Improved Cryptanalysis of SecurID

Scott Contini
Computing Department
Macquarie University
NSW 2109 Australia
scontini@comp.mq.edu.au

Yiqun Lisa Yin
EE Department
Princeton University
Princeton, NJ 08540
yyin@princeton.edu

October 21, 2003

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 given a single 2-bit vanishing differential, the running time of their attack would be about 2^{48} full hash operations.

In this paper, we first give a more detailed analysis of the attack in [1] and present several techniques to improve it significantly. Our theoretical analysis and implementation experiments show that the running time of our improved attack is about 2^{44} hash operations, though special cases involving $\geq$ 4-bit differentials (which happen about one third of the time) reduce the time further. 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, it appears 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 LED screen display. The secret key is also held within 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 [3] (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 38 years. It would require about 450 of these PC's to find the key in a month, which is attainable by anyone in a typical medium-sized corporation.

In this paper, we first go through a deeper analysis of the [1] algorithm, giving further justification of their conjectured running time of 2^{48}. We present three techniques to significantly speed up the filtering step, which is the bottleneck of their attack. Our theoretical analysis and implementation experiments show that the time complexity can be reduced to about 2^{44} hash operations when using only a single vanishing differential. If the vanishing differential involved $\geq$ 4-bits, which happens about one third of the time, then the running time can be further reduced to about 2^{40} hash operations.

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 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 2 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 - the longer the period is, the higher the chance that a token will have a vanishing differential. For example, 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.

We should note, however, that the longer the device is out of a user's control, the more likely that the user will recognise it and have it deactivated. So, we consider two realistic scenarios in which the token could be 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 1% chance that a collision would happen. This probability is small, but definitely *non-negligible*, especially considering that a single large corporation may have thousands of SecurID users. 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 [4]. This is a very 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, the token reassignment scenario becomes a very serious risk.

RSA Security has begun to upgrade their tokens to use an AES based hash. Based on the attacks described in this paper and [1], we recommend that all of the older tokens be replaced with the upgraded AES based tokens.

2 The SecurID hash function

In this section, we provide a high level description of the alleged SecurID hash function. Detailed descriptions can be found in [1, 3]. We will follow the same notation as those in [1] wherever possible.

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 $T_0T_1T_2$ where T_0 is the most significant. The expansion is of the form $T_0T_1T_2T_2T_0T_1T_2T_2$. Note that the least significant byte is replicated 4 times, and the other two bytes are replicated two 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 I.C. Wiener [3] (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 0xa, then bits 6, 7, 8, and 9 are taken. 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]. Interestingly, [1] mentions that there are 14-bits of redundancy on average, yet the attacks presented so far have exploited only a few of them.

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 bit 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. This particular property causes the hash function to have many easy-to-find collisions (called vanishing differentials) after a small number of subrounds within the *first* round. At the end of each subround, all the bits are shifted one bit position to the left.

We remark that both the R function and the S function are byte-oriented, that is, they update each of the 8 bytes in b separately. After the update, only *two* of the 8 bytes (B_0 and B_4) are modified, and the rest of the 6 bytes remain the same.

3 The attack of Biryukov, Lano, and Preneel

Biryukov, Lano, and Preneel recently presented a full key recovery attack that uses a single 2-bit vanishing differential. The attacker first guesses the subround N in which the vanishing differential occurs, and for each N a filtering algorithm is used to search the set of candidate keys that make such a vanishing differential possible. In [1], they only described the attack for $N = 1$ and stated that the algorithm would be similar for other N. According to their simulations, one only needs to do up to $N = 12$ to have a 50% chance of finding the key. Our own simulations suggest that $N = 16$ is a more accurate number. Although this discrepancy has little practical implications, we suggest a reason for it in section 8.1. For larger values of N, the cost of precomputation becomes prohibitive. On the other hand, it is expected that a hybrid algorithm between the attacks discussed here and [2] can be used to find more keys somewhat efficiently.

We now give a high-level description of the filtering algorithm for $N = 1$. Before the beginning, a table with entries of the form

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

is precomputed. The entries contain all combinations of key bit k_0 and data bytes B_0, B_4, B'_0, and B'_4 going into the first round (i.e. after the initial permutation) that will result in a vanishing differential at the end of the first subround. Note that none of the other data bytes have any involvement in the first subround, so whether a vanishing differential can happen or not for $N = 1$ is completely characterised by this table.

The filtering algorithm proceeds in four steps.

- *First Step.* For each entry in the precomputed table, try all possible values of $k_1, ..., k_{27}$. Together with k_0, 28 key bits are set, which determines 28 bits of b^0 from the initial key-dependent permutation. Since these bits overlap with the entries in the table by one nibble $\mathtt{B}_9$, key values that do not produce the correct nibble for both plaintexts in the vanishing differential are filtered out.

- *Second Step.* An entry that passes the first step is taken as input and the key bits $k_{28}, ..., k_{31}$ are guessed. Filtering is done based on the overlap in nibble $\mathtt{B}_8$.

- *Third Step.* Key bits $k_{32}, ..., k_{59}$ are guessed. Filtering is done based on the overlap in nibble $\mathtt{B}_1$.

- *Fourth Step.* Key bits $k_{60}, ..., k_{63}$ are guessed. Filtering is done based on the overlap in nibble $\mathtt{B}_0$.

Finally, each candidate key that passes the filtering steps is tested by performing a full hash function to see if it is the correct key.

As we can see, the running time of the above attack depends on the time complexity of each filtering step and the number of candidate keys that pass all four filtering steps. Based on simulation results [1], they estimated that the dominant factor is the third filtering step, which is equivalent to about 2^{48} full hash operations for N up to 12.

4 Improved analysis of the Biryukov, Lano, and Preneel attack

Biryukov, Lano, and Preneel only gave simulated results for $N = 1$. They suggested that

> for higher N, the overlap will be higher (because more bits play a role in the vanishing differential) and thus the filtering will be stronger.

and

> For $N > 1$, we expect the complexity of the attack to be lower due to stronger filtering.

Here we show that the results of their simulations can be justified by mathematical arguments, and that the conjecture of the filtering improving for larger N appears to be correct. We first analyse the case $N = 1$, and then generalise the argument to arbitrary N. Our analysis is an average-case analysis. The actual time complexity will depend upon the particular pair of plaintexts that caused the vanishing differential.

There is one subtlety that the reader should keep in mind in our analysis. During the first two filtering steps, only the values (k_0, B_4, B_4') of the precomputed table are involved. There may be more than one table entry overlapping in these values. In this case, we assume that the multiple entries are grouped together into a single entry until a later filtering step requires testing for the overlap separately. Since, as we will see, the number of multiple entries is very small, we assume that this does not incur a noticeable speed penalty.

4.1 Analysis of the attack for $N = 1$

Their simulations showed that the first step reduced the number of possibilities to 2^{27}, the second step further reduced the count to to 2^{25}, the third step increased the count to 2^{45}, and the fourth step resulted in 2^{41} true candidates. We analyse the second and fourth steps only: the other two can be analysed similarly.

Some properties of the precomputed table are necessary in the analysis. In [1], it is stated that the size of the precomputed table is 30 for $N = 1$, which we agrees with our computation. In Appendix A, we provide an analytical way of constructing the entries.

Analysis of the second step: We start by examing the precomputed table to count number of *unique* entries of the form (k_0, B_4, B_4'). In total, there are only 23, which is broken down into 7 with no difference, 16 with a 1-bit difference, and none with 2-bit differences.

There are a total of 2^{32} possible partial keys (each 32 bits) up to step two. Among them,

- A fraction of $\binom{56}{2}/\binom{64}{2} \approx .76$ will put no difference in the tuple (B_4, B_4').
- A fraction of $\binom{8}{1} \times \binom{56}{1}/\binom{64}{2} \approx .22$ will put a 1-bit difference in (B_4, B_4').
- A fraction of only $\binom{8}{2}/\binom{64}{2} \approx .01$ will put 2 difference bits in (B_4, B_4').

Of the $2^{32} \times 0.76$ keys that result in no difference in (B_4, B_4'), only a fraction of $\frac{7}{256}$ will match one of the 7 unique entries in the table for B_4 (which is the same as B_4'). Of those, only half will have the right key bit corresponding to what is stored for that entry of the table. Thus, the expected number of 32-bit keys resulting in no difference in B_4 that pass the second filtering step is

$$2^{32} \times 0.76 \times \frac{7}{256} \times \frac{1}{2} \approx 2^{25.4}.$$

For 1-bit differences, the calculation is similar, except we have 16 unique table entries, and there are 256×8 possible tuples (B_4, B'_4) with $B_4 \bigoplus B'_4$ differing in 1-bit. The expected number here is

$$2^{32} \times 0.22 \times \frac{16}{256 \times 8} \times \frac{1}{2} \approx 2^{21.8}.$$

For 2-bit differences, there are 0 in the table, so none of those will get through.

Combining these results, the expected number of 32-bit keys that pass through step 2 is

$$2^{25.4} + 2^{21.8} \approx 2^{25.5}$$

which closely agrees with the 2^{25} observed by simulation in [1].

Analysis of the fourth step: Without considering outcomes of previous steps, we can directly analyse the fourth step. This is because anything that matches an entry in the precomputed table will result in a vanishing differential for $N = 1$. In other words, the entries in the table are not only a necessary set of cases for a vanishing differential to occur at $N = 1$, but also sufficient. So, analysing the outcome of the fourth step is equivalent to determining the true number of candidates that need to be tested with the full SecurID hash function.

For each of the 30 table entries, we have:

- Only a portion of about $\frac{1}{2^{16}}$ of the 2^{64} keys will permute the bits of the first plaintext so that the bytes (B_0, B_4) match the table entry.
- Of those keys, only a portion of $1/\binom{64}{2}$ will permute the 2 difference bits in the right locations to match the (B'_0, B'_4) of that table entry.
- Only half of those keys will have the right key bit k_0 corresponding to what is in that entry of the table.

Thus, the expected number of final candidate keys is

$$30 \times 2^{64} \times \frac{1}{2^{16}} \times \frac{1}{\binom{64}{2}} \times \frac{1}{2} \approx 2^{40.9}$$

which is approximately 2^{41} that was observed in [1].

Another way of interpreting this result is that the probability of a randomly chosen 2-bit differential disappearing in subround 1 is $\frac{2^{40.9}}{2^{64}} \approx 2^{-23.1}$. This property will be useful in our later analysis.

4.2 Analysis of the attack for $N > 1$

Here we derive general formulas for the number of candidate keys that will pass the second and fourth steps, respectively, as well as the time complexity for the third step. As we discussed before, these are the dominating factors in estimating the running time of the attack. Similar to the case of $N = 1$, the formulas depend upon properties of the precomputed tables.

In the general case, the precomputed tables consist of the following entries:

- 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'), 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').

By legal values we mean that the combination of key bits and plaintext bits will cause the difference to vanish in subround N. The words W_0, W_0', W_4, W_4' each consist of $7+N$ bits and the number of key bits is N. Using this notation, observe that when $N = 1$ we have $(W_4, W_4') = (B_4, B_4')$ and $(W_0, W_0') = (B_0, B_0')$.

Analysis of the second step: Of the of 2^{32} key bits considered up to step 2,

- A fraction of $\binom{57-N}{2}/\binom{64}{2}$ will put no difference in the tuple (W_4, W_4').
- A fraction of $\binom{7+N}{1} \times \binom{57-N}{1}/\binom{64}{2}$ will put a 1-bit difference in (W_4, W_4').
- A fraction of only $\binom{7+N}{2}/\binom{64}{2}$ will put a 2-bit difference in (W_4, W_4').

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 $W_4 \bigoplus W_4'$ having hamming weight 1, and C_2 similarly except $W_4 \bigoplus W_4'$ having hamming weight 2. The expected number of keys causing no bit difference in (W_4, W_4') that will pass the filter in step two is:

$$2^{32} \times \frac{\binom{57-N}{2}}{\binom{64}{2}} \times \frac{C_0}{2^{7+N}} \times \frac{1}{2^N}$$

$$= 2^{19-2N} \times \frac{3192 - 113N + N^2}{63} \times C_0.$$

For 1-bit differences, the equation is

$$2^{32} \times \frac{\binom{7+N}{1} \times \binom{57-N}{1}}{\binom{64}{2}} \times \frac{C_1}{2^{7+N} \times \binom{7+N}{1}} \times \frac{1}{2^N}$$

$$= 2^{20-2N} \times \frac{57-N}{63} \times C_1.$$

For 2-bit differences, the equation is

$$2^{32} \times \frac{\binom{7+N}{2}}{\binom{64}{2}} \times \frac{C_2}{2^{7+N} \times \binom{7+N}{2}} \times \frac{1}{2^N}$$

$$= 2^{20-2N} \times \frac{C_2}{63}.$$

Hence, the expected number of candidates to pass the second step is then

$$T = \frac{2^{19-2N}}{63} \times \left[(3192 - 113N + N^2)C_0 + (114 - 2N)C_1 + 2C_2\right]. \quad (1)$$

In [1], the third step is the most time consuming. For each candidate that passes the second step, they must guess 28 bits of key and then perform a fraction of $\frac{28}{64}$ of the permutation for both plaintexts. Under the assumption that the permutation is 5% of the time required to do the full SecurID hash, the running time is equivalent to

$$T \times 2^{28} \times \frac{28}{64} \times 2 \times 0.05$$

full hash operations.

Note that when deriving the above formula, we assumed that for larger N, exactly four filtering steps (same as what was done when $N = 1$) were used. The filtering algorithm was not completely described for $N > 1$ in [1], but it is likely that they imagined that the number of filtering steps would increase. In particular, one may presume that the third step would involve guessing enough key bits so that the resulting permuted data bits just begin to overlap with W_0 and W_0', and an additional layer of filtering would be added for each key nibble guessed beyond that[1]. This speeds up the third step, which we will assume is still the most time consuming of the remaining filtering steps[2]. We proceed under this assumption.

In this way, the exact number of key bits guessed in the third step is $4 \times \lfloor \frac{29-N}{4} \rfloor$, and its running time is

$$T \times 2^{4 \times \lfloor \frac{29-N}{4} \rfloor} \times \frac{4 \times \lfloor \frac{29-N}{4} \rfloor}{64} \times 2 \times 0.05 \times s \quad (2)$$

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.

[1] The same idea should be applied to the first filtering step as well, but for the sake of brevity, we avoid making the description too complex.

[2] A sufficient but not necessary condition for the third step to be the most time consuming in the algorithm as it is stated 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. In the rare exceptions, the fourth step may be more time consuming. However, modifications to the fourth step can be made in order to speed it up significantly. For example, a large portion of false candidates can be eliminated in the fourth step by checking whether the hamming weights of the remaining bits for both plaintext pairs matches those of the table.

N	table size	C_0	C_1	C_2	T	Time for third step	Time for last step	Total time
1	30	7	16	0	$2^{25.5}$	$2^{47.6}$	$2^{40.9}$	$2^{47.6}$
2	350	24	128	84	$2^{25.4}$	$2^{44.2}$	$2^{41.5}$	$2^{44.4}$
3	2366	171	660	248	$2^{26.1}$	$2^{45.0}$	$2^{41.2}$	$2^{45.1}$
4	16784	1047	3778	1392	$2^{26.7}$	$2^{45.5}$	$2^{41.0}$	$2^{45.6}$
5	116184	6349	22700	8264	$2^{27.2}$	$2^{46.1}$	$2^{40.8}$	$2^{46.1}$
6	729236	37257	125824	42836	$2^{27.7}$	$2^{42.7}$	$2^{40.5}$	$2^{43.0}$

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

Analysis of the fourth step: Following section 4.1, the general formula for the number of final candidates is:

$$\text{table size} \times 2^{64} \times \frac{1}{2^{2N+14}} \times \frac{1}{\binom{64}{2}} \times \frac{1}{2^N} \approx 2^{39.0-3N} \times \text{table size}. \tag{3}$$

Combined analysis: The running time of algorithm [1] for a particular value of N is expected to be the approximately the sum of equations 2 and 3. For $N = 1..6$, these running times are given in Table 1.

Notice that even though the number of candidates T after the second filter steps are approximately the same as N goes from 1 to 2 and also from 5 to 6, the running times of the third steps drop greatly. 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 $2^{48.5}$ and larger values of N would appear to add minimally to this total.

5 Faster filtering

As illustrated in the previous section, the trick to speeding up the key recovery attack in [1] is faster filtering. We have found three ways in which their filtering can be sped up:

1. 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 amortize the cost of the permutation computations.

2. We can detect ahead of time when a large portion of keys will result in “bad” permutations in steps one and three, and the filtering process can skip past chunks of these bad permutations.

3. For $N = 1$, we can further speed up the third step of filtering by using a table-lookup to determine what the legal choices are for $\mathtt{K}_{14}$ (this would apply to other N as well, but the memory requirements quickly become quite large). Each table lookup replaces trying 8 choices for the nibble $\mathtt{K}_{14}$.

In what follows, we describe each of the above techniques in more detail.

The first 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 2. 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 step three 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 2 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 second speedup is dependent upon the first. It will apply to both the first and third filtering steps. During the process of trying a permutation, there will be large chunks of bad trial keys that can be identified immediately, and skipped. For example, consider $N = 1$ in the first filtering step. Whenever one of the difference bits is put into any of the bit indices 40..63 of the permuted data array, it can be skipped because the difference is not in a legal position. More generally, in the recursive procedure for key trials, we check during each trial key nibble whether it will result in a difference bit being put in an illegal place. If affirmative, then any key having the same most significant bits will also result in misplacing the difference bit, so the recursive branch for that key nibble can be skipped. This substantially reduces the number of trial keys. In the first step, this will skip past all but a fraction $\binom{39+N}{2}/\binom{64}{2}$ of the candidates. More importantly, between the first and third steps the amount of keys looked at in the search becomes a fraction $\binom{14+2N}{2}/\binom{64}{2}$ of the amount for the attack in [1].

These two strategies combined result in the following running time for the third filtering step:

$$T \times \frac{\binom{14+2N}{2}}{\binom{64}{2}} \times 2^{4\times\lfloor\frac{29-N}{4}\rfloor} \times \frac{1.07}{16} \times 2 \times 0.05 \times s \tag{4}$$

N	Time for third step	Time for last step	Total time
1	$2^{37.8}$	$2^{40.9}$	$2^{41.0}$
2	$2^{38.0}$	$2^{41.5}$	$2^{41.6}$
3	$2^{39.0}$	$2^{41.2}$	$2^{41.5}$
4	$2^{39.9}$	$2^{41.0}$	$2^{41.6}$
5	$2^{40.7}$	$2^{40.8}$	$2^{41.8}$
6	$2^{37.8}$	$2^{40.5}$	$2^{40.7}$

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

where T is still the T from equation 1 (though it no longer represents the number of candidates at the end of step two) and s is $\frac{96}{256}$ for $N = 1$, $\frac{12}{16}$ for $N = 2..5$, and 1 for all other values.

We only apply the third speedup for $N = 1$, due to increasing memory requirements. When we arrive at a leaf to try $\mathtt{K}_{14}$, there are only 8 data bits remaining to choose from. Let x represent the final 8 bits for the first plaintext, and x' for the second. We could precompute the legal choices for $\mathtt{K}_{14}$ for each possible (k_0, B_4, B_4', x, x') where (k_0, B_4, B_4') are from the 23 unique choices in the main filtering precomputation table. Thus the legal choices for $\mathtt{K}_{14}$ are obtained from a single table lookup, which replaces trying all possibilities. For $N = 1$, this gives a time of approximately:

$$T \times \frac{\binom{16}{2}}{\binom{64}{2}} \times 2^{24} \times \frac{1.07}{16} \times 2 \times 0.05 \times \frac{12}{16}.$$

The combined speedups give the run times in Table 2. In all cases, the third filtering step has become faster than the time for the last step. The total time for $N = 1..6$ is $2^{44.0}$, and larger values of N are expected to add minimally to it since all steps are getting faster.

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 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 only, 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 three filtering speedups and all filtering steps. Our code was written so that whenever a candidate past the first filtering step, it was immediately sent to the second. If a candidate past the second, then it immediately went to the third, and so on. So, we were doing a full key 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. According to Table 2, the running time is expected to be about $2^{37.8}$ hash operations, which should take about 2 weeks on our 2.4 GHz PC.

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 42.3 hours. If we pessimistically assume that the full search will take at most 2^4 times longer, the full running time would be 4 weeks, which is twice the expectations. To understand why we believe this is pessimistic, observe that the first difference is at bit index 15. So, whenever the first two key nibbles add up to a value between 16 and 19, the entire recursive branch corresponding to the second key nibble is skipped, according to our second filtering speedup. Since our search fixed the first key nibble at 3 at the second nibble went through values from 0 to 5, none of these big skips have happened yet. Thus, the second filtering speedup will become more effective during a full search. As further proof of this, we searched the space that had first key nibble set to `0xf` in 30.5 hours. We therefore have strong evidence that the implementation is close to the expectations from our analysis. Note also that since we are doing all filtering steps in this time, our results further substantiate the claim that the third filtering step is the dominant cost.

7 Vanishing differentials with $\geq$ four-bit differences

According to our simulations, about 25% of the first collisions (first occurrence of a vanishing differential for a given key) are actually from a 4-bit difference, and about 7% from larger differences. We would expect that our filtering algorithm performs exceptionally well in these circumstances. For example, consider 4-bit differences. When $N = 1$, we expect our second filtering speedup to skip all

N	table size	run time
1	910	$2^{37.5}$
2	9202	$2^{37.9}$
3	53358	$2^{37.4}$
4	311566	$2^{37.0}$

Table 3: Cost of the final step using a 4-bit differential for $N = 1..4$.

except a fraction of $\binom{16}{4}/\binom{64}{4} \approx 2^{-8.4}$ of the incorrect keys between filter steps one through three. Without going through the analysis, it seems reasonable to assume that the final testing of candidates is still the bottleneck.

The formula for number of final candidate keys for 4-bit differences can be derived similar to that of equation 3:

$$\text{table size} \times 2^{64} \times \frac{1}{2^{2N+14}} \times \frac{1}{\binom{64}{4}} \times \frac{1}{2^N}.$$

The formula is the same as that for 2-bit differences, except that term $\binom{64}{2}$ has been replaced by $\binom{64}{4}$, giving a factor of $2^{8.3}$ reduction in the number. Therefore, as long as the table size does not increase significantly, it is conceivable that 4-bit differentials could result in a faster attack than 2-bit differentials. In Table 3 we see that this is indeed the case for $N = 1..4$. The table size for $N = 1$ can also be verified analytically as described in Appendix A. We therefore conjecture that the total run time for an attack using one 4-bit vanishing differentials is equivalent to about 2^{40} hash operations, and for larger differentials, the algorithm should improve more.

Note that for $N = 1$, we have a probability of $\frac{2^{37.5}}{2^{64}} = 2^{-26.5}$ for a 4-bit vanishing differential to occur, and the corresponding probability for a 2-bit vanishing differential is $2^{-23.1}$. It may then seem hard to believe that 25% of the vanishing differentials are 4-bits, as claimed above. However, one should keep in mind that there are more input 4-bit differences because the least significant byte of the time is replicated 4 times in the time expansion function.

8 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.

8.1 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 corresponding to plaintexts B and B', which come 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 9.

Following the above thought process, it is evident that:

- Flipping time bits in T_1, T_1' or T_2, T_2' 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.[3]
- 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 collision usually happens early (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:

- Same input difference for all vanishing differentials.

[3] We suspect that this is the reason for the discrepancy between the paper [1] claiming that $\geq 50\%$ of the collisions happened in 12 subrounds and our own simulations which suggest 16 subrounds. Specifically, their data probably included multiple collisions, which made it biased towards a smaller number.

N	Time for last step using only a single collision	Time for last step using $z = 2$	Time for last step using $z = 4$	Time for last step using $z = 8$
1	$2^{40.9}$	$2^{40.1}$	$2^{39.2}$	$2^{37.3}$
2	$2^{41.5}$	$2^{40.5}$	$2^{39.5}$	$2^{37.4}$
3	$2^{41.2}$	$2^{40.1}$	$2^{39.0}$	$2^{36.6}$
4	$2^{41.0}$	$2^{39.8}$	$2^{38.5}$	$2^{35.8}$
5	$2^{40.8}$	$2^{39.4}$	$2^{38.0}$	$2^{35.0}$
6	$2^{40.5}$	$2^{39.0}$	$2^{37.4}$	$2^{34.0}$

Table 4: Time for the last step assuming the attacker became aware of z-bits that do not get permuted into words W_0, W_0', W_4, or W_4'.

- All input times differ in only a few bits.
- 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 second 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 4 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 9.

8.2 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 sketch for the new filtering algorithm when $N_1 = N_2 = 1$. Other cases can be handled similarly.

- *First Stage.* Take V_1 and guess the first 32 bits of the key. For each 32-bit key that produces a valid (B_4, B_4'), test it against V_2 to see if it also produces a valid (B_4, B_4'). (This is the first and the second filtering steps in the original attack.)
- *Second Stage.* For 32-bit keys that pass the above stage, do the same thing to guess the second 32 bits of the key. (This is the third and the fourth filtering steps in the original attack.)

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.

Based on analysis in Section 4, we know that the probability that a 32-bit key passes the first stage is $2^{25.5}/2^{32} = 2^{-6.5}$ (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 stages is $2^{40.9}/2^{65} = 2^{-23.1}$. If the two vanishing differentials are indeed *independent*, we would expect the number of keys to pass the first filtering to be

$$2^{32} \times 2^{-6.5} \times 2^{-6.5} = 2^{19}, \text{and}$$

and the number of keys to pass both filterings to be

$$2^{64} \times 2^{-23.1} \times 2^{-23.1} = 2^{17.8}.$$

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.

9 Using non-vanishing differentials with a vanishing differential

In Section 8.1, 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 were unable to find a simple formula that represents this probability. Fortunately, there is a wonderful computer algebra package known as Magma [5] that can be used to evaluate seemingly complex probabilities.

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:

$$\sum_{\text{all valid } c_r,\ 2 \leq r \leq 13} \frac{\prod_{i=2}^{7} \binom{4}{c_i} \times \prod_{i=8}^{13} \binom{2}{c_i} \times \binom{28}{2N+14-\sum_{i=2}^{13} c_i}}{\binom{64}{2N+14}} \tag{5}$$

where *valid* c_r 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 code that is given in Appendix C. The probabilities, and corresponding running time for the testing of final candidates are given in Table 5. 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 8.1 that when a collision happens early, other collisions are likely to follow soon after. Note that Table 5 also tells us that if we only have a single vanishing differential within a period of about a week or more, it is probably best to try key recovery by guessing higher values of N first, in order to minimise the expected run time.

One should not assume that the times for the last step given in Table 5 are the dominant cost in applying this strategy. Unlike the filtering speedups given in Sections 5 and 8.1, 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.

10 The threat of token reassignment

In the token reassignment scenario, we are concerned with a user who has had his token for a long period of time (example: a year or more), and has attempted to

N	Fraction of keys having property	Time for last step
1	$2^{-14.3}$	$2^{26.6}$
2	$2^{-11.7}$	$2^{29.8}$
3	$2^{-9.7}$	$2^{31.5}$
4	$2^{-8.1}$	$2^{32.9}$
5	$2^{-6.7}$	$2^{34.1}$
6	$2^{-5.7}$	$2^{34.8}$

Table 5: 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.

find the secret key before returning it to the company. In 100 randomly chosen key simulations consisting of 500 days of token outputs, we found:

- 46 keys had at least one vanishing differential.
- Among keys that did have a vanishing differential, the average number of vanishing differentials was 8.9.
- Only 13 of the 46 had a single vanishing differential. Of them, the least number of subrounds that the collision occurred in was 13 (supporting the analysis in Section 9).
- Of the 33 cases with multiple vanishing differentials, there were 11 that involved at least one instance of different differences. 5 of those met the requirement of having both differences within 16 subrounds, so that the algorithm from Section 8.2 would succeed.
- 23 keys had at least one vanishing differential within 16 subrounds, guaranteeing that the key could be found by one of our algorithms.

It is therefore quite evident that a significant fraction of users will be able to find the keys within their cards, assuming they recorded all data outputs. The user will have very high confidence in his success if he witnesses multiple vanishing differentials with the same difference, as described in Section 8.1. If the user only witnessed a single vanishing differential, he can apply the attack from Section 9. As long as the collision happened within about 16 subrounds, he will be able to discover the key. Therefore, reassigning a token to a new user is a serious security risk.

11 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 a factor of 16, giving an improved attack with time complexity about 2^{44} hash operations. In practice, however, the attacker would typically have more information than just a single vanishing differential. Using this extra information appears to reduce the time down to 2^{40} hash operations or lower, making the attack possible by anybody with a modern PC.

The attacks in this paper and in [1] are real. The main obstacle in mounting them is waiting for an internal collision. If the user's token is out of his control for a matter of a few days, then the chances of the collision happening are small, but not negligible. This means that most attackers will not have the opportunity for success, but some will. The attacks also show that once a SecurID card is assigned to a user, he or she has the ability to find the secret key with high probability. Thus, reassignment of tokens to other users is a very bad idea.

In contrast, an AES-based hash function ought to prevent any attacker from having a realistic chance of success. We therefore recommend that all SecurID cards containing the alleged hash function be replaced with RSA Security's newer, AES-based hash.

References

[1] A. Biryukov, J. Lano, and B. Preneel. *Cryptanalysis of the Alleged SecurID Hash Function*, http://eprint.iacr.org/2003/162/, 12 Sep, 2003.

[2] S. Contini, *The Effect of a Single Vanishing Differential in ASHF*, sci.crypt post, 6 Sep, 2003.

[3] 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.

[4] *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 .

[5] *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.

$$\begin{aligned} B'_4 &= ((((B_0 >>> 1) - 1) >>> 1) - 1) \oplus B_4, \\ B'_0 &= 100 - B_4. \end{aligned} \tag{6}$$

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).

Rewrite Equation 6, 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). For each possible value of (B_4, B'_4), we enumerate the possible values of (B_0, B'_0) as follows.

- 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

- 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

Combining the above two cases, we have $8 + 7 = 15$ 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 $2 \times 15 = 30$ entries. These entries match the results from our simulation.

Similar analysis also confirms that the size of the table for 4-bit vanishing differentials when $N = 1$ is 910.

B Example of multiple vanishing differentials

Table 6 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 4. Taking into consideration $N = 2$, the total time is expected to be on the order of 2^{38} operations.

C Magma code

Below is the Magma code that was used to compute the probabilities from Section 9.

```
/* this code assumes  j >= 7 */
j := 13;
cardinality_sets := 24 + (j-7)*2;
for N in [1..6] do
    w0w4size := 2*N + 14;
    c := [1: i in [2..j]];
    sum := 0.0;
    done := false;
    while not done do
        bits_taken := &+c;
```

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

Table 6: Example of 16 vanishing differentials that happened within 1.3 days, using key `b5 a9 f4 8c 16 23 a6 1a`.

```
if bits_taken le w0w4size then
    prod1 := &*[ Binomial( 4, c[i] ) : i in [1..6] ];
    if j gt 7 then
        prod2 := &*[ Binomial( 2, c[i] ) : i in [7..j-1] ];
    else
        prod2 := 1;
    end if;
    prod3 := Binomial( 64 - cardinality_sets, w0w4size - bits_taken );
    sum +:= prod1*prod2*prod3;
end if;
index := 1;
c[index] +:= 1;
while ((index le 6 and c[index] eq 5) or
  (index ge 7 and c[index] eq 3)) do
    /* there is a 'carry' in the counter */
    c[index] := 1;
    index +:= 1;
    if index ge j then
        done := true;
        break;
    end if;
    c[index] +:= 1;
```

```
        end while;
    end while;

    prob := sum/Binomial(64, w0w4size );
    print "Probability is 2^",Log(prob)/Log(2), "for N = ",N;
end for;
```