# Herding Hash Functions and the Nostradamus Attack

John Kelsey[1] and Tadayoshi Kohno[2]

[1] National Institute of Standards and Technology, john.kelsey@nist.gov
[2] CSE Department, UC San Diego, tkohno@cs.ucsd.edu

**Abstract.** In this paper, we develop a new attack on Damgård-Merkle hash functions, called the *herding attack*, in which an attacker who can find many collisions on the hash function by brute force can first provide the hash of a message, and later "herd" any given starting part of a message to that hash value by the choice of an appropriate suffix. We focus on a property which hash functions should have–Chosen Target Forced Prefix (CTFP) preimage resistance–and show the distinction between Damgård-Merkle construction hashes and random oracles with respect to this property. We describe a number of ways that violation of this property can be used in arguably practical attacks on real-world applications of hash functions. An important lesson from these results is that hash functions susceptible to collision-finding attacks, especially brute-force collision-finding attacks, cannot in general be used to prove knowledge of a secret value.

## 1 Introduction

Cryptographic hash functions are usually assumed to have three properties: Collision resistance, preimage resistance, and second preimage resistance. And yet many additional properties, related to the above in unclear ways, are also required of hash function in practical applications. For example, hash functions are sometimes used in "commitment" schemes, to prove prior knowledge of some information, priority on an invention, etc. When the information takes on more than a small number of possible values, there does not appear to be an obvious way to extend a collision finding attack to break the commitment scheme; therefore, collision resistance does not seem to be necessary to use the hash function in this way. This appears fortunate in light of the many recent attacks on collision resistance of existing hash functions[2, 3, 13, 19, 21–24] and the widespread use of hash functions short enough to fall to brute-force collision attacks[20].

We show that the natural intuition above is incorrect. Namely, we uncover (what we believe to be) subtle ways of exploiting the iterative property of Damgård-Merkle[6, 16] hash functions to extend certain classes of collision-finding attacks against the compression function to attack commitment schemes and other uses of hash function that do not initially appear to be related to collision resistance.

### 1.1 Example: Proving Prior Knowledge with a Hash Function

Consider the following example. One day in early 2006, the following ad appears in the *New York Times*:

> I, Nostradamus, hereby provide the MD5 hash $H$ of many important predictions about the future, including most importantly, the closing prices of all stocks in the S&P500 as of the last business day of 2006.

A few weeks after the close of business in 2006, Nostradamus publishes a message. Its first few blocks contain the precise closing prices of the S&P500 stocks. It then continues with many rambling and vague pronouncements and prophecies which haven't come true yet. The whole message hashes to $H$.

The main question we address in this paper is whether this should be taken as evidence that Nostradamus really knew the closing prices of the S&P500 many months in advance. MD5 has been the subject of collision attacks, and indeed is susceptible to brute force collision attacks, but there are no known preimage attacks. And yet, it seems that a preimage attack on MD5 would be necessary to allow Nostradamus to first commit to a hash, and then produce a message which so precisely describes the future after the fact.

### 1.2 Chosen Target Forced Prefix (CTFP) Preimage Resistance

The first question to address when considering the situation outlined above is to ask exactly what property of a hash function would have to be violated by Nostradamus in order to falsely "prove" prior knowledge of these closing prices. The property is not directly one of the commonly discussed properties of hash functions (collision resistance[3], preimage resistance, and second preimage resistance). Instead, we need an atypical property, which we will call "chosen target forced prefix" (CTFP) preimage resistance[4].

In order to falsely prove his knowledge of the closing prices of the S&P500, Nostradamus would first have to choose a target hash value, $H$. He then would have to wait until the closing values of the S&P500 stocks for 2006 were available. Finally, he would have to find some way to form a message that started with a description of those closing values, $P$, and ended up with the originally committed-to hash $H$.

Following this example, we can define CTFP preimage resistance as follows: In the first phase of his attack Nostradamus performs some precomputation and then outputs an $n$-bit hash value $H$; $H$ is his "chosen target". The challenger

[3] Collision resistance would preclude the attack, but does not appear to be necessary for the attack to fail.

[4] We are indebted to Dan Brown for pointing out a previous use of the same idea: In one of three independent proofs of the security of Pinstov-Vanstone signatures, the same property with a different name, "target value resistance," was used. See [4], in which it was conjectured that SHA1 had this property; our result shows that it does not if one can find collisions starting from two arbitrary IVs.

then selects some prefix $P$ and supplies it to Nostradamus; $P$ is the "forced prefix." In our informal security definition we place no restriction on how the challenger picks $P$, but for simplicity we may assume that the challenger picks $P$ uniformly at random from some large but finite set of strings. In the second phase of his attack, Nostradamus computes and outputs some string $S$. Nostradamus compromises the CTFP preimage resistance of the hash function if $\mathsf{hash}(P\|S) = H$. If we model the hash function as a random oracle [1], then unless Nostradamus is lucky and guesses $P$ in the first phase of his attack, we would expect him to have to try $O(2^n)$ values for $S$ in the second phase before finding one such that $\mathsf{hash}(P\|S) = H$. Consequently, it might seem reasonable to expect that Nostradamus would have to perform $O(2^n)$ hash function computations to compromise the CTFP preimage resistance of a real hash function. (While one could consider a more formal definition of CTFP for hash function families, and consider the relationship between CTFP-resistance and other security goals, we do not do so here but instead focus on our attacks.)

As described in detail below, the ability to violate the CTFP preimage resistance property allows an attacker to carry out a number of surprising attacks on applications of a hash function. Almost any use of a hash function to prove knowledge of some information can be attacked by someone who can violate this property. Many applications of hashing for signatures or for fingerprinting some information which are not vulnerable to attack by straightforward collision-finding techniques are broken by an attacker who can violate CTFP preimage resistance.

Further, when the CTFP definition is relaxed somewhat (for example, by allowing Nostradamus some prior limited knowledge or control over the format of $P$, giving him prior knowledge of the full (large) set of possible $P$ strings that might be presented, or allowing him to use any of a large number of encodings of $P$ with the same meaning), the attacks become still cheaper and more practical.

### 1.3 Herding Attacks

The major result of this paper is as follows: For Damgård-Merkle[6, 16] construction hash functions, CTFP preimage resistance can always be violated by repeated application of brute-force collision-finding attacks. More efficient collision-finding algorithms for the hash function being attacked may be used to make the attack more efficient, if the details of the collision-finding algorithms support this. An attack that violates this property effectively "herds" a given prefix to the desired hash value; we thus call any such attack violating the CTFP preimage resistance property a "herding attack."

The herding attack shows that the CTFP preimage resistance of a hash function like MD5 or SHA1 is ultimately limited by the collision resistance of the hash function. At a high level, and in its basic variant, the attack is parameterized by some positive integer $k$, e.g., $k = 50$, and by the output size $n$ of the hash function. In the first phase of a herding attack, the attacker, Alice, repeatedly applies a collision-finding attack against a hash function to build a *diamond structure*, which is a data structure reminiscent of a binary tree. With high probability it takes at most $2^{k/2+n/2+2}$ applications of the hash compression function (and possibly fewer, depending on details of more efficient collision-finding attacks[5]) to create a diamond structure with $2^{k+1}-2$ intermediate hash states, of which $2^k$ are used in the basic form of the attack. In the second phase of the attack, Alice exhaustively searches for a string $S'$ such that $P\|S'$ collides with one of the diamond structure's intermediate states; this step requires trying $O(2^{n-k})$ possibilities for $S'$. Having found such a string $S'$, Alice can construct a sequence of message blocks $Q$ from the diamond structure, and thus build a suffix $S = S'\|Q$ such that $\mathsf{hash}(P\|S) = H$; this step requires a negligible amount of work, and the resulting suffix $S$ will be $k+1$-blocks long. We stress that Alice can have significant control over the contents of $S$, which means that $S$ may not be "random looking" but may instead contain structured data suitable for the application that Alice is trying to attack. Table 1 present some parameters for a version of our attack.

**Table 1.** Herding with Short Suffixes

| output size | example | diamond width(k) | suffix length (blocks) | work |
|---|---|---|---|---|
| 128 | MD5 | 41 | 48 | $2^{87}$ |
| 160 | SHA1 | 52 | 59 | $2^{108}$ |
| 192 | Tiger | 63 | 70 | $2^{129}$ |
| 256 | SHA256 | 84 | 92 | $2^{172}$ |
| 512 | Whirlpool | 169 | 178 | $2^{343}$ |
| $n$ | | $(n-5)/3$ | $k+\lg(k)+1$ | $2^{n-k}$ |

### 1.4 Practical Impact

Our techniques for carrying out herding attacks have much in common with the long message second preimage attacks of [12]. However, those attacks required implausibly long messages, and so probably could never be applied in practice. By contrast, our herding attacks require quite short suffixes, and appear to be practical in many situations. Similarly, many recent cryptanalytic results on hash functions, such as [22, 23], require very careful control over the format of the messages to be attacked. This is not generally true of our herding attacks, though more efficient variants that make use of cryptanalytic results on the

[5] The collision finding attacks needed for constructing the diamond structure are somewhat different than those in recent results on MD5, SHA0, and SHA1[22, 23]. We are uncertain whether these attacks can be adapted to the requirements of constructing the diamond structure, though it seems plausible that it might work. For the diamond structure we need collisions between two messages starting with different IVs.

underlying hash functions will naturally have to follow the same restrictions as those attacks.

Near the end of this paper, we describe a number of ways in which our herding attacks and variations on them can be exploited. In developing the herding attack, we also describe a new method of building multicollisions for Damgård-Merkle hash functions which we believe to be of independent interest, and which may be useful in many other hash function attacks.

### 1.5 Related Work

The herding attack is closely related to the long message second preimage attacks in [8] and [12], and is ultimately built upon the multicollision-finding technique of [10]. Our technique for herding is related to the result of Lai and Massey [14] showing a meet-in-the-middle second preimage attack when pseudopreimages can be found cheaper than exhaustive search; in our attack, instead of finding pseudopreimages, we construct a message by repeated collision searches, and then do a meet-in-the-middle type attack to find a large set of possible second preimages on our own chosen message. Our results complement Coron, Dodis, Malinaud, and Puniya's work[5], which does not present attacks like the ones we present, but which shows that iterative hash functions like MD5 and SHA1 are not random oracles, even when their compression functions are. Variants of our attacks works against Coron, et al's fixes but do not violate their provable security bounds.

More broadly, our result re-enforces the lessons that might sensibly be taken from [7, 10–12, 15] on the many ways in which seemingly impractical hash function collisions may be applied in practice. The security properties of Damgård-Merkle hash functions against attackers who can find collisions are currently not well understood.

## 2 The Diamond Structure: A Building Block for Herding

In this section we introduce the *diamond structure*. This is a structure of messages constructed to produce a large multicollision of a quite different format than that of Joux[10]. Our multicollision is more expensive, and the same length. For example, a $2^k$ diamond-structure multicollision costs about $2^{n/2+k/2+2}$ work, relative to Joux' $k \times 2^{n/2}$ work. There are two reasons why the diamond structure lets an attacker do things which are not possible with only a Joux multicollision:

1. The diamond structure allows $2^k$ choices for the first block of a $2^k$ multicollision, whereas Joux multicollisions involve a sequence of pairs of choices for each part of the message.
2. The diamond structure contains $2^{k+1} - 2$ intermediate hash values, making the herding attack possible with short suffixes.

A diamond structure is essentially a Merkle tree built by brute force.

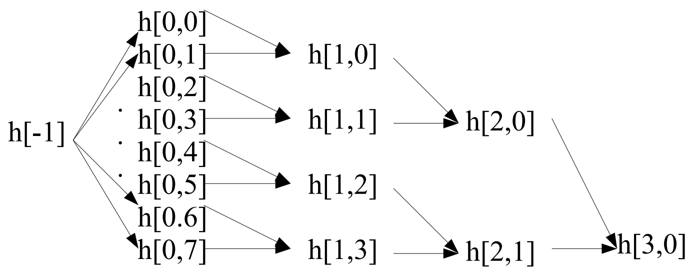


**Fig. 1.** The Basic Diamond Structure

Figure 1 describes the basic idea, where edges represent messages and values like $h[i, j]$ represent intermediate hash states. In the diagram, the attacker starts with eight different first message blocks, each leading to a different hash value; he then searches for collisions between pairs of these hash values, yielding four resulting intermediate hash values (at the cost of about $8 \times 2^{n/2}$ work using a naive algorithm). He repeats the process with the four remaining values, then the two remaining ones. The result is a diamond structure which is $2^k$ states wide, and contains $2^{k+1} - 1$ states total.

**Producing a Suffix from an Intermediate Hash Value.** Consider any of the starting hash values. A suffix which maps that hash value to the final hash $H$ is constructed by walking down the tree from the leaves to the root, appending the message blocks from each edge in the tree to produce a suffix.

Consider any intermediate hash value. Similarly, walking from that node down to the root of the tree yields a suffix which maps the intermediate hash value to the final hash $H$. Subsequently we discuss how to augment the suffix if the hash function includes the length of the message in its last block.

**Building the Structure.** Building the structure is more efficient than a naive approach suggests. Instead of fixing the position of each node within the tree and then searching for collisions, the attacker dynamically builds the tree structure during the collision search. To map $2^k$ hash values down to $2^{k-1}$, she generates about $2^{n/2+1/2-k/2}$ candidate message blocks from each starting hash value in a single level of the structure, and then finds collisions between the different starting values dynamically. The total work done to reduce $2^k$ hash values to $2^{k-1}$ is about $2^{n/2+k/2+1/2}$, and thus the work done to construct a full diamond structure with $2^k$ hash values at its widest point is about $2^{n/2+k/2+2}$.

The work done to build the diamond structure is based on how many messages must be tried from each of $2^k$ starting values, before each has collided with at least one other value. Intuitively, we can make the following argument, which matches experimental data for small parameters: When we try $2^{n/2+k/2+1/2}$ messages spread out from $2^k$ starting hash values (lines), we get $2^{n/2+k/2+1/2-k}$ messages per line, and thus between any pair of these starting hash values, we expect about $(2^{n/2+k/2+1/2-k})^2 \times 2^{-n} = 2^{n+k+1-2k-n} = 2^{-k+1}$ collisions. We thus expect about $2^{-k+k+1} = 2^1 = 2$ other hash values to collide with any given starting hash value.

If this search is done on a single processor, then each time a pair of lines collide, no further searching is done from those lines. There may be cases where two pairs of lines collide on the same hash value. This very slightly decreases the number of reachable hash values, but the expected number of these is extremely small. For example, in a $2^{55}$ diamond structure, there are about $2^{56}$ intermediate hashes which are the results of these collision searches. For a 160-bit hash, we thus expect roughly $2^{-49}$ such collisions, so we can ignore the effect of them on our result.

**Parallelizeability.** It is easy to adapt the parallel collision search algorithm of [20] to the construction of a diamond structure. The result of each iteration of the search algorithm yields both a seed for the next message block to try, and also a choice of which of the $2^k$ starting chaining values will be used.

**Employing Cryptanalytic Attacks.** The above discussion has focused on brute-force search as a way to build the diamond structure. An alternative is to use some cryptanalytic results on the hash function. Whether this will work depends on details of the cryptanalysis:

1. A collision-finding algorithm which produces a pair of messages from the same initial value is not useful in constructing the diamond structure. Similarly, an algorithm that can find collisions only from initial chaining values with a single difference is not useful.
2. An algorithm which works for any known IV difference can be directly applied to build the diamond structure, though one must fix the positions of the nodes within the diamond structure in advance. If the work to find a collision pair is $2^w$, then this algorithm should be used to reduce $2^k$ lines of hash values to $2^{k-1}$ lines so long as $w + k - 1 < n/2 + k/2 + 1/2$.
3. An algorithm which works for a subset $2^{-p}$ of all pairs of IVs can be used to construct the diamond structure if the pairs can be recognized efficiently. This is done by inserting one extra message block at each layer of the diamond structure, and using this to force selected pairs of lines to initial values from which the collision-search algorithm will work. The work necessary to find one collision between lines is now $2^{p/2+1} + 2^w$. This algorithm should be used to reduce $2^k$ lines to $2^{k+1}$ so long as $\lg(2^{p/2+1} + 2^w) + k - 1 < n/2 + k/2 + 1/2$.

**Expandable Messages.** Using the notation from [12], an $(a, b)$-expandable message is a set of messages of varying lengths, between $a$ and $b$ inclusive, all of which yield the same intermediate hash. Expandable messages may be found from any initial hash value using the techniques found in [12], and more efficiently found for some hash functions, including MD5 an SHA1, using techniques from [8]; in the latter case, the cost is around twice that of a brute-force collision finding attack.

If all $2^{k+1} - 2$ intermediate hash values from the diamond structure are used in the later steps of herding, then a $(1, k + 1)$-expandable message must be produced at the end of the diamond structure, to ensure that the final herded message is always a fixed length. This is necessary since we assume that the length of the message will be included in the last block. If only the widest layer of $2^k$ hash values is used, no expandable message is required.

**Precomputation of the Prefix.** If the full set of prefixes are known and small enough, the diamond structure can be computed from their resulting intermediate hashes. This follows from the fact that the starting hash values are arbitrary. This is discussed at more depth in Sections 3.1 and 4.

**Variant: The Elongated Diamond Structure.** Using ideas from [12], long messages offer a naive way to mount the attack; the diamond structure offers much shorter suffixes. However, the attacker can build a diamond structure with many intermediate hashes more cheaply than above, if she is willing to tolerate unreasonably long messages.

The widest layer of the diamond structure is chosen, with $2^k$ hash values. Then, the attacker computes $2^r$ message blocks for each of the $2^k$ hash values, thus producing a total of $2^{k+r}$ reachable intermediate states. He then constructs the collision tree as described above.

The total work done to build a $2^r$-long elongated diamond structure with $2^k$ values at its widest point is about $2^{r+k}+2^{k/2+n/2+2}$; this structure contains $2^{k+r}$ intermediate hash values, and yields suffixes of about $2^{r-1}$ message blocks on average. In general, for reasonable suffix lengths, the elongated diamond structure has only a small advantage over regular diamond structures. An elongated diamond structure must have an $(r, 2^r + r)$-expandable message appended to its end, to ensure that the final herded messages are always the same length, and so always have the same final hash value.

It is possible to parallelize much of the production of an elongated diamond structure. If the width is $2^k$ hash values at the beginning, then the construction of the structure can be parallelized up to $2^k$ ways.

## 3 How to Herd a Hash Function

The herding attack allows an attacker to commit to the hash of a message she doesn't yet fully know, at the cost of a large computation. This attack is closely related to the long message second-preimage attacks of [8, 12] and the multicollision-finding techniques of [10].

At a high level, the attack works as follows:

1. *Build the Diamond Structure:* Alice produces a search structure which contains many intermediate hash values. From any of these intermediate hash values, a message can be produced which will lead to the same final hash $H$. Alice may commit to $H$ at this point.
2. *Determine the Prefix:* Later, Alice gains knowledge of $P$.
3. *Find a Linking Message:* Alice now searches for a single-block which, if appended to $P$, would yield an intermediate hash value which appears in her search structure.

4. *Producing the Message:* Finally, Alice produces a sequence of message blocks from her structure to link this intermediate hash value back to the previously sent $H$.

At the end of this process, Alice has first committed to a hash $H$, then decided what message she will provide which hashes to $H$ and which begins with the prefix $P$.

**Building the Diamond Structure.** This step is described in Section 2.

**Finding a Linking Message.** Once a diamond structure is constructed and its hash $H$ is committed to, the attacker learns the prefix $P$. She must then find a linking message–a message which allows her to link the prefix $P$ into the diamond structure. See Figure 2. When there are $2^k$ intermediate hash values in the diamond structure, the attacker expects to try about $2^{n-k}$ trial messages in order to find a linking message.

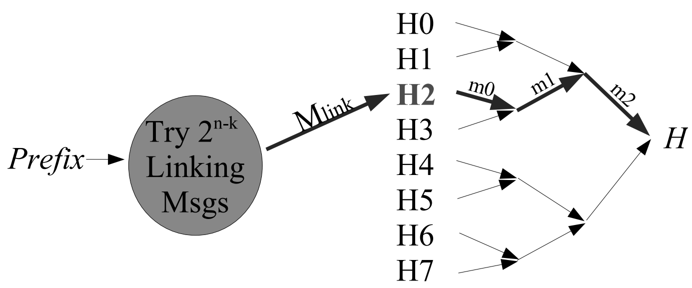


**Fig. 2.** Finding a Linking Message and Producing the Suffix

The starting chaining values for the diamond structure can be chosen arbitrarily. This makes it easy to parallelize the search for linking messages when herding a prefix into the first (widest) layer of the diamond structure. For example, the starting chaining values may be chosen to have their low 64 bits all zeros[18]; then each processor searching for a linking message need only check the list of starting hash values about once per $2^{64}$ trials.

**Producing the Message.** Once a linking message from $P$, $M_{link}$, is found, the suffix is produced as described above–basically, the attacker walks up the tree from the linked-to hash value to the root, producing another message block on each step. See Figure 2. If all $2^{k+1}-2$ intermediate hash values from the diamond structure are used when finding $M_{link}$, then the pre-determined expandable message must be appended to the end of the suffix.

### 3.1 Work Done for Herding Attacks

A maximally short suffix for the herding attack is found by producing a $2^k$ hash value wide diamond structure, and only searching for linking messages to the outermost (widest) level of hash values in the diamond structure, so that no expandable message is needed. In this case, the length of the suffix is $k+1$ message blocks, and the work done for the herding attack is approximately

$$2^{n-k} + 2^{n/2+k/2+2} \; . \tag{1}$$

Searching for linking messages to all $2^{k+1}-2$ intermediate hashes in the structure requires adding an additional $\lg(k)+1$ message blocks for a $(\lg(k), k+lg(k))$-expandable message, and decreases the work required to

$$2^{n-k-1} + 2^{n/2+k/2+2} + k \times 2^{n/2+1} \; , \tag{2}$$

the $k \times 2^{n/2+1}$ term arising from the search for an expandable message[12].

The cheapest herding attack with a reasonably short suffixes can be determined by setting the work done for constructing the diamond structure and finding the linking message equal. We thus get a diamond structure of width $2^k$, suffix length $L$, and total work $W$, where:

$$k = \frac{n-5}{3} \tag{3}$$

$$L = \lg(k) + k + 1 \tag{4}$$

$$W = 2^{n-k-1} + 2^{n/2+k/2+2} + k \times 2^{n/2+1} \approx 2^{n-k} \; . \tag{5}$$

Thus, using a 160-bit hash function, the cheapest attack with a reasonably short suffix involves a diamond structure with about $2^{52}$ messages at its widest point, producing a 59-block suffix, and with a total work for the attack of about $2^{108}$ compression function calls. See Table 1 for additional examples.

**Work for Herding Attacks with the Elongated Diamond Structure.** The cheapest herding attack with a suffix of slightly more than $2^r$ blocks can be determined by once again setting the work done for constructing the diamond structure and finding the linking message equal, so long as $k + r < k/2 + n/2$. We thus get an elongated diamond structure of width $2^k$, suffix length $L$, and total work $W$, where:

$$k = \frac{n-2r-3}{3} \tag{6}$$

$$L = \lg(k+2^r) + k + 1 + 2^r \tag{7}$$

$$W = 2^{n-k-r} + 2^{n/2+k/2+2} + k \times 2^{n/2+1} + 2^{k+r} \approx 2^{n-k-r+1} \; . \tag{8}$$

Thus, with a 160-bit hash function and a $2^{55}$ block suffix (about as long as is allowed for SHA1 or RIPEMD-160), an attacker would end up doing about $2^{90}$ work total to herd any prefix into the previously published hash value.

**Work for Herding from Precomputed Prefixes.** If the set of possible prefixes contains $2^k$ possible messages, the diamond structure can be built from the resulting $2^k$ intermediate hashes. In this case, there is no search for a linking message, and the total work for the attack is done in building the diamond structure.

### 3.2 Making Messages Meaningful

These attacks all involve producing a suffix to some forced prefix, which forces the complete message to have a specific hash value $H$. In order to use herding in a real deception, however, the attacker probably cannot just append a bunch of random blocks to the end of her predictions or other messages. Instead, she needs to produce a suffix which is at least somewhat meaningful or plausible. There are a number of tricks for doing this.

**Using Yuval's Trick.** Using Yuval's clever trick[25], the attacker can prepare a basic long document appropriate to her intended deception, and produce many independent variation points in the document. This allows the use of meaningful-looking messages for most contexts. For example, each message block in layer $i$ of the diamond structure could be a variation on the same theme, using about $n/2$ possible variation points. In practice, this likely will make the suffix longer, since it is hard to put 80 variation points in a 64-character message. However, this has almost no effect on the herding attack. If the attacker needs ten message blocks (640 characters) for each collision, her suffixes will be ten times longer, but no harder to find. The algorithm for finding them works the same way.

The contents of these suffixes must be pretty general. The natural way to handle this in most applications of herding is to write some common text discussing how the results are supposed to have been obtained ("I consulted my crystal ball, and spent many hours poring over the manuscripts of the ancient prophets...."). These can then be varied at many different points, independently, to yield many possible bitstrings all having the same meaning.

**Committing to Meaning, Not Bits.** For many of the attacks for which herding is useful, the goal is to falsely commit to some actual meaning, not necessarily some specific message string. For example, an attacker trying to prove her ability to predict the stock market is not really forced to use any fixed format for the contents of her stock market predictions, so long as anyone reading them will unambiguously be able to tell whether she got her predictions right.

This provides a great deal of extra flexibility for the attacker in using Yuval's trick, and also in arranging the different parts of the message to be committed to, in order to maximize her convenience.

## 4 Exploiting Prior Knowledge of the Prefix Space

As suggested in Sections 2 and 3.1, the attack becomes much more efficient if the prefix can be precomputed. In fact, it is often possible to precompute the message piecemeal in ways that leave a huge number of possible prefixes available, without requiring a huge amount of work.

Just as with the full herding attack, the precomputed version would not be useful against a random oracle–we make use of the iterative structure of existing hash functions to make the attack work.

**Precomputing All Possible Prefixes.** In the herding attack, the attacker may reasonably expect to produce a diamond structure with $2^{50}$ or more possible hash values. For a great many possible applications of the herding attack, this may be more than the possible number of prefix messages. The attacker may now take advantage of an interesting feature of the diamond structure: There is no restriction on the choice of starting hash values for the structure.

Let $2^k$, the width of the diamond structure, be the number of possible prefix messages that the attacker may need to herd to her fixed hash value. (If there are fewer prefix messages, the attacker appends one block to all the possible prefix messages, and varies that block to produce a set of prefix messages that is exactly the right size.) She computes the intermediate hash after processing each prefix message, and uses these intermediate hashes as the starting hash values for the diamond structure.

The initial work to construct the diamond structure in this way is the same as for the more general herding attack. However, the attacker now has the ability to immediately produce a message which starts with any possible prefix with the desired hash value. That is, she need not do a second expensive computation to herd the prefix she is given.

The attacker who has a larger set of possible prefixes than this is not lost; she may precompute the hashes of the most likely $2^k$ prefixes. Then, if any of those prefixes is presented to her, she can herd it immediately; otherwise, she must do the large computation, or simply allow her prediction or other deception to fail with some probability.

**Using Joux Multicollisions.** Joux multicollisions are not sufficient for the general herding attack. However, when the set of possible messages to be committed to is of the right form and can be precomputed, Joux multicollisions can be used to mount a weaker form of the herding attack.

Consider the case where the attacker wishes to commit to a sequence of "yes" or "no" predictions, without knowing which she will need to reveal later. An example of this would be a list of famous people who will or will not marry during the year. In the precomputation phase of the attack, the attacker determines a list of famous people and the order in which she will predict whether they will marry. Following the Joux multicollision technique, she produces a list of about $2^{n/2}$ variations on a "Yes, this person will marry this year" prediction and about $2^{n/2}$ variations on a "No, this person will not marry this year" prediction. Each prediction is independent; the attacker finds a colliding yes/no prediction for the first famous person, then for the second, and so on. See Figure 3. When finished, she publishes her list of famous people and the hash of her predictions for the future. At the end of the year, she "reveals" her predictions, choosing for each pair of colliding blocks the one that reflects what did happen that year.

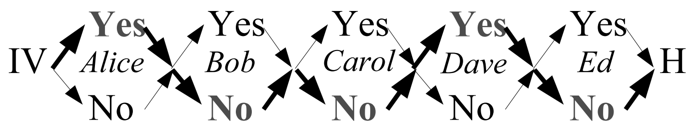


**Fig. 3.** Using Joux Multicollisions to Predict Who Will Get Married

This variant of the attack is much cheaper than those based on the diamond structure, but is also much less flexible. It can use existing cryptanalytic techniques on SHA1 and MD5 since, at each stage, the attacker is looking for two messages that collide starting from the same IV; of course, the use of existing cryptanalytic techniques might influence the structure of the attacker's yes/no predictions. Precomputations of enormous sets of prefixes become possible using this technique. Most importantly, it can be combined with the diamond structure and variations of the Joux multicollision to provide even more flexibility to the attacker, as we discuss below.

**Combining Precomputations and Joux Multicollisions.** In some cases, some large part of the information to be committed to will fit cleanly into the Joux multicollision structure, but other parts will not. For example, consider a prediction of the course and outcome of a national election in the United States[6]. Before the election is run, the attacker produces a set of 32 prefixes which describe the course of the election in broad terms, e.g., "Smith won a decisive victory," "Jones narrowly carried the critical swing states and won," etc. After this, each state's outcome is listed, e.g., "Alabama went for Smith, Alaska went for Jones, ...." The first part of the message is a precomputed diamond structure; the second part is a Joux multicollision allowing $2^{50}$ different outcomes.

**Applying the Joux Multicollision Idea to Diamond Structures.** An even more powerful way to structure these predictions is to concatenate precomputed diamond structures in a kind of super-Joux collision.

Consider the above description, but now suppose we wanted to specify one of 32 possible descriptions of how the election went in each state, e.g., "In Alabama, Smith won a resounding victory," or "In Maryland, Jones narrowly won after a series of vicious attack ads."

The attacker can string together 51 diamond structures total, one to describe the whole election, one for each state. This allows the attacker to "commit" to a prediction with $2^{255}$ possible values (requiring $2^{127.5+n/2+2}$ work with an n-bit hash function using a straightforward precomputed diamond structure), while doing much less work ($51 \times 2^{2.5+n/2+2}$). The attacker also gains enormous flexibility by being able to avoid the strict format of the Joux multicollisions.

[6] The only detail about US politics needed to understand this example is that all elections ultimately produce exactly one victor.

## 5 Applying the Attacks: Herding for Fun and Prophets

In this section, we describe how the herding attack can be used in many different contexts to do (what we believe to be) surprising things.

**Predicting the Future: The Nostradamus Attack.** The "Nostradamus attack" is the use of herding to commit to the hash of a message that the attacker doesn't even know. This destroys the ability to use hashes, for which collisions can be found, to prove prior knowledge of any information.

The Nostradamus attack is carried out in order to convince people that the attacker can tell the future. This could be based on some claimed psychic power, but also on some claimed improved understanding in science or economics, allowing detailed prediction of the weather, elections, markets, etc. This can also be used to "prove" access to some inside information, as with some attacker attempting to convince a reporter or intelligence agent that she has inside access to a terrorist cell or secretive government agency.

At a very general level, this attack works as follows:

1. The attacker presents the victim with a hash $H$, along with a claim about the kind of information this represents. She promises to produce the message that yields the hash after the events predicted have occurred.
2. The attacker waits for the events to unfold, just as the victim does.
3. The attacker herds a description of the events as they did unfold into her hash output, and provides the resulting message to the victim, thus "proving" her prior knowledge.

There are many variations on this theme; the predictions can be fully precomputed, completely unpredictable until they come to pass, or some mix of the two.

**Committing to an Ordering.** The techniques for many of the variants of the Nostradamus attack follow from the discussions in Sections 3 and 4. Here we suggest another possibility, which uses what we call a "hash router;" see Figure 4. Alice decides to prove (perhaps in a gambling context) that she can predict the outcome of a race with 32 entrants. She commits to a sequence of 32 hash outputs, $H_{0,1,\ldots,31}$. After the race is over, she produces 32 strings, $S_{0,1,\ldots,31}$ such that $S_i$ describes the entrant in the race who finished in $i$th place, and $H_i = \mathsf{hash}(S_i)$.

Alice builds a precomputed diamond structure starting from the names of the 32 entrants. When the diamond structure yields a final hash $H$, she produces 32 new message strings (probably simply strings like "1st place", "2nd place", etc.), and processes them from $H$ to get 32 different hash outputs. She commits to these hash outputs. When the time comes to reveal her choices, she produces 32 strings which commit her to the correct ordering of entrants in the race. Note that Alice can route any of her starting precomputed prefixes to any of the hash outputs.

**Retroactive Collisions.** Under normal circumstances, someone creating a hash collision must broadly know to what he is committing. While some clever attacks

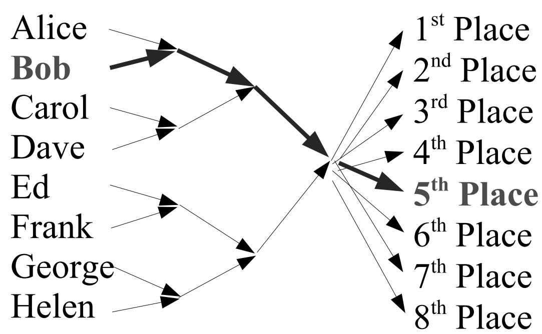


**Fig. 4.** Committing to an Ordering Using a "Hash Router"

have gotten around this by using some bits of the two colliding messages to change the meaning of later parts of a message[7, 9], these attacks are easy to detect by looking at the underlying data.

The herding attack may be used to "backdate" a collision. That is, the attacker sets up a collision today, and commits to its hash and perhaps one message with that hash. Later, she decides what document she wishes collided with the one she committed to, and so she herds that document to the same hash. The property of the hash function being violated is identical as in the case of "proving" prior knowledge, but the applications are quite different.

**Stealing Credit for Inventions.** The attacker can use the same idea to claim to be a brilliant inventor, while actually stealing other peoples' work. He submits hashes to a digital timestamping service periodically. After he sees some new invention he wants to claim, he herds a description of the invention to some old hash value.

To save the attacker from building multiple diamond structures, the attacker could construct a single diamond structure, and append a single message block which would vary for each submission.

**Tweaking a Signed Document.** Consider the case where Alice has a very reasonable document which she has signed, making some sensible predictions about the future or statements of fact or terms of agreement. She wants to make sure she can later "tweak" this document in some ways. Herding will permit this:

1. Using the precomputed variant with Joux multicollisions, she can produce two alternatives for each paragraph or section of the document.
2. Using the precomputed diamond with Joux multicollisions, she can produce many variations for some sections, and pairs of variations for others. She chooses one to produce initially, but can change to another without changing the hash.

3. Using the full herding attack, she can produce one "herded" document. Any variation in the "prefix" part of the document she wishes to make later can be made by carrying out another herding attack.

This attack can be used to tweak messages, contracts, news stories, signed/hashed software, etc.

**Random Number Fixing.** Alice and Bob want to agree on a shared random sequence for some game. Alice sends $\mathsf{hash}(X_1)$, then Bob responds with $X_2$. Finally, Alice reveals $X_1$, and Alice and Bob each derive random bits by combining $X_1$ and $X_2$ in some way. The herding attacks and its variations can be used to allow Alice to exert substantial control over the resulting random bit sequence. If the full herding attack isn't practical in this scenario, Alice can at least use the Joux multicollision variant to allow herself two choices per agreed random number, where Bob has no choice.

## 6 Finding Multiblock Fixed Points

Attacks on commitment schemes are not the only applications of the diamond structure and herding attack ideas. We can also find short cycles in hash functions. This is done in a simple way: We first construct a diamond structure, where each of the starting hash values in the structure are found by generating a random message block, and computing the compression function result of that message block from the hash function's initial value. If the diamond structure is $2^k$ wide, we then compute $2^{n-k}$ trial message blocks from the end of the diamond structure. We expect an intermediate collision, which yields a $k$-block fixed point for the hash algorithm.

This can be extended; with $2^{n-k+r}$ work, we expect about $2^r$ different $k$-block fixed points, all reachable from a legitimate message. These can be concatenated together; we can choose which of the $2^r$ $k$-block chunks of message we wish to append to the message next, without reference to previous choices. Further, any message can be "herded" to this set of fixed points with about $2^{n-k}$ work and $k$ appended blocks. For completeness, we recall that [17] show how to find single-block fixed points in Davies-Meyer constructions and [12] show how to find single-block fixed points in Snefru.

## 7 Conclusions

In this paper, we have defined a property of a hash function, Chosen Target Forced Prefix (CTFP) preimage resistance, which is both surprisingly important for real-world applications of hash functions, and also surprisingly dependent on collision resistance of the hash function. We have described a variation on the Joux multicollision technique for building tree-like structures of multicollisions called "diamond structures," and enumerated a number of techniques made possible by these structures. We have described a number of arguably practical attacks which use these techniques.

At a very basic level, we believe that the most important lesson the reader can take from this paper is that using iterated hash functions whose collision resistance has been violated is very difficult, even when the relevant security property does not appear to depend on collision resistance.

A great deal of research remains to be done in this area. The diamond structure seems likely to us to be about as useful in developing new attacks as the Joux multicollision result, and we hope to see others building on the work in this paper by finding other surprising things to do to iterated hash functions using herding attacks and the diamond structure. Additionally, there may be many other surprising ways in which iterated hash functions built on the Damgård-Merkle construction may be attacked when the attacker can find intermediate collisions.

## 8 Acknowledgments

The authors wish to thank Dan Brown, Morris Dworkin, Niels Ferguson, Hal Finney, Stuart Haber, Ulrich Kuehn, Bart Preneel, Christian Rechberger, Bruce Schneier, the many participants of the NIST hash workshop, and the anonymous referees for helpful comments and discussions on the subject of this paper. T. Kohno was supported by NSF CCR-0208842, NSF ANR-0129617, and NSF CCR-0093337. Part of this research was performed while T. Kohno was visiting the University of California at Berkeley.

## References

1. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS 93*, pages 62–73. ACM Press, 1993.
2. E. Biham and R. Chen. Near-collisions of SHA-0. In M. Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 290–305. Springer-Verlag, Berlin, Germany, 2004.
3. E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, and W. Jalby. Collisions of SHA-0 and Reduced SHA-1. In R. Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*. Springer-Verlag, Berlin, Germany, 2005.
4. D. R. Brown and D. B. Johnson. Hash functions based on block ciphers. In D. Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*. Springer-Verlag, Berlin, Germany, 2001.
5. J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*. Springer-Verlag, Berlin, Germany, 2005.
6. I. Damgård. A design principle for hash functions. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 416–427. Springer-Verlag, Berlin, Germany, 1989.
7. M. Daum and S. Lucks. Attacking hash functions by poisoned messages: The story of Alice and her boss, 2005. `http://www.cits.rub.de/MD5Collisions`.
8. R. D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, Jan. 1999.

9. M. Gebhardt, G. Illies, and W. Schindler. A note on practical value of single hash collisions for special file formats. NIST Cryptographic Hash Workshop, 2005. No published proceedings, available online at `http://www.csrc.nist.gov/pki/HashWorkshop/2005/Oct31_Presentations/Illies_NIST_05.pdf`.
10. A. Joux. Multicollisions in iterated hash functions. Application to cascaded constructions. In M. Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 306–316. Springer-Verlag, Berlin, Germany, 2004.
11. D. Kaminsky. MD5 to be considered harmful someday. Cryptology ePrint Archive, Report 2004/357, 2004. `http://eprint.iacr.org/`.
12. J. Kelsey and B. Schneier. Second preimages on $n$-bit hash functions for much less than $2^n$ work. In R. Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 474–490. Springer-Verlag, Berlin, Germany, 2005.
13. V. Klima. Finding MD5 collisions on a notebook PC using multi-message modifications. Cryptology ePrint Archive, Report 2005/102, 2005. `http://eprint.iacr.org/`.
14. X. Lai and J. L. Massey. Hash functions based on block ciphers. In R. A. Rueppel, editor, *EUROCRYPT'92*, volume 658 of *LNCS*. Springer-Verlag, Berlin, Germany, 1992.
15. A. Lenstra, X. Wang, and B. de Weger. Colliding X.509 certificates. Cryptology ePrint Archive, Report 2005/067, 2005. `http://eprint.iacr.org/`.
16. R. C. Merkle. One way hash functions and DES. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 428–446. Springer-Verlag, Berlin, Germany, 1989.
17. S. Miyaguchi, K. Ohta, and M. Iwata. Confirmation that some hash functions are not collision free. In I. Damgård, editor, *EUROCRYPT'90*, volume 473 of *LNCS*. Springer-Verlag, Berlin, Germany, May 1990.
18. B. Preneel, 2005. Personal communication.
19. V. Rijmen and E. Oswald. Update on SHA-1. In A. Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 58–71. Springer-Verlag, Berlin, Germany, 2005.
20. P. van Oorschot and M. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
21. X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the hash functions MD4 and RIPEMD. In R. Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*. Springer-Verlag, Berlin, Germany, 2005.
22. X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*. Springer-Verlag, Berlin, Germany, 2005.
23. X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 19–35. Springer-Verlag, Berlin, Germany, 2005.
24. X. Wang, H. Yu, and Y. L. Yin. Efficient collision search attacks on SHA-0. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*. Springer-Verlag, Berlin, Germany, 2005.
25. G. Yuval. How to swindle Rabin. *Cryptologia*, 3(3):187–189, 1979.