An updated version of a paper that will appear in SAC'03 preproceedings,
*Lecture in Computer Science*, Springer-Verlag, 2003.

# Cryptanalysis of the Alleged SecurID Hash Function*

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 analyze 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. We further show how to use this attack on the full hash to extract information on the secret key stored in the token if the attacker is given several months of the token output. However in this paper we do not address a broader issue of practical security of the fielded SecurID installations.
**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 [2], 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 [4]. 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 [3]. 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). However, it might be hard for a real-life attacker to mount the most effective of these attacks due to their adaptive chosen-plaintext nature, due to the special time-duplication format and due to initial key-dependent permutation which is applied to the input of the hash function. Still, vanishing differentials do occur for 10% of all tokens given only two months of token output and for 35% of all keys given a year of token output. Each vanishing differential leaks up to 19 bits of key-information. In this paper we show how to partially extract this information. We conjecture that given only a few vanishing differentials, full key recovery would take less that $2^{45}$ steps. 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 some results on the block cipher-like structure, the heart of the ASHF. In Sect. 4, we show how to use these observations for the cryptanalysis of the whole cipher. Sect. 5 presents the conclusions.

**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$^{\star}$ | $2^{16} - 2^{18}$ | $2^{18}$ | partial 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 |

$\star$ A full output of the 60-second token for 2-12 months.
$^{*}$ A full output of the 60-second token for one year.
$^{**}$ Here data complexity is measured in full hash outputs which would correspond to two consequitive outputs of the token.

## 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 [4] though we expect that our results apply to 30-second tokens with twice better complexities. 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 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. This transformation consists of an initial key-dependent permutation, four rounds of a block cipher-like function, and a final permutation. After every round the key is changed by XORing the key with the output of that round, hence providing a simple key scheduling algorithm. Finally the 64 bits (*i.e.*, 16 hexadecimal characters) are transformed into 16 digits by the conversion routine. 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 $B_0, B_1, \ldots B_7$ 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

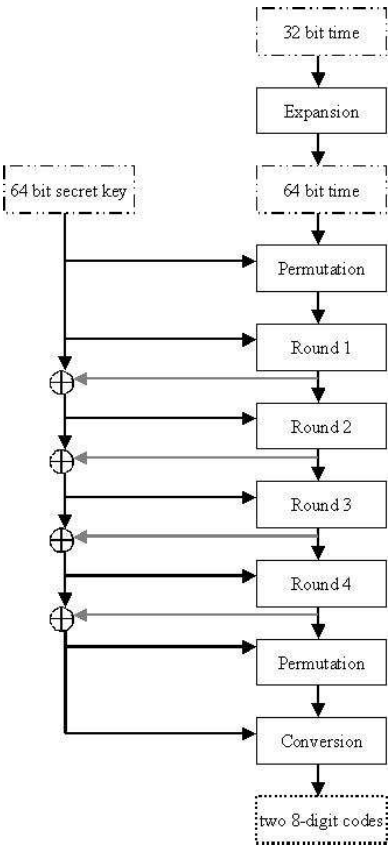


**Fig. 1.** Overview of the ASHF hash function

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 $2^{23}$ 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 $y$. All operations are to be thought of modulo 64. The key is divided into 16 tuples of four bits $\mathrm{K}_0\mathrm{K}_1 \ldots \mathrm{K}_{15}$. First a pointer $p_a$ jumps to bit $u_{K_0}$, and a pointer $p_b$ is set equal to this. Then the four bits strictly before $u_{p_b}$ will be the output bits $y_{60}y_{61}y_{62}y_{63}$.

For the subsequent bits, the pointer $p_b$ is decremented by four, the bits from $u_{p_b}$ till $u_{p_b+K_1-1}$ are overwritten by $u_{p_a}$ till $u_{p_a+K_1-1}$. Both pointers are incremented with $\mathrm{K}_1$, and the four bits before $u_{p_b}$ again yield the next four output bits $y_{56}y_{57}y_{58}y_{59}$. This process is repeated until the 16 4-tuples of the key are used up and the whole output word $y$ is determined.

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.

### 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 $R$.
   - 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 $S$.

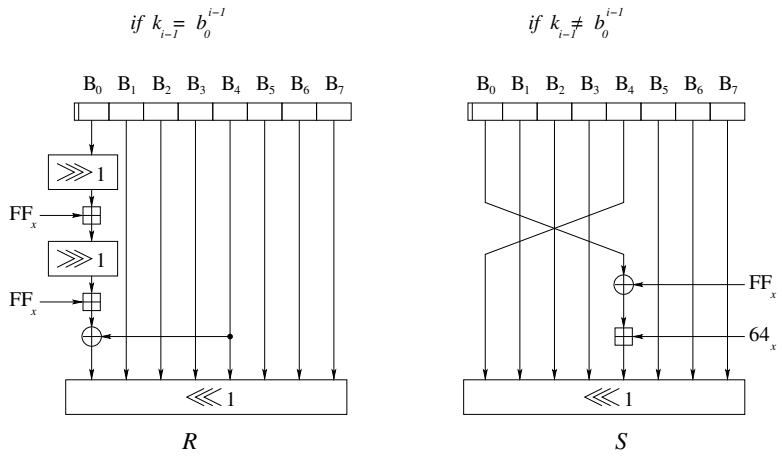


**Fig. 2.** Sub-round $i$ defined by the operations $R$ and $S$.

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 $k = k \oplus b^{64}$.

### 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 C. 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 $2^{-15}$.

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[1].

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 $R$:

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

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.

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

[1] 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 [6–8].

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 give a vanishing differential 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. 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. Another 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 for a subset of keys. Taking a random key, and two 64-bit time values that differ in two bits (for instance in $t_0$ and $t_{32}$, or in $t_{15}$ and $t_{47}$), 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 two months, *i.e.*, $2^{17.4}$ 8-digit (or 6 digit) hashes. 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. In a simulation of this experiment repeated for 1000 different keys, about 10% of the keys, had one or more vanishing differentials. Among these 10% the average number of vanishing differentials was 3, but the variance between different keys was high. If one extends the observation period from two months to one year then 35% of keys will have at least one vanishing differential. The attacker can use this information in several ways.

**Partial key recovery.** From these vanishing differentials, an attacker can deduce information on the initial key-dependent permutation, and thus on the secret key. This information is learned in two ways. First of all, the vanishing differentials can be grouped according to the input differential that causes them. An input differential can only cause a vanishing differential if the initial permutation puts the differences at adequate positions: the differences should enter $B_0$ and/or $B_4$ within an interval of a few subrounds, so that they can vanish simultaneously. This can give information about the sum of some key bits.

Furthermore, within the group of plaintext pairs with the same input differential, it is observed that some pairs are very similar. We then inspect the difference between these input pairs. It can be seen that flipping some bits in the input pair yields another input pair that also causes a vanishing differential (this would assume adaptive chosen plaintext capability). This boils down to saying that these bits do not interfere in the occurrence of the vanishing differential. Very often, the conclusion can be drawn that these bits are permuted to a position further away from $B_0$ and $B_4$ than the bits of the input differential (i.e., they will only enter $B_0$ and $B_4$ after the vanishing differential has occurred). Using this approach, it is possible to extract bits of information on the secret key for about 10–35% of the tokens (depending on the observation period). We conjecture that given a few vanishing differentials it is possible to perform a key-recovery attack on the full ASHF within $2^{45}$ analysis steps. See Appendix B for an example of several vanishing differentials.

**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 [2] 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. This weakness can still be detected in the full ASHF (with and without the required time format), which leaks information on the key in a number of scenarios. 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 [5].

## References

1. E. Biham, A. Shamir, *Differential Cryptanalysis of DES-like Cryptosystems,* Journal of Cryptology, volume 4, number 1, pp. 3–72, 1991.

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

2. V. McLellan, *Re: SecurID Token Emulator*, post to BugTraq, http://cert.uni-stuttgart.de/archive/bugtraq/2001/01/msg00090.html
3. Mudge, Kingpin, *Initial Cryptanalysis of the RSA SecurID Algorithm,* www.atstake.com/ research/reports/acrobat/initial_securid_analysis.pdf
4. I.C. Wiener, *Sample SecurID Token Emulator with Token Secret Import*, post to BugTraq, http://archives.neohapsis.com/archives/bugtraq/2000-12/0428.html
5. RSA security website, http://www.rsasecurity.com/company/news/releases/pr.asp?doc_id=1543
6. 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.
7. 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.
8. Internet Security, Applications, Authentication and Cryptography, University of California, Berkeley, *GSM Cloning* http://www.isaac.cs.berkeley.edu/isaac/gsm-faq.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 $b_7^{64}$,

$b_8^{64} \dots b_{31}^{64}$ are zero. It is easy to see that these values correspond to the value of bit $b_{32}$ after mixing in the key bit $k_{39}$, $k_{40} \dots k_{63}$ 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 $k_{39} \dots k_{63}$ of the key in the final round, is the complement of $b_7 \dots b_{31}$. 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 Example of Vanishing Differentials

In this section we provide an example of vanishing differentials observed in a simulation of a token output during one year. This example is taken from a test run with 100 different token keys. Out of these 45 keys produced at least one vanishing differential, 26 keys produced two or more, 19 produced three or more, 17 produced four or more. In terms of differences: 34 cases had single vanishing difference, 11 cases had two or more vanishing differences.

```
Token key: 659f8031552d30a2_x

1st Input               2nd Input               Difference
------------------------------------------------------------------
0401d0d00401d0d0        0407d0d00407d0d0        0006000000060000
0481d0d00481d0d0        0487d0d00487d0d0        0006000000060000
0441d0d00441d0d0        0447d0d00447d0d0        0006000000060000
04c1d0d004c1d0d0        04c7d0d004c7d0d0        0006000000060000
```

Below we show 64 bits of the input after the initial key-dependent permutation (the 1st line describes 32 bits from $B_0$ to $B_3$, then 2nd 32 bits represent $B_4$ to $B_7$). Here by $*$ we denote bits of the difference and by $V$ bits which can be flipped without spoiling the vanishing differential. Notice that permutation maps

the key bits in chunks. Average chunk has 3 bits. Notice also that, as expected, the bits that can be flipped are mapped after the regions of differences, and that differences are mapped into compact 8-bit areas which are 4-words away from each other.

```
                                                  *  *                                  V  V
55 20 21 54 22 23 52 53| 56 17 18 19 57 58 59 60| 44 45 46 47 48 49 50 51| 24 25 30 43 35 40 41 42

                                                                        *  *                  V  V
36 37 38 39 31 32 33 34| 26 27 28 29 61 62 15 16| 63 00 01 02 03 12 13 14| 04 05 10 11 06 07 08 09
```

One possible attack approach would be to partially reconstruct the mapping of the 8-bit areas around the differences, guess the control bits for 6-8 steps from the moment that the first difference enters the $B_0/B_4$ area and then check if the guess leads to vanishing of the differences. The power of this filter is about $2^{-17}$.

## C 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 |
| ... | | | | | | | | |