Hash Pile Ups: Using Collisions to Identify Unknown Hash Functions

R. Joshua Tobin
School of Mathematics
Trinity College Dublin,
Ireland.
Email: tobinrj@tcd.ie

David Malone
Hamilton Institute
National University of Ireland,
Ireland.
Email: David.Malone@nuim.ie

***Abstract*—Hash functions are often used to consistently assign objects to particular resources, for example to load balancing in networks. These functions can be randomly selected from a family, to prevent attackers generating many colliding objects, which usually results in poor performance. We describe a number of attacks allowing us to identify which hash function from a family is being used by observing a relatively small number of collisions. This knowledge can then be used to generate a large number of colliding inputs. In particular we detail attacks against small families of hashes, Pearson-like hash functions and linear hashes, such as the Toeplitz hash used in Microsoft's Receive Side Scaling.**

I. Introduction

Hash functions are often used to spread load across several resources. For example, in the common case of a hash table, a hash function is used to give a consistent assignment of objects to linked lists. In modern networking, similar techniques are used by high-end network cards to assign packets to CPUs [1], by switches/routers to assign packets to links (e.g. Cisco CEF load-sharing [2], 802.3 LAG [3] or OSPF ECMP [4]), or by routers/load balancers to assign flows/requests to servers (e.g. F5 BigIP hash based load balancing [5]).

In [6], the authors described *algorithmic complexity attacks*, where by choosing the inputs to an algorithm, an attacker could provoke poor performance of an algorithm, rather than typical performance. Attacks on known hash functions are a canonical example of this. Attacks can be frustrated by the use of a randomly selected (keyed) hash function. Typically these hashes will not be cryptographic hash functions, as the computational cost associated is too high (e.g. Figure 1). Consequently, generating collisions once the hash function is known is not usually computationally expensive.

In this paper we look at attacking such keyed hash functions in situations where we can determine if two inputs collide, even though we do not know the actual value of the hash function. This may be possible in a number of contexts.

1) In the case of a hash table, the length of the hash chains might be estimated by timing lookups. By sequentially adding objects and measuring the chain lengths each time, it may be possible to determine which objects have been added to the same chain.
2) In the case where the hash is used to assign objects to a processing resource, such as a CPU or server, then we can load one of the resources to produce a change in response time. After that, we may see which objects collide by checking their response time.
3) In the case of a device assigning packets to a CPU, we might send packets back-to-back and look for packet reordering. Packets going to the same CPU should never be reordered, whereas packets going to different CPUs may occasionally be reordered. The chance of reordering might even be increased by choosing the protocol or packet size to vary the processing time for the packets.
4) In the case where packets are being load balanced across links, such as in LAG (link aggregation group) or ECMP (equal cost multi-path), it may be possible to determine which flows share a route using a probing tool such as traceroute (e.g. [7]).
5) In the case of load balancing of more complex services, such as HTTP or DNS, it may be possible to fingerprint the assigned server by some higher level mechanism.

The last two attacks, which depend on a clear fingerprint,

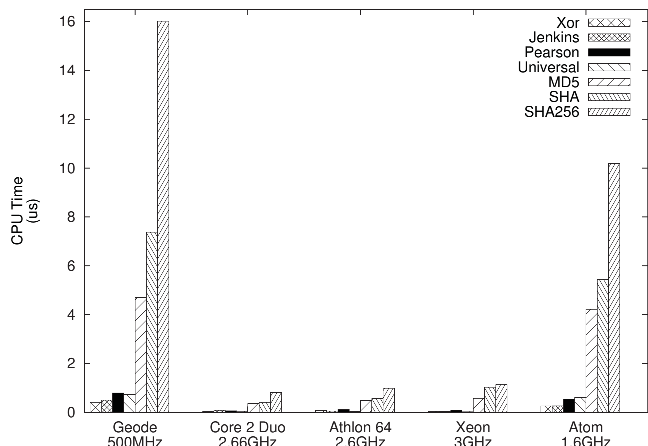

Fig. 1. The average cost of hashing an IPv6 flow record with different algorithms on a selection of CPUs. Xor is a simple per-byte xor; Jenkins is the hash used in Section II; Pearson is the hash used in Section III; Universal is a Carter-Wegman hash used from [6]; MD5, SHA and SHA256 are standard OpenSSL implementations. Results are averaged over a range of hash keys and records.

are easy to implement while some of the timing attacks might be more challenging. However, a number of these attacks have been demonstrated to be feasible. For example, [8] shows a timing attack against packets sent through a stateful firewall. The limitations of similar attacks in the face of network jitter have been studied in [9].

Given that we can test if inputs (say x and y) of a hash collide (i.e. if $h(x) = h(y)$), the question we will study is: can we quickly determine the hash h if we know it is drawn from some family H. In the following sections we will show that it is often possible to find h surprisingly quickly. In particular, we demonstrate how this can be done when the family satisfies certain general conditions. These conditions include having a small range of hashes, and having a linear function underlying the hash (even in the presence of an additional non-linear layer).

We consider two ways to measure the amount of work required to determine h and cause collisions. First, we consider the average number of objects we have to insert to cause a particular number of collisions. We call this the number of probe strings (i.e. objects inserted) required to cause L objects to collide. We also consider the average number of comparisons, as depending on the exact nature of the attack, this may be more representative of the work required. We note that with no information about a hash function with uniformly distributed output, we require on average KL probe strings to cause L objects to collide, where K is the number of output buckets. This blind attack requires no comparisons.

II. Small Hash Set

We first consider what happens if the set from which we select hash functions is relatively small. Suppose we have H different hash functions which each map to K output buckets. We are aiming to identify a particular h in use. We may pick inputs to the hash $s_1, s_2, \ldots$ until we find that $h(s_i) = h(s_j)$. Using Birthday Paradox arguments, this will require about $\sqrt{K}$ probe strings and about K comparisons. The attack is summarised in Algorithm 1.

If H is small enough to enumerate, we then calculate the subset of all H functions which collide on this set of strings. We note that h must agree on these colliding strings, and each of the other hash functions in H will match with probability $1/K$. Thus, we eliminate all but the correct hash with probability

$$\left(1-\frac{1}{K}\right)^{H-1} = \left(\left(1-\frac{1}{K}\right)^{K}\right)^{\frac{H-1}{K}} \approx e^{-\frac{H}{K}}.$$

If we are left with more than one possible hash, we may repeat this attack until just one hash remains. Even if we do not reuse information between rounds of this attack, we would expect to be successful in $e^{\frac{H}{K}}$ attempts. A more optimistic attacker that reused information between attempts might hope to have around HK^{-n} hashes remaining after n attempts, and so require around $\log_K H$ attempts to identify h. This suggests between $\sqrt{K}\log_K H$ and $\sqrt{K}e^{\frac{H}{K}}$ probe strings or between

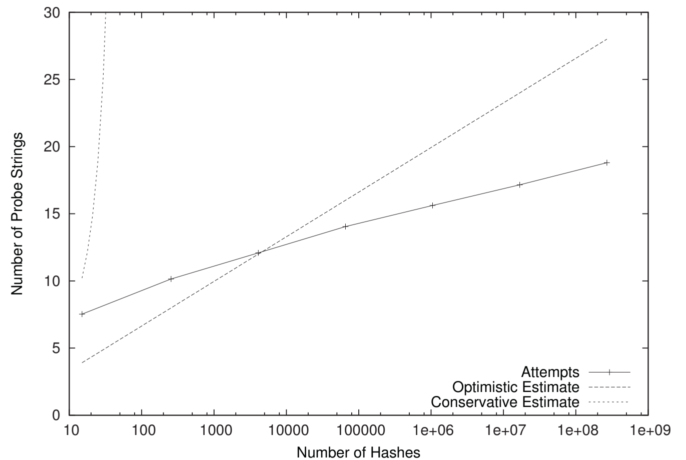

Fig. 2. The average number of probe strings required to determine the hash, as the number of possible hashes h increases. the number of output buckets is $k = 16$ and values shown are averaged over 1000 trials.

$K\log_K H$ and $Ke^{\frac{H}{K}}$ comparisons are required for the attack to identify the hash function.

To demonstrate this, we apply the attack to the keyed version of Bob Jenkins's hash from Appendix A.2 of [10]. We evaluate the hash mod 16, so we have 16 output buckets and vary the range of the *initval* parameter of the hash. Figure 2 shows the average number of probe stings we need to use to recover *initval* over 1000 different runs, where we reuse information between collisions. In practice we find the conservative estimate of the attack's performance is too conservative. Performance is closer to the crude optimistic estimate, and can even require fewer probes than expected when the number of hashes is larger. This is because when more probe strings are required, multiple collisions with previous strings become more likely, and each collision provides us with more opportunities to reduce the number of candidate hashes.

For this attack to be ineffective, we either require K to be very large or require H to be considerably bigger than K. Hashes are sometimes keyed by initialising some internal state, such as a 32-bit integer. In this case we have $H = 2^{32}$, and this analysis suggests that we should have K much larger (say $K > 2^{64}$) than H. For many applications, such a large value of K will be impractical. When $H = 2^{32}$, if collisions can be found, then an attack that progressively eliminates candidate hashes is likely to uniquely identify h quickly.

Note, this is similar to the black box attack described in [8], however they make one comparison per hash function, rather than finding colliding inputs and then eliminating. As we have demonstrated, this attack could be used against hashes such as Bob Jenkins's hash function [11], which was originally unkeyed, but variants with small keys have been used in the Linux Kernel and also suggested in [10].

III. Pearson-like Hash Functions

This section outlines an attack on a hash built from a known group operation and a unknown permutation (effectively the

Algorithm 1 A simple attack when H is small enough to enumerate.

```
candidates ← {h_1, ... h_H}
while |candidates| > 1 do
    Insert random s_i until collision between some values s_i and previous s_j.
    candidates ← {h ∈ candidates : h(s_i) = h(s_j)}
end while
Use h ∈ candidates to determine collisions.
```

key). Hashes such as Pearson's hash [12] are in this class. The attack in this case will recover the permutation, and so the key. Pearson's hash has been suggested for use in load balancing, for example in [13], however this is with a well-known key. Note this is an extended version of an attack on this hash which discussed with Pearson [14].

A. The Model

We present the attack on a generalisation of the Pearson hash. Let G be any finite group, with its operation denoted by $\otimes$. We have a hash function h, which maps from strings of elements onto G, defined recursively as follows:

$$h(s_1 s_2 \cdots s_n) = T(h(s_1 s_2 \cdots s_{n-1}) \otimes s_n)$$

where $s_i \in G$, $T \in S_G$ is a permutation of G, and $h(\epsilon) = 0$ (ϵ denoting the empty string, and 0 the identity of G). Here, the hash is determined by T, which is an arbitrary permutation of G, so there are $H = |G|!$ possible hashes.

Pearson's original hash is obtained by setting G to be the set $\{0, 1, \cdots, 255\}$ with the group operation $\otimes$ being xor (i.e. $\mathbb{Z}_2^8$). However, the attack we will describe will work for any finite group G, and so we leave G unspecified for this discussion.

This hash accepts arbitrary strings, but our attack only requires the use of fixed length strings. Our aim is to recover the permutation T, which is unknown to us, by using information on which strings collide.

B. Attack Description

The attack, which is summarised in Algorithm 2, has several steps which we will now discuss.

First, we probe with $h(x, 0, \cdots, 0)$ and $h(0, x, \cdots, 0)$ for all $x \in G$ and identify collisions. That gives us all values of a, b for which $h(a, 0, \cdots, 0) = h(0, b, \cdots, 0)$. By applying the definition of h:

$$\begin{aligned} & h(a, 0, \cdots, 0) = h(0, b, \cdots, 0) \\ \Leftrightarrow \quad & T^{n-2}(T(a) \otimes 0) = T^{n-2}(T(0) \otimes b) \\ \Leftrightarrow \quad & T(a) = T(0) \otimes b \end{aligned}$$

For all a there exists some b which satisfies the above equation, so if we know $T(0)$ we have determined the value of $T(a)$ for all a. That is, once we know $T(0)$, we know the entire permutation. There are $|G|$ different values for $T(0)$, and so there are $|G|$ possible permutations. This first step has used $2|G|$ probe strings and, on average, about $|G|^2/4$ comparisons.

Next, we insert $h(0, 0, \cdots, x)$ for each $x \in G$. Then we generate random strings $s_1 s_2 \cdots s_n$ and insert them. Each random string will collide with the string $0, 0, \cdots, x$ for exactly one value of x. After inserting our random string we know which x matches $s_1 s_2 \cdots s_n$. We then test which of each of the $|G|$ remaining permutation tables permits this collision. If there is more than one possible table left, we repeat this with another random string.

How many random strings will we have to insert to identify the permutation? Each time we test a possible $T(0)$ value against a collision between a random $h(s_1 s_2 \cdots s_n)$ and $h(00 \cdots 0x)$, we expect it to match by chance with probability $1/|G|$. So, after checking the $|G|$ possible $T(0)$, there will be exactly one left with probability $(1 - 1/|G|)^{|G|-1} = .3686 \approx 1/e$, if $|G|$ is large.

After t choices of the random string, we've effectively put each possible $T(0)$ through t tests. All $|G| - 1$ will have been eliminated on trial some trial between $1, \ldots t$ with probability $(1 - (1/|G|)^t)^{|G|-1}$. Thus, the chance of being eliminated on exactly trial t is $(1 - (1/|G|)^t)^{|G|-1} - (1 - (1/|G|)^{t-1})^{|G|-1}$. As this probability decreases rapidly in t, we note that the average number of trials, $\bar{t}$, required will be small. The number of comparisons for this phase will be $\bar{t}|G|/2$.

For Pearson's original hash, Figure 3 shows the distribution of t, i.e., the number of random strings required to determine T for a random sample of 10^6 different permutations T. For comparison, we also show the predicted distribution. The average number of probe strings required to determine the permutation T per trial was 769.636, which is $3|G|$ plus the average number of probes from Figure 3. The number of comparisons is more widely distributed, between about 13×10^3 and 21×10^3, with the mean number of comparisons 16.7×10^3, close to the expected value of 16.6×10^3.

With this attack, we can create L colliding inputs using just $L + O(3|G|)$ probe strings. The number of comparisons will be $|G|^2/4 + \bar{t}|G|/2$.

In this attack we have used several fixed strings. Firstly we hashed strings of the form $a000\ldots0$ and $0b00\ldots0$. In fact, we could have appended an arbitrary fixed suffix to $a0$ and $0b$, rather than the all zeros suffix, without changing the attack. Prepending an arbitrary prefix p makes the attack slightly more complex, where we end up with relations:

$$T(X \otimes a) = T(X) \otimes b$$

where $X = h(p)$ is unknown. In this case we may parameterise T by X and $T(X)$, and then use later stage collisions to eliminate possibilities. Then, without sending more probes,

Algorithm 2 An attack against Pearson-like Hashes.

Insert $(x, 0, \dots, 0)$ and $(0, y, \dots, 0)$ for $x, y \in G$ and note collisions
$C \leftarrow \{(x, y) : h(x, 0, \dots, 0) = h(0, y, \dots, 0)\}$
candidates $\leftarrow \{T \in S_G : T(a) = T(b) \otimes T_0, \forall (a, b) \in C, T_0 \in G\}$
Insert $(0, 0, \dots, x)$ for $x \in G$.
while $|\text{candidates}| > 1$ **do**
  Insert random s and find x so that $h(s) = h(0, 0, \dots, x)$.
  candidates $\leftarrow \{T \in \text{candidates} : h_T(s) = h(0, 0, \dots, x)\}$
end while
Use $h \in$ candidates to determine collisions.

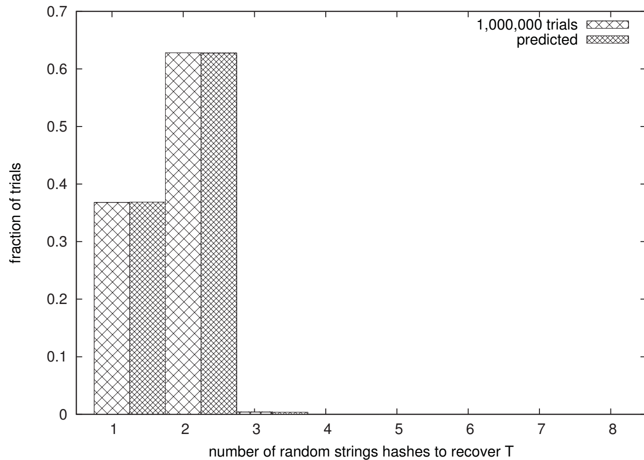

Fig. 3. The fraction of random hashes required to uniquely determine the permutation table. 10^6 trials.

we may test each possible pair $(X, T(X))$, to determine which actually lead to a collision between $a000\dots0$ and $0b00\dots0$. This reduces the number of possible remaining hashes to $|G|$, and then the attack can continue as if there is no prefix. The number of comparisons and probes is then the same as for the attack with no prefix, though some increased local computation is required.

IV. Linear Hashes

In this section we will consider a hash function that is an unknown linear function. An example of this hash is the Toeplitz function, recommended as part of Microsoft's Receive Side Scaling for network cards [1]. Here, a random Toeplitz matrix is used as the key for the hash function. The input strings are of size 64, 96, 256 or 288 bits, depending on the use of IPv4/IPv6 and the availability of layer-4 port numbers. The output is between 2 and 128 buckets, that is 1 to 7 bits. Note that before the output is used, it is looked up in an array to map the hash output to a CPU. We will consider the hash without this indirection table initially, and then consider how to attack the hash with the indirection table.

Some common unkeyed hash functions are linear (e.g. the IPSX hash in [10]), and this technique could also be used to identify collisions for them.

A. The Model

In this section we consider a hash function h which is an unknown linear function $h : \mathbb{Z}_2^n \to \mathbb{Z}_2^m$ over $\mathbb{Z}_2$. The attack we will describe can be generalised to homomorphisms between other groups.

B. The Attack

The steps of this attack are summarised in Algorithm 3. Let $E = \{e_1, \dots, e_k\}$ be a linearly independent subset of $\mathbb{Z}_2^n$, representing bits we can vary. Now, by probing, say, $h(x + e_1), \dots, h(x + e_k)$ we get a partition of E into $E_1, \dots, E_l$ so that all the members of E_i collide. This requires k probe strings and on average $kl/2$ comparisons.

Now we may choose even-sized subsets of each of E_i, say $E'_1, \dots E'_l$ and consider

$$h\left(x + \sum_{e \in \bigcup E'_i} e\right) = h(x) + \sum_{e \in \bigcup E'_i} h(e) = h(x),$$

because pairs from the same E'_i will cancel. We may choose the E'_i in $2^{|E_i|-1}$ ways, and each choice will give a different input $\sum_{e \in \bigcup E'_i} e$ because the set E is linearly independent. Thus we have generated

$$2^{|E_1|-1} \times \dots \times 2^{|E_l|-1} = 2^{|E_1|+\dots+|E_l|-l} = 2^{k-l}$$

collisions by using $k + 2^{k-l}$ probes and $kl/2$ comparisons.

Note, the largest that l may be is 2^m, the number of output buckets. This gives a lower bound of 2^{k-2^m} collisions. Suppose we control 64 bits of the input string and we are assigning inputs among $16 = 2^4$ resources (maybe CPUs or routes), then this leads to $2^{64-16} = 2^{48}$ collisions. However, if m is large, then this lower bound can be very pessimistic. We can get an alternative estimate by observing that if we throw k balls into 2^m urns, then the average number of urns with balls will be $a = 2^m \left(1 - \left(1 - \frac{1}{2^m}\right)^k\right)$. Heuristically, we expect around 2^{k-a} collisions, though this will not be exact because we are substituting the average of l rather than using its distribution.

For example, consider a linear hash with 6 output bits, or 64 output buckets[1]. We consider how the attack performs as

[1]We choose 6 output bits because it gives a reasonable spread of outputs, but also allows us to conduct a comparison with the non-linear indirection table in the next section. Our results generalise to other output sizes.

Algorithm 3 An attack against Linear Hashes.

$E = \{e_1, \ldots, e_k\}$.
Partition E into $E_1, \ldots E_l$ by inserting and noting collisions.
for all $E_1` \subset E_1, \ldots, E_l` \subset E_l$ and $|E_i`|$ even **do**
  Insert $\sum_{e \in \bigcup E_i'} e$.
end for

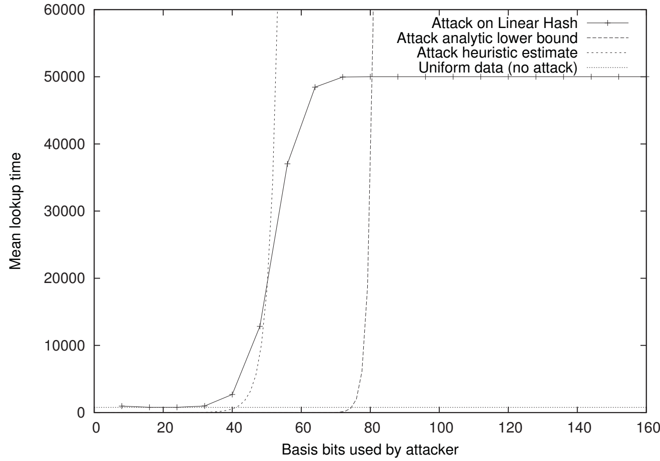

Fig. 4. Attacking a hash with 64 outputs with a variable number of input bits. Mean lookup time for 100,000 inserted strings.

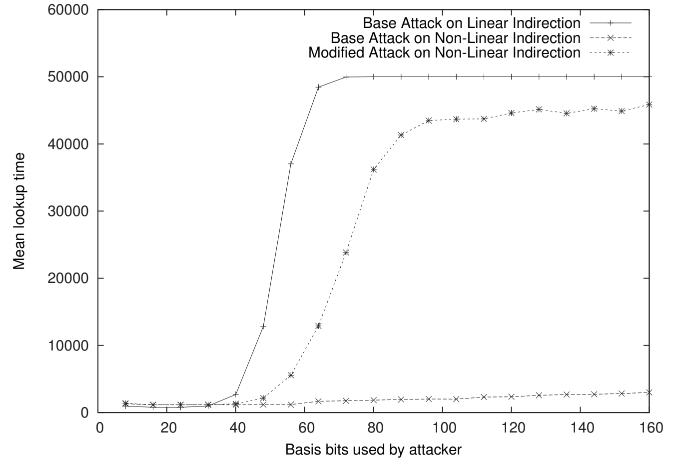

Fig. 5. Attacking a hash with 64 outputs with a variable number of input bits in the presence of an indirection table, where the attacker knows that the indirection table has changed once during the attack. Mean lookup time for 100,000 inserted strings.

the number of basis vectors k used by the attacker varies. Figure 4 shows the mean object lookup time when we insert 10^5 strings. If the above method produces fewer than 10^5 strings, we continue by using random strings, until 10^5 strings have been inserted. The results shown are averages over 10^3 different Toeplitz matrices.

As expected, the attack performs well against the hash once a moderate number of bits are available to the attacker. For comparison, we show the mean lookup time when strings are uniformly distributed over the buckets, which demonstrates how effective the attack can be. We also show our lower bound and heuristic estimate for the mean lookup time. We see that the lower bound is quite conservative, but the heuristic gives a reasonable indication of when the attack becomes effective.

C. Modified Attack

We now consider the case where the linear function is composed with an 'indirection table' — a further mapping which reduces the range of the hash to something suitable for a particular application (for example, assignment of packets to processors [1]).

If the indirection table is linear, then the composition of the hash function and the indirection table will also be linear. In this case, we can still use the attack described above. However, if the indirection table is not linear, our previous attack will be less effective — now the composition of the hash function and the indirection table is not linear, and our attack requires linearity to combine inputs.

For example, consider the situation tested in the previous section. Now the linear part of our hash outputs 7 bits, but using an indirection table we map this to 6 bits (again 64 buckets). As was said above, if the indirection table is linear, we are in the situation described shown in Figure 4. So instead assume the indirection table is not linear, and is instead a randomly generated mapping. Figure 5 shows how the mean lookup time changes in this case. We see that this significantly weakens the attack, though more collisions are generated than we would expect if the strings were uniformly distributed.

In some situations, we can improve the attack, even with a nonlinear indirection table. We are interested in the inputs that collide in the linear component of the hash function, and not those that collide in the non-linear indirection table. We can distinguish between these two cases by exploiting the fact that in practice the indirection table may change regularly; indeed, this is suggested as a method of load balancing for RSS. We first insert a set of strings, as before. We identify the collisions, and then we wait for the indirection table to change. We then insert these strings again, and record the collisions. Those strings that collide in both tests are likely to be collisions in the linear hash. We then continue with the attack as originally described. Figure 5 shows the effectiveness of this modified attack. We see that the attack regains much of its previous effectiveness. By observing collisions for more than two indirection tables, the attack could be further improved.

While we have described these attacks for $\mathbb{Z}_2^m$, a group

where all elements have order 2, similar attacks could be conducted on groups with higher order elements. If h was a homomorphism between groups H and G and and d is a number such that g^d is always the identity, then we can perform the same attack by choosing subsets of E_i which have a number of elements divisible by d. Instead of linear independence of E we would want a set with a small number of group relations. However, the effectiveness of this attack is reduced as d increases, because the number of subsets with a multiple of d elements decreases rapidly. Groups with a larger d are subject to other attacks, such as noting that for colliding inputs x, y and that $h(N(x - y)) = 0$ for integers N (e.g. *advanced hash flooding* [15]).

V. Discussion and Conclusion

In this paper we have shown how collisions between strings or objects can be used to identify which hash function h from a family of hash function H is in use. If the hash functions themselves are not collision resistant, this leaves the system open to algorithmic complexity attacks. We have discussed how finding collisions may be possible in practice.

For three cases, we demonstrate attacks. In the first case, the set of hash functions is small, and we can simply enumerate via collisions found using the birthday paradox. In the second case, the key indexing the hash function is a permutation, and so the large number of permutations (e.g. 256! of one byte) means the hashes are not subject to enumeration. However, for Pearson-like hashes we show how the permutation can be discovered using a relatively small number of tests, and then collisions can easily be generated.

We then consider hash functions that are linear, particularly with respect to xor. We show how to identify collisions for these hash functions, even without explicitly identifying the function. This attack can also be extended if a time-varying nonlinear part is present. While this attack can be extended to other groups, it is less effective for groups with a large index. For example, the Carter-Wegman families of universal hash discussed in [6] are linear, but modulo a large prime, and so the elements have a large index. However, attacks against these functions when used as part of a MAC [16], [17] are likely to generalise to the situation considered here.

The abstract attack that we presented at the start of this paper, determining a hash function from a family using collisions, can be applied to almost any family of functions. While we have identified some general conditions that should be avoided (e.g. the size of the family being small with respect to the number of outputs or the functions being homomorphisms over groups with small index elements), it would be interesting to establish if there are conditions that can provide some guarantees that identifying the hash or colliding inputs is hard. Another option is to design cryptographic-strength hash functions that are computationally less expensive, e.g. SipHash [15].

An alternative would be to look at methods for changing the hash function, either periodically or when an attack is suspected. As we show in Section IV-C, if this is not done carefully, it can open the system to more effective attacks. The nature of the techniques applicable in this situation will vary from application to application. For example, for routing packets it may be acceptable to change the hash function from time to time, as the exact route that packets take through the network is usually not important as long as packet reordering is uncommon. However, if we are assigning HTTP flows to load-balanced web servers, then reassignment of existing flows will usually break the connection, which will usually be considered unacceptable. However, there still may be general lessons to be learned regarding how detection, timing and rehashing can be performed in this context.

Acknowledgments

This work was supported by Science Foundation Ireland grant 08/SRC/I1403 and 07/SK/I1216a.

References

[1] Microsoft, "Scalable networking: Eliminating the receive processing bottleneckintroducing RSS," http://download.microsoft.com/download/5/D/6/5D6EAF2B-7DDF-476B-93DC-7CF0072878E6/NDIS_RSS.doc, April 2004.
[2] Cisco, "Cisco's express forwarding (CEF)," http://www.cisco.com/en/US/tech/tk827/tk831/tk102/tsd_technology_support_sub-protocol_home.html.
[3] IEEE Standards, "IEEE Std 802.3, 2000 edition: IEEE standard for information technologytelecommunications and information exchange between systemslocal and metropolitan area networksspecific requirements," 2000.
[4] C. E. Hopps, "Analysis of an equal-cost multi-path algorithm," 2000.
[5] F5, "Hash load balancing and persistence on BIG-IP LTM," http://devcentral.f5.com/Tutorials/TechTips/tabid/63/articleType/ArticleView/articleId/135/Hash-Load-Balancing-and-Persistence-on-BIG-IP-LTM.aspx.
[6] S. Crosby and D. Wallach, "Denial of service via algorithmic complexity attacks," in *Proceedings of the 12th USENIX Security Symposium*, August 2003.
[7] B. Augustin, T. Friedman, and R. Teixeira, "Measuring load-balanced paths in the internet," in *Proceedings of the Internet Measurement Conference*, October 2007.
[8] N. Bar-Yosef and A. Wool, "Remote algorithmic complexity attacks against randomized hash tables," *E-business and Telecommunications, Communications in Computer and Information Science*, vol. 23, pp. 162–174, 2009.
[9] S. Crosby, D. Wallach, and R. Riedi, "Opportunities and limits of remote timing attacks," *ACM Transactions on Information and System Security*, vol. 12, no. 3, pp. 17:1–17:29, Janurary 2009.
[10] T. Zseby, M. Molina, N. Duffield, S. Niccolini, and F. Raspall, "Sampling and filtering techniques for IP packet selection," RFC 5475, 2009.
[11] R. Jenkins, "Algorithm alley," *Dr. Dobb's Journal*, September 1997.
[12] P. K. Pearson, "Fast hashing of variable-length text strings," *Communications of the ACM*, vol. 33, no. 6, pp. 677–680, 1990.
[13] B. Volz, S. Gonczi, T. Lemon, and R. Stevens, "DHC load balancing algorithm," RFC 3074, 2001.
[14] P. Pearson, "Discussion on fast hashing," Personal Communication, 2008.
[15] J.-P. Aumasson and D. J. Bernstein, "SipHash: a fast short-input PRF," Cryptology ePrint Archive, Report 2012/351, 2012, http://eprint.iacr.org/.
[16] H. Handschuh and B. Preneel, "Key-recovery attacks on universal hash function based MAC algorithms," *Advances in Cryptology–CRYPTO 2008*, pp. 144–161, 2008.
[17] J. Black and M. Cochran, "MAC reforgeability," in *Fast Software Encryption*. Springer, 2009, pp. 345–362.