Fast Software-Based Attacks on SecurID

Scott Contini[1] and Yiqun Lisa Yin[2]

[1] Macquarie University, Computing Department, NSW 2109 Australia,
`scontini@comp.mq.edu.au`,
[2] Princeton University, EE Department, Princeton, NJ 08540, USA,
`yyin@princeton.edu`

Abstract. SecurID is a widely used hardware token for strengthening authentication in a corporate environment. Recently, Biryukov, Lano, and Preneel presented an attack on the alleged SecurID hash function [1]. They showed that *vanishing differentials* – collisions of the hash function – occur quite frequently, and that such differentials allow an attacker to recover the secret key in the token much faster than exhaustive search. Based on simulation results, they estimated that the running time of their attack would be about 2^{48} full hash operations when using only a single 2-bit vanishing differential.
In this paper, we present techniques to improve the [1] attack. Our theoretical analysis and implementation experiments show that the running time of our improved attack is about 2^{45} hash operations. We then investigate into the use of extra information that an attacker would typically have: multiple vanishing differentials or knowledge that other vanishing differentials do not occur in a nearby time period. When using the extra information, we believe that key recovery can always be accomplished within about 2^{40} hash operations.

1 Introduction

The SecurID, developed by RSA Security, is a hardware token used for strengthening authentication when logging in to remote systems, since passwords by themselves tend to be easily guessable and subject to dictionary attacks. The SecurID adds an "extra factor" of authentication: one must not only prove themselves by getting their password correct, but also by demonstrating that they have the SecurID token assigned to them. The latter is done by entering the 6- or 8-digit code that is being displayed on the token at the time of login.

Each token has within it a 64-bit secret key and an internal clock. Every minute, or every half-minute in some tokens, the secret key and the current time are sent through a cryptographic hash function. The output of the hash function determines the next two authenticator codes, which are displayed on the LCD screen. The secret key is also known to the "ACE/server", so that the same authenticator can independently be computed and verified at the remote end.

If ever a user loses their token, they must report it so that the current token can be deactivated and replaced with a new one. Thus, the user bears some responsibility in maintaining the security of the system. On the other hand, if the user were to temporarily leave his token in a place where it could be observed by others and then later recover it, then it should not be the case that the security of the device could be entirely breached, assuming the device is well-designed.

The scenario just described was considered in a recent publication by Biryukov, Lano, and Preneel [1], where they showed that the hash function that is alleged to be used by SecurID [4] (ASHF) has weak properties that could allow one to find the key much faster than exhaustive search. The attack they describe requires recording all outputs of the SecurID using a PC camera with OCR software, and then later searching the outputs for indication of a *vanishing differential* – two closely related input times that result in the same output hash. If one is discovered, the attacker then has a good chance of finding the internal secret key using a search algorithm that they estimated to be equivalent to 2^{48} hash function operations. On a 2.4 GHz PC, 2^{48} hash operations take about 111 years[3]. It would require over 1300 of these PC's to find the key in a month.

In this paper, we present three techniques to significantly speed up the filtering, which is the bottleneck of their attack. Our theoretical analysis and implementation experiments show that the time complexity can be reduced to about 2^{45} hash operations when using only a single vanishing differential.

We then investigate into the use of extra information that an attacker would ordinarily have, in order to speed up the attack further. This information consists of either multiple vanishing differentials, or knowledge that no other vanishing differentials occur in a nearby time period of the observed one. In either case, the running time can be reduced significantly. Our preliminary analysis suggests that after a vanishing differential is observed, the attacker would nearly always be able to perform the key search algorithm in 2^{40} hash operations or less. On a typical PC, this can be done in about 5 months, making the computing power requirements for the search attainable by almost any individual.

The success probability of all attacks (including [1]) depend upon how long the attacker must wait for a vanishing differential to occur. Simulations have shown that in any one-week period, 1% of the SecurID cards will have a vanishing differential; in any one-year period, 35% of the tokens will have a vanishing differential. According to these statistics, we mention two realistic scenarios in which the token could be compromised. In the first scenario, a user may be on vacation for one week and left his token behind in a place where others could observe it, in which case there is a small but definitely *non-negligible* chance that a collision would happen. In the second scenario, the success is much more likely. Since the cost of SecurID tokens is very expensive, tokens are often reassigned to new users when a previous owner leaves a company [5]. This is a bad idea, since the original user would have a high chance of being able to find the internal key, assuming he recorded many of the outputs while it was in his possession. In light of our new results, token reassignment becomes a very serious risk.

[3] Requires some optimisations to Wiener's code, such as re-ordering bytes to eliminate bswaps.

2 The SecurID Hash Function

We provide a high level description of the alleged SecurID hash function, following the same notation as in [1] wherever possible. More detailed descriptions can be found in [1,4].

The function can be modeled as a keyed hash function $y = H(k, t)$, where k is a 64-bit secret key stored on the SecurID token, t is a 24-bit time obtained from the clock every 30 or 60 seconds, and y is two 6- or 8-digit codes. The function consists of the following steps:

- an expansion function that expands t into a 64-bit "plaintext",
- an initial key-dependent permutation,
- four key-dependent rounds, each of which has 64 subrounds,
- an exclusive-or of the output of each round onto the key,
- a final key-dependent permutation (same algorithm as the initial one), and
- a key-dependent conversion from hexadecimal to decimal.

Throughout the paper, we use the following notation to represent bits, nibbles, and bytes in a word: a 64-bit word b, consisting of bytes $B_0, ..., B_7$, nibbles $\mathtt{B}_0, ..., \mathtt{B}_{15}$, and bits $b_0 b_1 ... b_{63}$. The nibble $\mathtt{B}_0$ corresponds to the most significant nibble of byte 0 and the bit b_0 corresponds to the most significant bit. The other values are as one would expect.

For our analysis, only the time expansion, key-dependent permutation, and the key-dependent rounds are of interest. In the next three sections, we will describe them in more detail.

2.1 Time Expansion

The time t is a 24-bit number representing twice the number of minutes since January 1, 1986 GMT. So the least significant bit is always 0, and if the token outputs codes every minute, then the expansion function will clear the 2nd least significant bit as well. Let the result be represented by the bytes $T_0 T_1 T_2$ where T_0 is the most significant. The expansion is of the form $T_0 T_1 T_2 T_2 T_0 T_1 T_2 T_2$. Note that the least significant byte is replicated 4 times, and the other two bytes are replicated 2 times each.

2.2 Key-Dependent Permutation

We give a more insightful description of how the ASHF key-dependent permutation really works. The original code, obtained by Wiener [4] (apparently by reverse engineering the ACE/server code), is quite cryptic. Our description is different, but produces an equivalent output to his code.

The key-dependent permutation uses the key nibbles $\mathtt{K}_0 \ldots \mathtt{K}_{15}$ in order to select bits of the data for output into a permuted_data array. The data bits will be taken 4 at a time, copied to the permuted_data array from right to left (i.e. higher indexes are filled in first), and then removed from the original data array.

Every time 4 bits are removed from the original data array, the size shrinks by 4. Indexes within that array are always modulo the number of bits remaining.

A pointer m is first initialised to the index $\mathtt{K}_0$. The first 4 bits that are taken are those right before the index of m. For example, if $\mathtt{K}_0$ is 0x2, then bits 62, 63, 0, and 1 are taken. As these bits are removed from the array, the index m is adjusted accordingly so that it continues to point at the same bit it pointed to before the 4 bits were removed. The pointer m is then increased by a value of $\mathtt{K}_1$, and the 4 bits prior to this are taken, as before. The process is repeated until all bits have been taken.

Note that once the algorithm gets down to the final 3 or less key and data nibbles, the number of data bits remaining is at most 12 yet the number of choices for each key nibble is 16. Hence, multiple keys will result in the same permutation, which we call "redundancy of the key with respect to the permutation." This was used in the attack [2], and to a lesser extent in [1].

2.3 Key-Dependent Rounds

Each of the four key-dependent rounds takes as inputs a 64-bit key k and a 64-bit value b^0, and outputs a 64-bit value b^{64}. The key k is then exclusive-ored with the output b^{64} to produce the new key to be used in the next round.

One round consists of 64 subrounds. For $i = 1, ..., 64$, subround i transforms b^{i-1} into b^i using a single key bit k_{i-1}. Depending on whether the key k_{i-1} is equal to b_0^{i-1}, the value b^{i-1} is transformed according to two different functions, denoted by R and S. The details of R and S are not so important for our research, with the exception of two properties:

1. Both the R and the S functions are byte-oriented, that is, they update each of the eight bytes in b^i separately. After the update, only bytes B_0 and B_4 are modified, and the other six bytes remain the same.
2. The way R and S are used causes the hash function to have easy-to-find collisions after a small number of subrounds within the *first* round.

At the end of each subround, all the bits are rotated one bit position to the left. So, up to subround $N \leq 25$ of the first round, only $2N + 14$ data bits have been involved in the computation. This property is used in the Biryukov, Lano, and Preneel attack.

3 The Attack of Biryukov, Lano, and Preneel

The attack of Biryukov, Lano, and Preneel [1] can determine the full 64-bit secret key when given a single collision of the hash function. Suppose that two input times t and t' get expanded and permuted to become 64-bit words b and b', and the two words collide in subround N of the first round. The collision from the pair (t, t') is called a *vanishing differential.* In their key recovery attack, the attacker first guesses the subround N, and then uses a *filtering algorithm* for each N to search the set of candidate keys that make such a vanishing differential possible. According to their simulations, one only needs to do up to $N = 12$ to have a 50% chance of finding the key.[4] A summary of their description for $N = 1$ is given below. For simplicity, assume that a 2-bit vanishing differential is used, though this need not be the case.

A one-time cost precomputation table is needed before the filtering starts. The table contains entries the form

$$(k_0, B_0, B_4, B'_0, B'_4).$$

where k_0 represents a key bit, (B_0, B_4) represent data bytes of b after the initial keyed permutation, and (B'_0, B'_4) represent data bytes of b' after the permutation. The exact entries in the table are those where (B_0, B_4) differs from (B'_0, B'_4) in exactly 2-bits known as the "difference bits," and for which a vanishing differential occurs during the first subround. Since none of the other key bits or data bytes are involved in the first subround, whether a vanishing differential can happen or not for $N = 1$ is completely characterised by this table.

For each entry in the table, the filtering proceeds in two phases, each of which contains two steps.

- *First Phase.* (process the first half of the key bits)
  - *Step One.* Guess key bits $k_1, ..., k_{27}$. Together with k_0, 28 key bits are set, which determines 28 bits of b and b' after the initial key-dependent permutation. Since these bits overlap with the entries in the table in nibbles $\mathtt{B}_9$ and $\mathtt{B}'_9$, a key value that does not produce the correct nibbles for both b and b' is filtered out.
  - *Second Step.* Continue to guess key bits $k_{28}, ..., k_{31}$. Filtering is done using overlaps in nibbles $\mathtt{B}_8$ and $\mathtt{B}'_8$.
- *Second Phase.* (process the second half of the key bits)
  - *First Step.* Continue to guess key bits $k_{32}, ..., k_{59}$. Filtering is done using overlaps in nibbles $\mathtt{B}_1$ and $\mathtt{B}'_1$.
  - *Second Step.* Continue to guess key bits $k_{60}, ..., k_{63}$. Filtering is done using overlaps in nibbles $\mathtt{B}_0$ and $\mathtt{B}'_0$.

Finally, each candidate key that passes the filtering is tested by performing a full hash function to see if it is the correct key. For general N, the two phases of filtering each involve $\lceil \frac{7+N}{4} \rceil$ data nibbles, so the phases each have $\lceil \frac{7+N}{4} \rceil$ steps.

4 Analysis of the Biryukov, Lano, and Preneel Attack

Biryukov, Lano, and Preneel estimated the time complexity of their attack through simulation. They provided results for $N = 1$: step 1 of phase 1 reduced the number of possibilities to 2^{27}, step 2 of phase 1 further reduced the count to to 2^{25}, step 1 of phase 2 increased the count to 2^{45}, and step 2 of phase

[4] Our own simulations suggest that one needs to search up to $N = 16$. The discrepancy is due to differences in the way the attack is viewed, which we elaborate on in Section 7.1. For larger values of N, the cost of the precomputation stage becomes prohibitive.

2 resulted in 2^{41} true candidates. For larger values of N, they expect that the complexity of the attack would be lower due to stronger filtering.

Here we analyse their algorithm, giving some mathematical justification for the simulation results they observed and also showing that their conjecture of the filtering improving for larger N appears to be correct. In our analysis, we sometimes treat probabilities as if they are independent, which is not always true, but it is assumed that it provides a reasonable approximation.

Some properties of the precomputed tables are used in the analysis. For a given value of N, the table entries are of the following form:

- legal values for the key bits in indices $0, \ldots, N-1$,
- legal values for the plaintext pairs after the initial permutation in bit indices $32, 33, \ldots, 38+N$ which we label as (W_4, W_4') (we use the subscript 4 because the words begins at byte B_4), and
- legal values for the plaintext pairs after the initial permutation in bit indices $0, 1, \ldots, 6+N$ which we label as (W_0, W_0') (the word begins at byte B_0).

The words W_0, W_0', W_4, W_4' each consist of $7+N$ bits and the number of key bits is N. By "legal values" we mean that the combination of plaintext bits after the initial permutation and key bits will cause the difference to vanish in subround N. We also have one other requirement, which was previously overlooked (including in an earlier version of this research): the values of the two bits in b (or b') where the differences are located must be the same, due to the way the time expansion works. This reduces the number of table entries and results in a speedup to the filtering. Although this is one of our three main filtering speedups, we apply it to the analysis of the original [1] algorithm in order to keep things as clean as possible.

Analysis of final number of candidates: Analysing the final step is equivalent to determining the true number of candidates that need to be tested with the full SecurID hash function. The expected number of true candidates can easily be determined since anything that matches an entry in the precomputed table will result in a vanishing differential. In other words, the entries in the table are not only a necessary set of cases for a vanishing differential to occur, but also sufficient.

For each entry in the precomputed table, we have:

- Only a portion of about $1/\binom{64}{2}$ of the 2^{64} keys will permute the 2 difference bits into the locations corresponding to what is in that table entry.
- With probability $\frac{1}{2}$, the value of the two difference bits will match those in the table (recall, the 2 bits in b must be the same, and the corresponding bits in b' are the complement).
- With probability $\frac{1}{2^{2N+12}}$, the remaining permuted data bits will match the table entry.
- With probability $\frac{1}{2^N}$ the guessed key bits will match the entry of the table.

Hence, the expected number of final candidates is:

$$\text{table size} \times 2^{64} \times \frac{1}{\binom{64}{2}} \times \frac{1}{2} \times \frac{1}{2^{2N+12}} \times \frac{1}{2^N} \quad . \tag{1}$$

Run time analysis of phase 2, step 1: Phase 2, step 1 of the Biryukov, Lano, and Preneel attack is typically the dominant cost. To analyse it, we must first determine the number of candidates passing phase 1.

Define C_0 to be the number of unique table entries of the form $(k_0, \ldots, k_{N-1}, W_4, W_4')$ where $W_4 = W_4'$, C_1 similarly except $W_4 \oplus W_4'$ having hamming weight 1, and C_2 similarly except $W_4 \oplus W_4'$ having hamming weight 2.

Among the of 2^{32} key bits considered in phase 1, a fraction of $\binom{57-N}{2}/\binom{64}{2}$ will put no difference in the tuple (W_4, W_4'). Of those, only a fraction of $\frac{C_0}{2^{7+N}}$ will match one of the C_0 unique entries in the table for W_4 (which is the same as W_4'). With probability $\frac{1}{2^N}$, the guessed key bits will match those in the table as well. Thus, the expected number of 32-bit keys resulting in no difference in (W_4, W_4') that pass phase 1 is:

$$2^{32} \times \frac{\binom{57-N}{2}}{\binom{64}{2}} \times \frac{C_0}{2^{7+N}} \times \frac{1}{2^N} = 2^{19-2N} \times \frac{3192 - 113N + N^2}{63} \times C_0 \quad .$$

For 1-bit differences, the equation is

$$2^{32} \times \frac{\binom{57-N}{1}}{\binom{64}{2}} \times \frac{1}{2} \times \frac{C_1}{2^{6+N}} \times \frac{1}{2^N} = 2^{20-2N} \times \frac{57 - N}{63} \times C_1 \quad .$$

For 2-bit differences, the equation is

$$2^{32} \times \frac{1}{\binom{64}{2}} \times \frac{1}{2} \times \frac{C_2}{2^{5+N}} \times \frac{1}{2^N} = 2^{21-2N} \times \frac{C_2}{63} \quad .$$

The $\frac{1}{2}$ in this last equation accounts for whether the two difference bits in the first plaintext match the table entry (the bits must be the same). Thus, the expected number of candidates to pass the phase 1 is

$$T = \frac{2^{19-2N}}{63} \times \left[(3192 - 113N + N^2)C_0 + (114 - 2N)C_1 + 4C_2\right] \quad . \tag{2}$$

The first step in phase 2 involves guessing enough key bits so that the resulting permuted data array just begins to overlap with W_0 and W_0'. The exact number of key bits guessed in this step is $4 \times \lfloor \frac{29-N}{4} \rfloor$. Under the assumption that the permutation is 5% of the time required to do the full SecurID hash, the running time is equivalent to

$$T \times 2^{4 \times \lfloor \frac{29-N}{4} \rfloor} \times \frac{4 \times \lfloor \frac{29-N}{4} \rfloor}{64} \times 2 \times 0.05 \times s \tag{3}$$

full hash operations, where s is the speedup factor that can be obtained by taking advantage of the redundancy in the key with respect to the permutation. The value of s is $\frac{96}{256}$ for $N = 1$, $\frac{12}{16}$ for $N = 2..5$, and 1 for all other values.

We remark that in some cases, there is a chance that the second step of phase 2 may be a bit more time consuming than the first. A sufficient but not necessary condition for step 1 to be the most time consuming is if the fraction of values that remain is less than $\frac{\lfloor \frac{29-N}{4} \rfloor}{16}$ of the values considered. This is usually the case. We shall ignore the exceptional cases for now, but will deal with them when we present our filtering speedups.

Combined analysis: The running time of algorithm [1] for a particular value of N is expected to be the approximately the sum of equations 3 and 1. For $N = 1..6$, these running times are given in Table 1. Again, we reiterate that the table sizes are different from [1] because of an extra condition due to the time expansion, which also gives a small improvement in the running time. The analysis for $N = 1$ closely matches the simulated results from [1][5].

Table 1. Computing the running time estimates of algorithm [1] for $N = 1..6$.

N	Table size	C_0	C_1	C_2	T	Time for phase 2, step 1	Time for testing final candidates	Total time
1	12	5	2	0	$2^{25.0}$	$2^{47.0}$	$2^{40.6}$	$2^{47.0}$
2	152	11	64	44	$2^{24.3}$	$2^{43.2}$	$2^{41.3}$	$2^{43.5}$
3	1130	64	362	128	$2^{24.8}$	$2^{43.6}$	$2^{41.2}$	$2^{43.9}$
4	7292	453	1750	712	$2^{25.5}$	$2^{44.3}$	$2^{40.9}$	$2^{44.4}$
5	48212	2775	10614	3864	$2^{26.0}$	$2^{44.9}$	$2^{40.6}$	$2^{44.9}$
6	276788	15076	52716	19520	$2^{26.4}$	$2^{41.4}$	$2^{40.1}$	$2^{41.9}$

Even though the number of candidates T after the first phase are approximately the same as N goes from 1 to 2 and also from 5 to 6, the running times of the phase 2, step 1 drop significantly. This is because one less nibble of the key is being guessed, and an extra filtering step is being added. In general, we see the pattern that larger values of N are contributing less and less to the sum of the running times, which agrees with the conjecture from [1]. The total running time for $N = 1$ to 6 is $2^{47.7}$ and larger values of N would appear to add minimally to this total. For vanishing differentials that involve $\geq$ 4-bits, which happens about one third of the time, preliminary analysis suggests that the run time is better.

[5] A small discrepancy for T exists due to the fact that their simulations involved a precomputed table about twice as big ours.

5 Faster Filtering

Table 1 illustrates that the trick to speeding up the key recovery attack in [1] is faster filtering. We have found three ways in which their third filtering can be sped up:

1. Only include entries in the precomputed table that actually can be derived from the time expansion. In particular, the values of the two bits in b (or b') where the differences are located must be the same.
2. In the original filter, a separate permutation is computed for each trial key. This is inefficient, since most of the permuted bits from one particular permutation will overlap with those from many other permutations. Thus, we can amortise the cost of the permutation computations.
3. We can detect ahead of time when a large portion of keys will result in "bad" permutations in steps 1 of both phase 1 and phase 2, and the filtering process can skip past chunks of these bad permutations.

The first technique was already applied to the analyses in the previous section. Without this improvement, the running time would have been about 50% worse.

The second technique is aimed at reducing the numerator of the factor $\frac{4\times\lfloor\frac{29-N}{4}\rfloor}{64} = \frac{\lfloor\frac{29-N}{4}\rfloor}{16}$ in equation 3. To do this, we view the key as a 64-bit counter, where k_0 is the most significant bit and k_{63} is the least. In phase 2, step 1 of the filter, the bits $k_0, \ldots, k_{31}$ are fixed and so are some of the least significant bits (the exact number depends upon N), so we can exclude these for now. The keys are tried in order via a recursive procedure that handles one key nibble at a time. At the j^{th} recursive branch, each of the possibilities for nibble $\mathtt{K}_{7+j}$ are tried. The part of the permutation for that nibble is computed, and then the $j+1^{\text{st}}$ recursive branch is taken. The level of recursion stops when key nibble $\mathtt{K}_{7+\lfloor\frac{29-N}{4}\rfloor}$ is reached. Thus, the $\lfloor\frac{29-N}{4}\rfloor$ from equation 3 gets replaced with the average cost per permutation trial, which is $\sum_{i=0}^{\lfloor\frac{29-N}{4}\rfloor-1} 2^{-4i} \approx 1.07$. Observe that when $N = 1$, this results in a factor of $\frac{7}{1.07} \approx 6.5$ speedup. This trick alone knocks more than 2 bits off the running time.

The third speedup is dependent upon the second. It will apply in both phases of the filtering. During the process of trying a permutation, there will be large chunks of bad trial keys that can be identified immediately and skipped. In particular, whenever a difference bit is placed outside of words (W_0, W_0') and (W_4, W_4'), the key can be skipped because the difference is not in a legal position. Moreover, any other key with the same most significant bits (up to the key nibble that placed the difference bit) will also result in illegal values, implying that the entire recursive branch can be skipped. Heuristically, one would expect that the number of keys that get tested for filtering in phase 2, step 1 to be about a fraction of about $\binom{14+2N}{2}/\binom{64}{2}$ of the number for the attack in [1]. However, this over simplifies the analysis. A more proper analysis can be done similar to our analysis in the previous section.

The combined speedups give the run times in Table 2. In all cases, phase 2, step 1 has become faster than the time for testing the final candidates. The running time for $N = 1..6$ is $2^{43.6}$, so we conjecture that the run time for N up to 16 is no more than $16/6 \times 2^{43.6} \approx 2^{45}$. We remark that the run times for the third speedup ignore the overhead time for rejecting keys in phase 2 where the difference bit gets put outside of (W_0, W_0'), but such overhead time is expected to make little difference. We have also ignored the time for other filtering steps of the algorithm. Of those, only step 2 of phase 2 is expected to have comparable cost to step 1 of phase 2. In fact, it can be more costly, especially when $N \equiv 2 \bmod 4$. However, there are several possible speedups for this step, particularly when N is small (this restriction is for practical reasons) where the run time becomes most relevant. Such speedups involve using additional preocomputed lookup tables to determine valid keys from the remaining data bits and testing whether the hamming weight of the remaining data bits matches that of the precomputed table entries before blindly trying keys. Therefore, it seems fair to assume that the testing of final candidates will always be the dominant cost in the modified algorithm.

Table 2. Running times using our improved filter, for $N = 1..6$.

N	Time for phase 2, step 1	Time for testing final candidates	Total time
1	$2^{38.7}$	$2^{40.6}$	$2^{40.9}$
2	$2^{36.4}$	$2^{41.3}$	$2^{41.3}$
3	$2^{37.1}$	$2^{41.2}$	$2^{41.3}$
4	$2^{37.9}$	$2^{40.9}$	$2^{41.1}$
5	$2^{38.6}$	$2^{40.6}$	$2^{40.9}$
6	$2^{35.7}$	$2^{40.1}$	$2^{40.2}$

Although it appears that we cannot do much better using only a single vanishing differential, we can improve the situation if we use other information that an attacker would have. In later sections we will show that we can improve the time greatly if we take advantage of multiple vanishing differentials, or if we take advantage of knowledge that no other vanishing differentials occur within a small time period of the observed one.

6 Software Implementation

The attack of Biryukov, Lano, and Preneel was specially designed to keep RAM usage low - only one of the precomputed table entries needs to be in program memory at a time. We tested our ideas only for $N = 1$ and 2-bit differences, and since the table size is small, we took the freedom of implementing a slight variant of their attack which kept the whole precomputed table in memory at once.

We programmed all filtering steps of both phases and the three main filtering speedups. In addition, we programmed an extra "table lookup" speedup that would improve the running time by a factor of 8 for $N = 1$. The extra speedup is only applicable for small values of N due to the memory requirements. Thus, the running time is expected to be 8 times faster than the $2^{38.7}$ listed in Table 2. On our 2.4 GHz PC, this translates to about 8 days of effort.

Our code did the search in numerical order, when the key is viewed as a counter as described in Section 5. The only thing we did not do was testing the final candidates using the real function. Instead, we just stopped when we arrived at the target key. So our implementation was designed to test and time the filtering only, in order to confirm that filtering is significantly faster than testing of the final candidates.

At the time of writing, we have not done the full key search yet. However, we have done a search that starts out knowing the correct first nibble of the key. The key we were searching for is `356b48b3ae15c271` which yields a vanishing differential when times `0x1c3ba8` and `0x1c3aa8` are sent in. We were able to find the key in 13.8 hours. If we assume that the full search will take at most 2^4 times longer, the full running time would be 9.2 days, which is on target of expectations.

7 Multiple Vanishing Differentials

There are two scenarios for multiple vanishing differentials: when they have the same difference and when they have different differences. The former is more likely to occur, but in either case we can speed up the attack.

7.1 Multiple Vanishing Differentials with the Same Difference

According to computer simulations, about 45% of the keys that had a collision over a two month period will actually have at least 2 collisions. There is a simple explanation for this, and a way to use the observation to speed up the key search even more.

Consider a vanishing differential which comes from times $t = T_0T_1T_2$ and $t' = T_0'T_1'T_2'$. As we saw earlier, the only bits that determine whether the vanishing differential will occur at a particular subround are those that get permuted into words W_0, W_0', W_4, and W_4'. Suppose we flip one of the bits in T_2 and T_2' (the same bit in each). This bit will be replicated four times in the time expansion. If, after the permutation, none of those bits end up in W_0, W_0', W_4, or W_4', then we will witness another vanishing differential. The new vanishing differential will follow the same difference path and disappear in the same subround. Thus, new information is learned that can be used to speed up the key search, which we explain below. In the case that another vanishing differential does *not* occur, information is also learned which can improve the search, which is detailed in Section 8.

Following the above thought process, it is evident that:

– Flipping time bits in T_1, T_1' or T_0, T_0' will only replicate the flipped bit twice in the expansion. Since there are only two bits that are not allowed to be in W_0, W_0', W_4, and W_4', the collision is more likely to occur. On the other hand, the time between the collisions is increased, since these are more significant time bits.
– Multiple vanishing differentials are more likely to occur when the first collision happened in a small number of subrounds. This is because the words W_0, W_0', W_4, and W_4' are smaller, giving more places where the flipped bits can land without interfering with the collision.[6]
– The converse of these observations is that when multiple vanishing differentials occur, it is most often the case that the collisions all happened in the same subround and followed the same difference path. Moreover, the collisions usually happen within a few subrounds.

By simply eying the time data that caused the multiple vanishing differentials, one can determine with close to 100% accuracy whether this situation has happened. The signs of it are: 1) Same input difference for all vanishing differentials, 2) All input times differ in only a few bits, and 3) It is the same bits that differ in all cases. An example is given in Appendix B.

The attacker learns $z \geq 2$ bits which cannot be permuted to words W_0, W_0', W_4, or W_4'. This new knowledge can be combined with our third filtering speedup to skip past more bad keys. The expected number of final key candidates to be tested becomes a fraction of $\binom{50-2N}{z} / \binom{64}{z}$ of the values given in Table 2. See Table 3 for a summary of these figures when $z = 2$, $z = 4$, and $z = 8$. The times can be further reduced using information about where certain related plaintexts did not cause a vanishing differential: see Section 8.

Table 3. Number of final candidates assuming the attacker became aware of z-bits that do not get permuted into words W_0, W_0', W_4, or W_4'.

N	Number of final cands using only a single collision	Number of final cands with $z = 2$	Number of final cands with $z = 4$	Number of final cands with $z = 8$
1	$2^{40.6}$	$2^{39.8}$	$2^{38.9}$	$2^{37.0}$
2	$2^{41.3}$	$2^{40.3}$	$2^{39.3}$	$2^{37.2}$
3	$2^{41.2}$	$2^{40.1}$	$2^{39.0}$	$2^{36.6}$
4	$2^{40.9}$	$2^{39.7}$	$2^{38.4}$	$2^{35.7}$
5	$2^{40.6}$	$2^{39.2}$	$2^{37.8}$	$2^{34.8}$
6	$2^{40.1}$	$2^{38.6}$	$2^{37.0}$	$2^{33.6}$

[6] This is the reason for the apparent discrepancy between our research claiming that one needs to precompute up to $N = 16$ in order to have a $\geq 50\%$ of find the key and [1] claiming 12. In our view, the attacker has a single token and will perform a key search once a single vanishing differential has occurred. In their view, the attacker has several tokens for a fixed period of time, and the attacker selects a vanishing differential randomly among all vanishing differentials that have occurred [3]. Since their view includes multiple vanishing differentials, the expected number of subrounds is less.

7.2 Multiple Vanishing Differentials with Different Differences

Given two vanishing differentials with different differences, the number of candidate keys can be reduced significantly by constructing more effective filters in each step. Denote the two pairs of vanishing differentials V_1 and V_2, and their N values N_1 and N_2.

We first make a guess of (N_1, N_2). The number of guesses will be quadratic in the number of subrounds tested up to. The following is a simplified sketch for the new filtering algorithm.

– *First Phase.* Take V_1 and guess the first 32 bits of the key. For each 32-bit key that produces a valid (W_4, W_4'), test it against V_2 to see if it also produces a valid (W_4, W_4').
– *Second Phase.* For 32-bit keys that pass phase 1, do the same thing to guess the second 32 bits of the key.

The main idea here is to do double filtering within each stage so that the number of candidate keys is further reduced in comparison to when only a single vanishing differential is used.

When $N_1 = N_2 = 1$, the probability that a 32-bit key passes phase 1 (see Table 1) is $2^{25.0}/2^{32} = 2^{-7.0}$ (assuming using the original filter of [1] - it is even more reduced using our improved filter), and the probability that a 64-bit key passes both phases is $2^{40.6}/2^{64} = 2^{-23.4}$. If the two vanishing differentials are indeed *independent*, we would expect the number of keys to pass the first phase to be

$$2^{32} \times 2^{-7.0} \times 2^{-7.0} = 2^{18}$$

and the number of keys to pass both phases to be

$$2^{64} \times 2^{-23.4} \times 2^{-23.4} = 2^{17.2}.$$

Experimental results will reveal whether these figures are attainable in practice, but even if they are not, a big speed up is still expected. The situation should be better in the cases where differences with hamming weights ≥ 4 are involved.

We should mention the caveat that the chances of success using the above technique are lower, since we need *both* difference pairs to disappear within 16 subrounds. On the other hand, the cost of trying this algorithm for two difference pairs is expected to be substantially cheaper than trying the previous algorithms for only one. Therefore, the double filtering should add negligible overhead to the search in the cases that it fails, and would greatly speedup the search when it is successful.

8 Using Non-Vanishing Differentials with a Vanishing Differential

In Section 7.1, we argued that even if only a single vanishing differential occurs over some time period, the search can still be sped up if one takes advantage of knowing where related differentials do not vanish. Here, we give the details.

Assume a vanishing differential occurred at times t and t', but no vanishing differential occurred among the time pairs $(t \oplus 2^i, t' \oplus 2^i)$ for $i = 2, \ldots, j$. We start with $i \geq 2$ because in the most typical case, where authenticators are displayed every minute, the least two significant bits of the time are 0 (see Section 2.1). For the values $2 \leq i \leq 7$, the difference is replicated 4 times in the time expansion, and for $i \geq 8$, it is replicated twice.

For each value of i, we learn a set of 2 or 4 bits for which at least one in each set must be permuted into the words W_0, W_0', W_4, or W_4'. Let us label these sets as $U_2, \ldots, U_j$. For simplicity, we will take $j = 13$, which corresponds to no other vanishing differential within a window of 2.8 days before or after the observed one. So, we are interested in the probability of at least one bit in each of these sets getting permuted into words W_0, W_0', W_4, or W_4'.

We say a set U_i is *represented* with $c_i \geq 1$ bits if exactly c_i bits from U_i get permuted into W_0, W_0', W_4, or W_4'. The number of ways $2N + 14$ bits can be selected to end up in W_0, W_0', W_4, or W_4' is $\binom{64}{2N+14}$. The number of ways that exactly c_i bits are represented in the selection for $2 \leq i \leq 13$ is

$$\prod_{i=2}^{7} \binom{4}{c_i} \times \prod_{i=8}^{13} \binom{2}{c_i} \times \binom{28}{2N + 14 - \sum_{i=2}^{13} c_i}.$$

The first product tells the number of ways of selecting c_i bits from each set that has 4 bits, the second product is the same except for among sets with 2 bits, and the third product is the number of ways of selecting the remaining bits from the 28 bits that are not among any of the U_i. Thus, our desired probability is:

$$\sum_{\text{all valid } (c_2, \ldots, c_{13})} \frac{\prod_{i=2}^{7} \binom{4}{c_i} \times \prod_{i=8}^{13} \binom{2}{c_i} \times \binom{28}{2N+14-\sum_{i=2}^{13} c_i}}{\binom{64}{2N+14}} \qquad (4)$$

where *valid* $(c_2, \ldots, c_{13})$ means that each value is at least 1, but the sum of all values is no more than $2N + 14$.

We have computed these probabilities using the Magma [6] computer algebra package. The probabilities, and corresponding running time for the testing of final candidates are given in Table 4. Monte Carlo experiments have been done to double-check the accuracy of these results. The fact that the probabilities are so small for low values of N is consistent with the argument in Section 7.1 that when a collision happens early, other collisions are likely to follow soon after.

One should not assume that the times for the testing the final candidates given in Table 4 are the dominant cost in applying this strategy. Unlike the filtering speedups given in Sections 5 and 7.1, the use of non-vanishing differentials seem to require more overhead in checking the conditions. So although we do not have an exact running time, we confidently surmise that the use of non-vanishing differentials will reduce the time down below 2^{40} hash operations.

Table 4. Assuming no more vanishing differentials occur within 2.8 days before or after of a given vanishing differential, the final testing of candidates can be improved by the amounts given in this table.

N	Fraction of keys having property	Time for testing final candidates
1	$2^{-14.3}$	$2^{26.3}$
2	$2^{-11.7}$	$2^{29.6}$
3	$2^{-9.7}$	$2^{31.5}$
4	$2^{-8.1}$	$2^{32.8}$
5	$2^{-6.7}$	$2^{33.9}$
6	$2^{-5.7}$	$2^{34.4}$

9 Conclusion

The design of the alleged SecurID hash function appears to have several problems. The most serious appears to be collisions that happen far too frequently and very early within the computation. The involvement of only a small fraction of bits in the subrounds exacerbates the problem. Moreover, the redundancy of the key with respect to the initial permutation adds an extra avenue of attack. Altogether, ASHF is substantially weaker than one would expect from a modern day hash function.

Our research has shown that the key recovery attack in [1] can be sped up by more than a factor of 8, giving an improved attack with time complexity about 2^{45} hash operations. In practice, the attacker can actually obtain more information than just a single collision. We have shown that, with this extra information, the time complexity can be further reduced to about 2^{40} hash operations, making the attack doable by anyone with a modern PC.

acknowledgements: We are grateful to Joe Lano for his insights and helpful comments, and for his hospitality while the first author of this document visited Brussels.

References

1. A. Biryukov, J. Lano, B. Preneel. *Cryptanalysis of the Alleged SecurID Hash Function*, In Proceedings of SAC 2003, to appear in LNCS. A longer version of this paper is available online from http://eprint.iacr.org/2003/162.
2. S. Contini, *The Effect of a Single Vanishing Differential in ASHF*, sci.crypt post, 6 Sep, 2003.
3. J. Lano, private communication, 28 Oct, 2003.
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 , 21 Dec, 2000.

5. *Tips on Reassigning SecurID Cards and Requesting New SecurID Cards*, AMS Newsletter, March 2002, Issue No. 117. Available at http://www.utoronto.ca/ams/news/117/html/117-5.htm .
6. *The Magma Computer Algebra Package*. Information available at http://magma.maths.usyd.edu.au/magma/ .

A Analysing Precomputed Tables

Using computer experiments, we were able to exhaustively search for valid entries in the precomputed table up to $N = 6$ for 2-bit vanishing differentials and up to $N = 4$ for 4-bit differentials at this point. It was predicted in [1] that the size of the table gets larger by a factor of 8 as N grows and it may take up to 2^{44} steps and 500GB memory to precompute the table for $N = 12$.

Here we make an attempt to derive the entries in the table analytically when $N = 1$. If we could extend the method to $N > 1$, we may be able to enumerate the entries analytically without expensive precomputation and storage.

We start with Equation (6) in [1]. Note that we are trying to find constraints for the values in the subround $i-1$. So for simplicity, we will omit the superscript $i - 1$ from now on, and Equation (6) becames the following.

$$\begin{aligned} B'_4 &= ((((B_0 >>> 1) - 1) >>> 1) - 1) \oplus B_4, \\ B'_0 &= 100 - B_4 \ . \end{aligned} \tag{5}$$

We first note that B_0 and B'_0 have to be different in the msb. Therefore, there is at least one bit difference in (B_0, B'_0). The other bit difference can be placed either in the remaining 7 bits of (B_0, B'_0) or any of the 8 bits in (B_4, B'_4).

Rewriting Equation 5, we have

$$B_0 = (((B_4 \oplus B'_4) + 1) <<< 1) + 1) <<< 1.$$

Since there are at most one bit difference in (B_4, B'_4), it can only take on 9 possible values: 0 (for no bit difference) or 2^i (for one bit difference in bit i). Below, for each possible value of (B_4, B'_4), we enumerate the possible values of (B_0, B'_0). During the enumeartion, we also take into consideration the additional requirement that the two bits in b where the differences occur must be the same (See Section 4).

– If $B_4 \oplus B'_4 = 0$, then B_0 =0x06. Since there is no bit difference in (B_4, B'_4), we know that B_0 and B'_0 differ in two bits – one of them must be the msb, and the other can be any of the remaining 7 bits.

$B_4 \oplus B'_4$	B_0	B'_0	k_0
0x00	0x06	0x87, 84, 82, 8e, 96, a6, c6	0

The additional requirement rules out two possible values of B'_0 (0x84, 0x82), leaving 5 possible combinations.

– If $B_4 \oplus B'_4 = 2^i$, then there is only one bit difference in (B_0, B'_0), which is the msb. In this case, there are only one choice for B'_0 for each B_0.

$B_4 \oplus B'_4$	B_0	B'_0	k_0
0x01	0x0a	0x8a	0
0x02	0x0e	0x8e	0
0x04	0x16	0x96	0
0x08	0x26	0xa6	0
0x10	0x46	0xc6	0
0x20	0x86	0x06	1
0x40	0x07	0x87	0
0x80	0x08	0x88	0

The additional requirement rules out every combination above except the first one (B_0 =0x0a and B'_0 =0x8a).

Combining the above two cases, we have $5 + 1 = 6$ pairs of (B_0, B'_0), each of which giving a valid tuple $(k_0, B_0, B_4, B'_0, B'_4)$, where k_0 is the msb of B_0.

Finally, note that if (k_0, a, b, c, d) is a valid tuple, than (k_0, c, d, a, b) is also a valid typle. For example, if (0, 0x06, 0xdd, 0x87, 0xdd) is valid, then (0, 0x87, 0xdd, 0x06, 0xdd) is also valid. Therefore, the table consists of a total of $2 \times 6 = 12$ entries. These entries match the results from our simulation.

B Example of Multiple Vanishing Differentials

Table 5 is an example where 16 vanishing differentials happened within 1.3 days. All had the same difference path, which collided at $N = 2$. One can see that only the 4 least significant bits of time byte T_1 differ. Since each of these bits are duplicated twice, the expected running time of the last steps is given by $z = 8$ in Table 3. Taking into consideration $N = 2$, the total time is expected to be on the order of 2^{38} operations.

Table 5. Example of 16 vanishing differentials that happened within 1.3 days, using key `b5 a9 f4 8c 16 23 a6 1a`.

First plaintext	Second plaintext
`1e 80 8c 8c 1e 80 8c 8c`	`1e 90 8c 8c 1e 90 8c 8c`
`1e 81 8c 8c 1e 81 8c 8c`	`1e 91 8c 8c 1e 91 8c 8c`
`1e 82 8c 8c 1e 82 8c 8c`	`1e 92 8c 8c 1e 92 8c 8c`
`1e 83 8c 8c 1e 83 8c 8c`	`1e 93 8c 8c 1e 93 8c 8c`
`1e 84 8c 8c 1e 84 8c 8c`	`1e 94 8c 8c 1e 94 8c 8c`
`1e 85 8c 8c 1e 85 8c 8c`	`1e 95 8c 8c 1e 95 8c 8c`
`1e 86 8c 8c 1e 86 8c 8c`	`1e 96 8c 8c 1e 96 8c 8c`
`1e 87 8c 8c 1e 87 8c 8c`	`1e 97 8c 8c 1e 97 8c 8c`
`1e 88 8c 8c 1e 88 8c 8c`	`1e 98 8c 8c 1e 98 8c 8c`
`1e 89 8c 8c 1e 89 8c 8c`	`1e 99 8c 8c 1e 99 8c 8c`
`1e 8a 8c 8c 1e 8a 8c 8c`	`1e 9a 8c 8c 1e 9a 8c 8c`
`1e 8b 8c 8c 1e 8b 8c 8c`	`1e 9b 8c 8c 1e 9b 8c 8c`
`1e 8c 8c 8c 1e 8c 8c 8c`	`1e 9c 8c 8c 1e 9c 8c 8c`
`1e 8d 8c 8c 1e 8d 8c 8c`	`1e 9d 8c 8c 1e 9d 8c 8c`
`1e 8e 8c 8c 1e 8e 8c 8c`	`1e 9e 8c 8c 1e 9e 8c 8c`
`1e 8f 8c 8c 1e 8f 8c 8c`	`1e 9f 8c 8c 1e 9f 8c 8c`