases/pr.asp?doc_id=1543
10. I.C. Wiener, *Sample SecurID Token Emulator with Token Secret Import*, post to BugTraq, http://archives.neohapsis.com/archives/bugtraq/2000-12/0428.html

A A Linear Property

An interesting property of the S and R function is that the msb of B_4, b_{32}, will be biased after applying S or R. Moreover, this bias is dependent on the value of the key bit. This can be seen as follows: suppose that after $i-1$ subrounds, B_0 and B_4 are random. Without taking the rotation to the left into account, these are the possible cases:

- $k_{i-1} = 0$:
  - $b_0^{i-1} = 0$: The word will undergo the R operation. B_4 remains unchanged through the R operation, and the probabilities are $p(b_{32}^i = 0) = 1/2$ and $p(b_{32}^i = 1) = 1/2$.

  - $b_0^{i-1} = 1$: The word will undergo the S operation. B_0^{i-1} is uniformly distributed within the interval $[128,\ 255]$, thus B_4^i is uniformly distributed within the interval $[101,\ 228]$. Hence $p(b_{32}^i = 0) = 101/128$ and $p(b_{32}^i = 1) = 27/128$.
- $k_{i-1} = 1$:
  - $b_0^{i-1} = 1$: The word will undergo the R operation. B_4 remains unchanged through the R operation, and the probabilities are $p(b_{32}^i = 0) = 1/2$ and $p(b_{32}^i = 1) = 1/2$.
  - $b_0^{i-1} = 0$: The word will undergo the S operation. B_0^{i-1} is uniformly distributed within the interval $[0,\ 128]$, thus B_4^i is uniformly distributed within the interval $[0,\ 100] \cup [228,\ 255]$. Hence $p(b_{32}^i = 0) = 27/128$ and $p(b_{32}^i = 1) = 101/128$.

This means that overall, after the R or S function, $p(k_{i-1} = b_{32}^i) \approx 35.55\%$. This property can be used to break one round of the block cipher using ciphertext-only: the bias will simply propagate to the left after the rotation step, and will then remain unchanged until it enters B_0 after 25 more subrounds. The attack scenario is as follows: given 100 ciphertexts, we count the number of times b_7^{64}, $b_8^{64} \ldots b_{31}^{64}$ are zero. It is easy to see that these values correspond to the value of bit b_{32} after mixing in the key bit k_{39}, $k_{40} \ldots k_{63}$ respectively. So if b_7 is equal to 0 more than 50 times, this means that $k_{39} = 1$, and so on for the other values. 25 bits of the key can be determined with a very high probability (when using 100 ciphertexts, the probability of error is only 0.05%). The other 39 key bits can now be determined by an exhaustive search.

This attack would be easily extendible to the four rounds, if the key schedule of ASHF was not data-dependent. The data-dependent key schedule does not allow to add up the occurrences for different ciphertexts, because the key in the final round will be different for every ciphertext. It is possible that the designers were aware of this property and therefore chose the data-dependent key schedule.

However, we can mount the following key-recovery attack on the full four-round function: given a ciphertext, we know that the most probable value for $k_{39} \ldots k_{63}$ of the key in the final round, is the complement of $b_7 \ldots b_{31}$. The probability that this guess for the 25 bits is correct is $(165/256)^{25} \approx 2^{-16}$. With 2^{39} more guesses, we can determine the other bits of the key in the final round. It is then possible, knowing the plaintext-ciphertext pair, to work our way back to the secret key. This attack requires on average 2^{15} plaintexts and corresponding ciphertexts, and 2^{55} steps.

Finding the secret key given the plaintext, the ciphertext and the key in the final round is not straightforward, as the round function is not a bijection. An *ad hoc* algorithm has been implemented to perform this. This means that each step is expensive in terms of CPU usage.

B Differential Trail

Table 2. Unrestricted differential trail for the first 50 subrounds (avalanche demonstration).

Subround	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
0	00	00	00	00	00	00	00	01
1	00	00	00	00	00	00	00	02
2	00	00	00	00	00	00	00	04
3	00	00	00	00	00	00	00	08
4	00	00	00	00	00	00	00	10
5	00	00	00	00	00	00	00	20
6	00	00	00	00	00	00	00	40
7	00	00	00	00	00	00	00	80
...								
24	00	00	00	00	01	00	00	00
25	02	00	00	00	02	00	00	00
26	1a	00	00	00	04	00	00	01
27	08	00	00	00	1c	00	00	02
28	38	00	00	00	70	00	00	04
29	f4	00	00	00	e0	00	00	08
30	68	00	00	01	40	00	00	11
31	4c	00	00	02	80	00	00	22
32	22	00	00	05	00	00	00	45
33	00	00	00	0a	cc	00	00	8a
34	98	00	00	15	98	00	01	15
35	2c	00	00	2b	88	00	02	2a
36	10	00	00	56	28	00	04	55
37	50	00	00	ac	e0	00	08	aa
38	c0	00	01	58	a0	00	11	55
39	c2	00	02	b1	10	00	22	ab
40	0e	00	05	63	ec	00	45	56
41	e4	00	0a	c7	d8	00	8a	ac
42	c2	00	15	8f	3e	01	15	59
43	72	00	2b	1e	50	02	2a	b2
44	a0	00	56	3c	64	04	55	64
45	9c	00	ac	78	9c	08	aa	c9
46	14	01	58	f0	b8	11	55	92
47	4a	02	b1	e1	70	22	ab	25
48	e0	05	63	c2	94	45	56	4a
49	b6	0a	c7	84	0a	8a	ac	94
50	3e	15	8f	08	0b	15	59	29
...								

Table 3. A Differential characteristic with probability 2^{-15} for the first round

Subround	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
0	00	00	00	02	00	00	00	05
1	00	00	00	04	00	00	00	0a
2	00	00	00	08	00	00	00	14
3	00	00	00	10	00	00	00	28
⋮								
20	00	20	00	00	00	50	00	00
21	00	40	00	00	00	a0	00	00
22	00	80	00	00	01	40	00	00
23	03	00	00	00	00	80	00	00
24	00	00	00	00	03	00	00	00
25	06	00	00	00	00	00	00	00
26	00	00	00	00	00	00	00	01
27	00	00	00	00	00	00	00	02
28	00	00	00	00	00	00	00	04
29	00	00	00	00	00	00	00	08
30	00	00	00	00	00	00	00	10
31	00	00	00	00	00	00	00	20
32	00	00	00	00	00	00	00	40
33	00	00	00	00	00	00	00	80
34	00	00	00	00	00	00	01	00
35	00	00	00	00	00	00	02	00
36	00	00	00	00	00	00	04	00
37	00	00	00	00	00	00	08	00
38	00	00	00	00	00	00	10	00
39	00	00	00	00	00	00	20	00
40	00	00	00	00	00	00	40	00
41	00	00	00	00	00	00	80	00
42	00	00	00	00	00	01	00	00
43	00	00	00	00	00	02	00	00
44	00	00	00	00	00	04	00	00
45	00	00	00	00	00	08	00	00
46	00	00	00	00	00	10	00	00
47	00	00	00	00	00	20	00	00
48	00	00	00	00	00	40	00	00
49	00	00	00	00	00	80	00	00
50	00	00	00	00	01	00	00	00
51	02	00	00	00	00	00	00	00
52	00	00	00	00	04	00	00	00
53	08	00	00	00	00	00	00	00
54	04	00	00	00	00	00	00	00
55	06	00	00	00	00	00	00	00
56	00	00	00	00	00	00	00	01
57	00	00	00	00	00	00	00	02
58	00	00	00	00	00	00	00	04
59	00	00	00	00	00	00	00	08
60	00	00	00	00	00	00	00	10
61	00	00	00	00	00	00	00	20
62	00	00	00	00	00	00	00	40
63	00	00	00	00	00	00	00	80
64	00	00	00	00	00	00	01	00