JOURNAL OF COMPUTER AND SYSTEM SCIENCES 22, 265–279 (1981)

New Hash Functions and Their Use in Authentication and Set Equality

MARK N. WEGMAN AND J. LAWRENCE CARTER

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

Received November 5, 1980

In this paper we exhibit several new classes of hash functions with certain desirable properties, and introduce two novel applications for hashing which make use of these functions. One class contains a small number of functions, yet is almost universal$_2$. If the functions hash n-bit long names into m-bit indices, then specifying a member of the class requires only $O((m + \log_2 \log_2(n)) \cdot \log_2(n))$ bits as compared to $O(n)$ bits for earlier techniques. For long names, this is about a factor of m larger than the lower bound of $m + \log_2 n - \log_2 m$ bits. An application of this class is a provably secure authentication technique for sending messages over insecure lines. A second class of functions satisfies a much stronger property than universal$_2$. We present the application of testing sets for equality.

The authentication technique allows the receiver to be certain that a message is genuine. An "enemy"—even one with infinite computer resources—cannot forge or modify a message without detection. The set equality technique allows operations including "add member to set," "delete member from set" and "test two sets for equality" to be performed in expected constant time and with less than a specified probability of error.

INTRODUCTION

Hash functions are functions which map from larger domains to smaller ranges. They may be viewed as a means of assigning an abbreviation to a name. A desirable property of a hash function is that "most of the time," when the hashed values of two quantities are the same, the quantities are the same. While one must be careful of what one means by "most of the time," this intuition has led to the two applications described in this paper. For instance, the set equality tester works by maintaining an abbreviation for each set (namely, its hashed value). It will declare two sets to be equal if the abbreviations are the same. Thus, it is correct most of the time when it says the sets are equal, and always right when it says they are unequal. While this idea is not new [13], we here use the techniques of universal hashing so that the probability of making an error is small for *any* pair of sets, not just for randomly chosen sets. We first will briefly summarize universal hashing to make this distinction clearer.

Hashing can be used to achieve fast average performance for a variety of applications, most notably for an associative memory such as a compiler's symbol

0022-0000/81/030265–15$02.00/0

table or a database. One would like "average performance" to mean the performance averaged over all inputs that the application will be run on. However, one usually doesn't know the actual data, and so one chooses a hash function which will work well assuming each input is equally likely. Recently an approach to hashing has been developed which allows one to achieve and prove a fast average performance, without needing to assume anything about the probability distribution on the inputs [6]. In this approach, one has a collection of hash functions, instead of just one. Each time the application is run, a hash function is randomly selected from the set. (If this is impractical, functions may be chosen less frequently, or perhaps only once. See [6] for a more complete discussion.) If the set of functions is carefully chosen to be what is called a universal$_2$ class, then many applications of hashing will have provably good expected performance for any distribution of inputs, not just the uniform distribution.

Known universal classes contain a fairly large number of hash functions. For instance, the functions in a typical class can hash n-bit long names, and the class contains $2^{O(n)}$ functions. Thus, $O(n)$ bits are required to specify a randomly chosen function. In Section 3, we present a set of functions which is "almost" strongly universal$_2$ and is much smaller—only $\log(n)$ bits are required. This improvement can make some applications of hashing practical, for instance the authentication technique described below. Also, if used in conjunction with the extendible hashing scheme of [9], one can make a fast, practical and completely general associative memory subroutine package.

A possibly important use of these functions, described in Section 2, is a provably secure authentication system. This system allows the receiver of a message to be sure of the authenticity of a message—that the message was not forged or modified by an unauthorized "enemy." It is necessary for the sender and the receiver to share a secret key whose length is on the order of the log of the length of the message. But unlike digital signatures based on public key cryptosystems, it can be proven that this system is secure even against an enemy with infinite computing power. Also, there are no messages which just happen to be easy to forge.

Section 4 gives a refinement of the authentication system which allows many messages to be sent using the same secret key, with each message requiring an additional but shorter key. The total length of the keys required for sending multiple messages asymptotically achieves the lower bound.

The application which motivated defining strongly universal$_\omega$ classes of functions—a set equality tester—is given in Section 5. Assuming that sets are constructed using certain specified operations, we give a technique which has an input independent and small chance of error for determining that sets are equal. The expected running time of the algorithm is input independent and is linear in the number of construction operations and equality tests. This is the only algorithm we have seen with the dubious distinction of requiring probabilistic analyses of two kinds—it can both make a mistake and take a long time doing so. As Gill has observed [11], one can always make the running time constant at the expense of a larger probability of error. Alternatively, one can use the technique in conjunction

with an ordinary set equality testing method. If the probabilistic test says the sets are different, it must be correct. If it says the sets are the same, one could then perform the conventional test.

Finally, in Section 6, we show how to construct a strongly universal$_\omega$ set of functions. The functions in this set can be evaluated rapidly.

1. Strongly Universal Sets of Hash Functions

To be universal$_2$, a set of functions from A to B must only satisfy a requirement on the probability that a randomly chosen function will map two points of A to the same value. For a set of functions to be strongly universal$_n$, a randomly chosen function must, with equal probability, map any n distinct points of A to any n values in B; in other words, any n points must be distributed randomly throughout B by the functions. More formally,

Definition. Suppose H is a set of hash functions, each element of H being a function from A to B. H is *strongly universal$_n$* if given any n distinct elements $a_1,...,a_n$ of A and any n (not necessarily distinct) elements $b_1,...,b_n$ of B, then $|H|/(|B|^n)$ functions take a_1 to b_1, a_2 to b_2, etc. ($|X|$ means the number of elements in the set X.) A set of hash functions is strongly universal$_\omega$ if it is strongly universal$_n$ for all values of n.[1]

Carter and Wegman [6] present several classes of hash functions which turn out to be strongly universal$_2$. Strongly universal$_n$ sets of functions can be created using polynomials over finite fields. In particular, let A and B both be the same finite field. Let H be the class of polynomials of degree less than n. H is strongly universal$_n$ since given any n distinct elements of A and corresponding elements of B, there is exactly one polynomial of degree less than n which "interpolates" through the designated pairs. (The standard linear algebra proof which uses the invertibility of the Vandermonde matrix also works with finite fields.)

It may seem peculiar to define a set of hash functions with A and B being the same size. However, it is easy to make B smaller by, for instance, just choosing the last bits of the hashed value. If the size of the field is a power of two, the result will still be a strongly universal$_n$ class of functions; otherwise, it will still be "close."

In Section 6, we will exhibit a strongly universal$_\omega$ set of functions.

2. Digital Signatures and Authentication Tags

It is often desirable to be able to send a message over an insecure line and yet allow the receiver to be certain of the identity of the sender. (For convenience, we will use "A" or "he" to denote the purported sender of a message, and "B" or "she" for the receiver.) Before modern communication methods existed, A could add a handwritten signature to a message. B could compare the signature on a message to

[1] The only strongly universal$_\omega$ set of functions from A to B is the set of all functions from A to B; however, we feel it gives useful intuition to think of this class as defined here.

what she knew was A's signature to assure herself the message was genuine. Someone who intercepted the message could not cut off the signature and paste it to a different document without detection. We seek to gain these advantages in the case of digital messages.

A signature serves a number of functions:

(1) It can assure B that the message was sent by A or someone authorized by him.

(2) It can be used to prove (perhaps in a court) that A or someone authorized by him sent the message, and

(3) It can assure B that A himself, not just someone authorized by A, actually sent the message.

Authentication tags provide a method of accomplishing the first of these three functions. If there is an agency which everyone trusts, then authentication tags can also be used to provide the second function. Before presenting these authentication tags, we would like to review two other methods which also have some of the properties of signatures—digital signatures and encryption. None of the methods we will discuss can be used for the third function since the methods do not depend on physical properties of A but only on some knowledge which he could reveal to someone else.

A digital signature as discussed by Diffie and Hellman [7] consists of a string of bits which is concatenated to a message. Only A knows the function which is used to generate the signature, but he also publishes a checking function. This checking function allows anyone to test whether the signature is valid for the particular message. Moreover, without the signing function (but even with the checking function) it is difficult to determine the correct signature to any alternate message. For all messages there exists a valid signature.

Diffie and Hellman [7], and Rivest *et al.* [12] have presented "public key cryptosystems" which allow the above (in addition to allowing an interesting type of encryption.) However,

THEOREM. *No public key cryptosystem is unbreakable. That is, an enemy with unbounded computing resources can forge messages.*

Proof. An enemy with enough time can guess all possible signatures for a particular message and when a valid signature is found use it.

Thus, all such signature schemes can be cracked in nondeterministic polynomial time, and none have so far been shown to be NP-complete. In fact, [1, 2] have shown that unless $NP = \text{co-}NP$ (which many people believe is unlikely) no public key cryptosystem can be NP-complete.

A traditional encryption scheme also provides some of the function of a signature. Suppose A and B have agreed upon an encryption and a decryption method, which they keep secret. When B receives any string of bits, she applies the decryption

algorithm. If the result looks sensible, she has some assurance that the string of bits was indeed a message sent by A. This approach has several disadvantages. Firstly, it could happen that if a certain portion of any encrypted valid message is twiddled in a certain way, then the result still has a reasonable chance of being an encrypted valid (but different) message. If so, an enemy could interfere with the communication, even without being able to decrypt any message. This is a technique for creating malicious mischief.

If the enemy knows the message being sent, things are even worse. For example, suppose the messages between a bank teller and the central office is encrypted by an exclusive-or with a random and never reused bit string. (This is known to be a probably unbreakable encryption technique.) A thief makes a deposit and the teller sends a record of the transaction to the central office. The thief now intercepts the encoded message and prevents it from being sent to the central office. Since the thief knows both the plaintext and encrypted version of the message, he can exclusive-or them together to recover the one-time random bit string. He can now substitute a different message—perhaps a record of a larger deposit—exclusive-ored with the now-known bit string. The moral is that a good encryption technique may be of no help in proving authorship. One can only trust the authenticity of messages to the extent that one can prove that such systematic changes are impossible.

A second disadvantage of using encryption for determining authorship is that there must be some redundancy in the communication. If most strings of bits were valid messages and if the encryption process didn't increase the length of a message, then B would accept most transmissions as valid. Thus, one must be careful about applying any data compression to the messages. With authentication tags, one can separate the secrecy aspects of communication from the authentication aspects. One can use the tags with or without encrypting the message, and the only redundancy needed is the extra bits needed for the tag itself.

Unlike digital signatures and encryption, a carefully constructed authentication tag system has the property that it is provably impossible for a forger to have more than an arbitrarily small chance of creating a message which the receiver will accept as genuine.

An authentication tag system can be formalized as follows: There is a set M of possible messages and a set T of authentication tags. For instance, M might be the set of all character strings of length 10,000 or less, and T might be the set of bit strings of length 100. There is also a (publicly known) set of functions F, where each function in F maps M into T. To use the system, A and B agree upon a secret "key" which specifies one of the functions f in F. When A transmits a message m in M, he also sends the authentication tag $f(m)$. B checks that f applied to the message she received is indeed the tag she received. If so, she has some assurance the received message is not a forgery. It must be impossible to find the function from a message and its tag. Otherwise the forgers might intercept, analyze and replace the message with one of their own. In fact, knowing the value of f on one message must give no information about the value of f on any other message. We will show a little later how this can be accomplished.

It is natural to wonder, if A and B have agreed upon any secret authentication function that nobody has any knowledge about, why won't the system automatically be completely unbreakable? Why restrict the choice of f to be from the set of functions F? It turns out that it is important to both the security and the usability of the system to specify F. Suppose we simply request A and B to collaborate on choosing a function. This requires time and imagination, and unless they are skilled, their choice is likely to be poor. For instance, if they decided that the first letters of each paragraph of the message should spell out "GIPWOLLEY," then most small changes by the forger to the message would leave these characters the same and thus would be undetected. So the secret function must be dependent on most of the characters in the message. If the communicants plan to calculate such a function by hand, they will most likely make a mistake. So either they must know how to program a computer, or there must be a software package designed to help them create functions. Such a package, if it is at all usable, will limit the choice of functions, and immediately we have to worry that the choice is sufficiently large to ensure security—which of course is the subject of this section. But let's suppose that the communicants are able to program. We suspect that a sizable percentage of programmers, when faced with the problem of creating a secret function which is dependent on all the bits of a message, would choose a function of the form, "Accept only messages which are congruent to 289 modulo 831," or perhaps "Compress the message by exclusive-OR-ing it together in 32-bit chunks, and accept only messages whose result is hex '7A28E910'." For both of these methods, it is easy to distort a message to another message which has the same authentication tag, even without knowing the specific integers involved. If the least common multiple of all numbers less than 1000 is added to a message, then its residue modulo anything less than 1000 (as well as modulo many other numbers) will be unchanged. Incidentally, this LCM is fewer than 1000/ln(2) bits long. To foil the exclusive-OR techniques is even easier: if S is any bit string such that the result of compressing S using 32-bit exclusive-OR's is hex "00000000," then $S \oplus M$ will compress to the same value that M compresses to. The point of this discussion is to illustrate why it is important to have a mathematical foundation behind the authentication scheme.

An authentication tag differs from a signature in that the receiver B can also create the authentication tag and thus a proper message. B cannot prove to a third party that the message originated with A since the third party will realize that B could have fabricated it. It may appear that this disadvantage is not shared by the digital signature technique. This is only partly true. If A wants to deny authorship of a message which he did indeed send, he can do so (at the cost of a bit of embarrassment) by professing that he accidentally revealed his secret signing function to B or to someone else.

On the other hand, if there is a universally trusted agency then authentication tags are as good as digital signatures for establishing the authorship of messages. This works as follows: Each individual person X shares his secret function f_x only with the agency. To send a message to B, A sends his message (tagged using his function) to the agency. The message must contain the name of the sender and receiver. The

agency first verifies that the message it receives has the correct tag for the sender named in the message. Then it stores a copy, appends the tag via f_B to the message, and forwards it on to B. When B verifies that the tag is correct, she accepts the message. B can now prove to anyone who trusts the agency that A sent her the message by simply asking the agency to check its records.

A second possible disadvantage of authentication tags is that only a finite number of messages can be sent using a particular function. We will prove that any unbreakable scheme can only be used a finite number of times, with that number dependent on the size of the key and the desired probability of guessing the correct tag. We will also show that our scheme approaches the theoretical bound on the minimum key size needed to send a given number of messages with a desired level of security.

To compensate for these disadvantage, we can construct an authentication scheme which is provably unbreakable. That is to say no resources other than knowledge of the key allow forgers to find the correct tag for a forged message. As the length of the tag grows the likelihood of a correct tag being appended to a message by forgers who don't know the key becomes more and more remote.

To make this more precise, we will say that an authentication system is *unbreakable with certainty p* if after a function f is randomly chosen and after forgers are given any message m and the corresponding tag $f(m)$, the forgers cannot find a different message m' for which they have better than a probability of p of guessing the correct tag. Note that this definition must hold for any m, even one chosen by the forgers.

To create an authentication system which is unbreakable with certainty p, we can simply choose T to have at least $1/p$ elements, and let F be a strongly universal$_2$ class of hash functions from M to T. If we let H' be the subset of H which maps m to $f(m)$, we see that the only information that the forgers have available is that the secret function is one of the functions in H'. However, the definition of strongly universal$_2$ implies that for any m' distinct from m, the proportion of functions in H' which map m' to any particular tag t' is $1/|T|$. Since $|T| \geqslant 1/p$, any choice the forger makes has no more than a probability of p of being correct.

Gilbert, MacWilliams and Sloane [10] have found rather complicated strongly universal$_2$ sets of functions for exactly this purpose. The difficulty with their set and with other previously known strongly universal$_2$ sets is that the set of functions is so large that specifying a function in the class requires a key at least as long as the original message. It is desirable to use a key considerably shorter than the message. A second problem is that only one message per key can be sent, since knowledge of two message-tag pairs may give some information about the value of the function on some third message. We will solve these problems separately.

3. A Small, Almost Strongly Universal$_2$ Class

We wish to construct a set of hash functions from some large space A' to a space B'. In the above, A' is the set of messages and B' is the set of possible tags. Let a' and b' be the length of the messages and tags, respectively. Let $s = b' + \log_2 \log_2(a')$. Let H be some strongly universal$_2$ class of functions which map bit strings of length $2s$ to ones of length s. The multiplicative scheme of [6] is fine for this purpose. Each member of H' will be constructed from a sequence of length $\log_2 a' - \log_2 b'$ of members of H. Suppose $f_1, f_2, \ldots$ is some such sequence. We will specify how to apply the associated member f' of H' to a message. The message is broken into substrings of length $2s$. If necessary, the last substring should be padded with blanks. Thus, the message will be broken into $\lceil a'/2s \rceil$ substrings. f_1 is applied to all the substrings and the resulting substrings are concatenated. By concatenating the resulting substrings, we have obtained a string whose length is roughly half the originals string's length. This process is repeated using $f_2, f_3, \ldots$ until only one substring of length s is left. The tag (i.e., the result of the hash function f') is the low-order b' bits of this substring. The key needed to specify f' is the concatenation of the keys needed to specify $f_1, f_2, \ldots$. The multiplicative scheme suggested in [6] has a key roughly twice the size of the input. If this class is used for H, the size of the key for H' will be $4s \log_2(a')$. Thus, the key is roughly four times the length of the tags times the log of the length of the message. By comparison, the multiplicative scheme by itself would have a key whose length was twice that of the message.

Observe that assuming the functions in H can be evaluated in time proportional to s, the functions in H' also can be evaluated in time proportional to the length of the message.

The sense in which H' is "almost" strongly universal$_2$ is given in the following theorem.

THEOREM. *Given any two distinct messages m_1 and m_2 and any two tag values t_1 and t_2, the number of functions which take m_1 to t_1 is $1/|B'|$ times the total number of functions. However, fewer than $2/|B'|$ of these functions will also take m_2 to t_2.*

Proof sketch. Each time we halve the length of the messages, there is a small $(1/(2^s))$ chance that the two resulting strings are now identical. Since we iterate the halving process $\log_2 a' - \log_2 b'$ times, the chance that the two strings are identical at the next to last step is at most $\log_2 a'/(2^s)$, which is equal to $1/(2^{b'})$. Now the fact that the function that does the last reduction is chosen from a strongly universal$_2$ class can be used to show that m_1 will be taken into any tag with equal probability, and as long as the penultimate strings were different, m_2 will also be taken into any string with probability equal to $1/|B'|$. Thus, if $t_1 \neq t_2$, then less than $1/|B'|$ of the functions will take m_2 to t_2, and otherwise, less than $2/|B'|$ will.

The above theorem can be contrasted with the definition of strongly universal$_2$, which says that $1/|B'|$ of the functions must take m_1 to t_1, and that $1/|B'|$ of these functions will also take m_2 to t_2. In terms of the authentication scheme, the theorem

states after the enemy knows one message-tag pair, he can do no better than to find another message-tag pair which has probability $2/|B'|$ of being correct. Thus, the scheme is unbreakable with certainty $2/|B'|$, and this certainty can be made smaller than any predetermined value.

4. Authenticating Multiple Messages

The above method does not allow us to tag more than one message using the same function, since once the enemy knows two message-tag pairs, he may be able to determine more such pairs. One way around this problem might be to use a universal$_n$ function, which would allow us to send $n-1$ messages, but a better method is as follows: Let F be a strongly universal$_2$ set of functions from M to B, where B is the set of bit strings of length k. Each message in M must contain a message number between 1 and n. The secret key shared by the sender and receiver now consists of two parts. The first part specifies a function f in F. The second part of the key is a sequence $(b_1,..., b_n)$ of elements of B. The sender must be certain never to send two messages with the same message number. To create the authentication tag t_i for the message m_i (a message with message number i) the sender first calculates $f(m_i)$ and then exclusive-or's this result with b_i. Since each message contains a message number, the receiver can duplicate this process to verify the tag is correct. (If a message is unnumbered, it is automatically rejected as a forgery.) We wish to show that this scheme is unbreakable with certainty $1/(2^k)$.

Theorem. *Suppose some key $(f, (b_1,..., b_n))$ has been chosen randomly from the set of keys. Let $m_1,..., m_n$ be any n messages, with the restriction that the message numbers must all be different (we assume that m_i has number i.) Suppose a forger knows only the set F and the set of messages and their corresponding tags $t_i = f(m_i) \oplus b_i$. (We use $\oplus$ to denote the exclusive-or operation.) Then there is no new message (with any message number) for which the forger has a better than $1/(2^k)$ chance of correctly guessing the tag.*

Proof. Suppose the forger wishes to guess the tag to the new message m. Without loss of generality, we assume m has the message number 1. For each t in B, define $S_t = \{(g, b) \mid g \in F,\ b \in B,\ g(m_1) \oplus b = t_1$ and $g(m) \oplus b = t\}$. In other words, S_t is the set of partial keys (partial since only the first of n elements of b are specified) which are consistent with the fact that m_1 has tag t_1, and which give the bogus message m the tag t. It isn't hard to show that since F is strongly universal$_2$, each of the S_t's have the same size. Further, there is exactly one way to extend each partial key in S_t to a complete key which also assigns tag t_i to message m_i for $i = 2,..., n$ (namely, let $b_i = g(m_i) \oplus t_i$.) Thus, of all the keys which are consistent with the information which the forger has available, as many will assign to m any one tag as any other tag. Thus, the forger's probability of guessing the correct tag for m is $1/(2^k)$.

A similar theorem holds when you use an almost strongly universal$_2$ class.

The next theorem shows that the number of bits required by this scheme is asymptotically optimal. To make this precise, we define OPT(n) to be the smallest key size needed to prevent a forger from having more than a specified probability of success at forging at least one out of n messages. We prove that as n approaches infinity, the number of bits our scheme uses to send n messages with the same security, divided by OPT(n), approaches 1. A similar theorem has also been proved by Fak [8]. We will consider a slightly more general scenario in which the messages need not have message numbers. Suppose a function has been selected from a set F. The forger chooses a message m_1 and tries to guess the correct tag. He is then told the correct tag t_1. Now the forger selects a second message m_2, trys to guess the tag, and then is told the correct tag t_2. This process is repeated n times. If we wish, we may require the forger to choose each message from a restricted subset of the set of all messages, or we may even have a fixed sequence of messages—these variations don't affect the following theorem.

THEOREM. *In the above scenario, if the forger's probability of success on his ith guess is* $\leqslant p_i$, *then F must contain at least* $1/(p_1 p_2 \cdots p_n)$ *functions.*

Proof. Let $F_0 = F$ and $F_k = \{f \in F \mid f(m_i) = t_i \text{ for } i = 1,..., k\}$. The forger might use the following strategy in his guessing: After choosing the ith message, he enumerates the set F_{i-1}, randomly chooses a member of it, and guesses the tag $f(m_i)$. Since this has $\leqslant p_i$ chance of success, it must be the case that $|\{f \in F_{i-1} \text{ s.t. } f(m_i) = t_i\}| \leqslant p_i |F_{i-1}|$. The set on the left-hand side is F_i, so we have $|F_{i-1}| \geqslant (1/p_i)|F_i|$. This is true for each i, so we have $|F_0| \geqslant (1/p_1) \cdots (1/p_n)|F_n|$. The theorem follows since $F_0 = F$ and $|F_n| \geqslant 1$.

COROLLARY. *When* $p_1 = p_2 = \cdots = p$, *then it requires at least* $n(-\log_2(p))$ *bits to specify a randomly chosen member of F for any scheme which is unbreakable with certainty p and be able to send n messages. Note that the scheme we presented above requires* $n(-\log_2(p)) + K$ *bits, where K is the number of bits needed to specify an element of a strongly universal*$_2$ *class of hash functions.*

5. TESTING SET EQUALITY

In this section we present a linear time algorithm for probabilistically testing many sets for equality. More formally, suppose we have a sequence of requests which may name an arbitrary number of sets and an arbitrary number of elements. Each request can be one of the following four commands:

ADD(x, S)—Adds the element x to the set named S. This operation may not be used if x is already a member of S. If S has not previously been used in any commands, it is treated as a name for the empty set.

DELETE(x, S)—Removes element x from the set named S. This may only be used if x is a member of S.

TEST(S_1, S_2)—Returns "true" if the sets named by S_1 and S_2 are equal, and "false" otherwise. TEST may possibly call two sets equal when they are not, but it cannot call them unequal when they are actually equal.

FIND(S)—Returns a list of the names of sets which are equal to S. S will always be included in this list. This list may, by mistake, contain the names of sets which are actually not equal to S, but no equal set can be overlooked.

Given any $\varepsilon > 0$, we can process a sequence of requests in expected time linear in the number of requests times $(-\log(\varepsilon))$. The error probability for each TEST request will be less than ε, and for each FIND request, less than ε times the number of set names in existence at the time of the request. In addition to the four requests mentioned above, one may also have either COPY and DIFF requests, or ELEMENT, LIST and CONTAINERS requests. These are defined as follows:

COPY(S_1, S_2)—S_1 becomes another name for the set named by S_2.

DIFF(S_1, S_2)—Assigns S_1 to be the symmetric difference of S_1 and S_2.

ELEMENT(x, S)—Returns "true" if x is a member of the set named S and "false" otherwise. This response is always correct.

LIST(S)—Returns a list of the elements of the set named S.

CONTAINERS(x)—Returns a list of the names of all the currently-existing sets which contain x.

The algorithm which performs these operations is a modification of a known heuristic (see [13]) which, when two sets are unequal, will sometimes rapidly determine that they are unequal. (The rest of the time, the heuristic is unable to decide if they are equal or not.) We modify the scheme so that given any sequence of requests which create unequal sets, there is a high probability that the algorithm will determine they are unequal. The probability may be made so high that the lack of showing they are unequal is sufficient evidence for the particular application to treat them as equal. For instance, one can make the probability less than the probability that the computer would make a hardware error during the additional time which would be required to perform a complete test.

We view this algorithm as a tool to be used for other applications, rather than because we are directly interested in testing set equality. Thus, it is important that we can prove that the probability of making a mistake is not dependent on the particular characteristics of the input string, otherwise our application could only use sequences of requests which had those characteristics. We accomplish this by constructing a class of algorithms and showing that for the worst input, there is only a low probability that a randomly chosen algorithm will produce an error.

The basic idea behind the set equality tester is that with each set name S_i, there is an "abbreviation" V_i. When a change is made to a set, the abbreviation is updated, and two sets are claimed to be equal if their abbreviations are equal. Associated with

each abbreviation, there must be a list of the set names which currently have that abbreviation. Of course, we store the abbreviations (and their corresponding lists) in a hash table. Incidentally, since the V_i's are produced by hashing, it is sufficient simply to mask off bits as the hash function used to store them in this table.

To be more specific, let G be a group with operation $\oplus$ and denote the inverse of x in G by x^{-1}. Let h be a hash function chosen from a strongly universal$_\omega$ set which maps elements of the sets being constructed into G. One might implement this scheme with G being the set of bit strings of a certain length, and $\oplus$ being exclusive-or. This operation is easy to perform, and the inverse operation—the identity function—is even easier.

The first time a set name S_i is used, its corresponding V_i is set to the identity element of G. The requests are handled as follows:

ADD(x, S_i) is implemented by updating V_i to $V_i \oplus h(x)$, and removing S_i from the list associated with the old value of V_i and adding it to the new. (In order to be able to perform the removal in constant time, we must keep a pointer with the name S_i which points to where this name is in the list associated with V_i.)

DELETE(x, S_i) is implemented similarly, except $h(x)^{-1}$ is used in place of $h(x)$.

TEST(S_i, S_j) is implemented by comparing V_i with V_j.

FIND(S_i) simply returns the list associated with V_i.

COPY(S_i, S_j) removes S_i from V_i's list, sets $V_i = V_j$, and adds S_i to V_j's list.

DIFF(S_i, S_j) sets V_i to $V_i \oplus V_j$, and updates the lists associated with the old and new abbreviations appropriately. We observe that for DIFF to work correctly, x^{-1} must be x for all $x \in G$, which is the case when the group operation is exclusive-or.

ELEMENT, LIST and CONTAINERS—to be able to perform these operations, one must maintain several additional hash tables. One contains all pairs (x, S_i) which have been the subject of an ADD request but not a DELETE request. This table is used to answer ELEMENT requests. A second hash table associates with each x a list of the sets which contain x. This table is consulted to perform a CONTAINERS request. In order to be able to update this list in constant time when a DELETE(x, S_i) request is encountered, it is necessary to associate with the pair (x, S_i) in the first hash table a pointer to S_i in the list associated with x. Finally, to be able to answer LIST(S_i) requests, we need to associate to the name S_i a list of elements which are currently in S_i. Again, we need to have a pointer stored with the pair (x, S_i) which will enable us to remove x from the list associated with S_i in constant time when a DELETE(x, S_i) request is encountered.

These techniques may be modified to deal with multi-sets instead of sets. A multi-set is similar to a set, except one counts how many times an element has been inserted into the multi-set. Two multi-sets are equal only if each element has been inserted the same number of times. In the above discussion we suggested using exclusive-or as the operation $\oplus$. If one uses ordinary addition instead, then a scheme

similar to the one described above can process the requests, ADD(x, k, S), which adds k copies of x to S (we no longer need the restriction that x was not already in S), DELETE(x, k, S), TEST(S_i, S_j) and FIND(S_i). Analogously to the set equality case, one can additionally add either the COPY and MULTISET_UNION commands, or the ELEMENT, LIST and CONTAINERS commands. Here, ELEMENT(x, S_i) returns how many times x occurs in S_i. A difficulty with using addition instead of exclusive-or is that the V_i's can get arbitrarily large, and thus the assumption that the operations take constant time may no longer be valid. However, if the sequence of requests never makes very large multi-sets, then this will not be a problem.

We now mention three applications of set equality testers. The generation of LR or LALR parse tables can be speeded up since a considerable amount of time is spent checking if a set of items is new or has already been generated. Here, the FIND command is particularly useful. Secondly, a graph may be represented as a set of nodes and a set of edges. Testing labeled graphs for equality is now easy. Finally, the memory state of a computer may be represented as a set of pairs, each pair consisting of an address and the value stored in that address [3]. If a value in memory is changed, we delete the pair consisting of the address and the old value, and add the pair of the address and the new value. We can use our method to see if a memory state has been seen before, and thus whether the program is looping (see [4]).

It is conceivable that the equality test could be extended to other requests. If the equality test could be extended adequately, a good part of a language like SETL might fit in such a scheme. However, a recent result [14] suggests that it is impossible to find a fast test for two sets being disjoint.

6. Implementing Strongly Universal$_\omega$ Sets

We can create a strongly universal$_\omega$ set of functions as follows: Each function will map the set of names of set elements (that is, the x_i's of the set equality tester) to the group G. The techniques of [6] give us the ability to use an associative memory which requires constant expected time per request. We assume the ability to generate random numbers. We will use these two abilities to create a partial function f defined only on all the inputs we have seen. $f(x)$ is computed as follows: if there is a value associated with x in the associative memory, then $f(x)$ is that value. Otherwise, $f(x)$ equals a value chosen randomly from G, and x and that value are stored in the associative memory. Thus, the value of hash on any element is independent of its values on any other elements.

We can now prove:

THEOREM. *The probability that the set equality tester described above incorrectly answers a* TEST *request is no more than the reciprocal of the number of elements in the group.*

Proof. There are two imaginable errors: our algorithm might say two equal sets were unequal, or it might say two unequal sets were equal. If $V_i \neq V_j$ then sets S_j and S_i must be unequal. Thus, the first type of error cannot occur. We will now find the probability of saying two unequal sets are equal.

If sets S_j and S_i are unequal then either set S_j or set S_i must have an element not contained in the other. We enumerate the elements of set S_i by $x_{i,k}$. Without loss of generality we will assume S_i has an element, $x_{i,1}$ not contained in S_j. If $V_i = V_j$ then

$$h(x_{i,1}) \oplus h(x_{i,2}) \cdots = h(x_{j,1}) \oplus h(x_{j,2}) \cdots$$

which is true if and only if

$$h(x_{i,1}) = h(x_{i,2})^{-1} \oplus h(x_{i,3})^{-1} \cdots \oplus h(x_{j,1}) \oplus h(x_{j,2}) \cdots .$$

It follows from the definition of strongly universal$_\omega$ that for any $b \in G$ (where G is the range of the hash function) the probability that $h(x_{i,1}) = b$ is independent of the value of b and the values of h on other elements, so the probability that $h(x_{i,1}) = h(x_{i,2})^{-1} \oplus h(x_{i,3})^{-1} \cdots \oplus h(x_{j,1}) \oplus h(x_{j,2}) \cdots$ is the reciprocal of the number of elements in G. Thus, the probability that $V_i = V_j$ when $S_i \neq S_j$ is the reciprocal of the number of elements in G.

Notice that one indeed needs to use a strongly universal$_\omega$ set of functions, since the above reasoning requires that the value of $h(S_1)$ be independent of the value of h on all the other elements of sets S_i and S_j.

If the group elements are all bit string of length 100, the error probability per test would be $1/2^{100}$. This might well be less than the probability of a machine error in the extra time necessary to do a complete check.

REFERENCES

1. L. ADLEMAN, Private communication, Dec. 1977.
2. G. BRASSARD, Relativized cryptography, *in* "Proceedings, Twentieth Annual Symposium on Foundations of Computer Science," pp. 383–391, October, 1979.
3. D. BROWN, Kraft storage and access for list implementation, *in* "Proceedings, Twelfth Annual ACM Symposium on Theory of Computing, April 1980," pp. 100–107.
4. J. COCKE, private communication (May, 1978).
5. J. L. CARTER, J. GILL, R. FLOYD, G. MARKOWSKY, AND M. WEGMAN, Exact and approximate membership testers, *in* "Proceedings, Tenth Annual ACM Symposium on Theory of Computing, May 1978," pp. 59–65.
6. J. L. CARTER AND M. N. WEGMAN, Universal classes of hash functions, *J. Comput. System Sci.* **18**, No. 2 (April 1979), 143–154.
7. W. DIFFIE AND M. HELLMAN, New directions in xryptography, *IEEE Trans. Inform. Theory* (November 1976).
8. V. FAK, Repeated use of codes which detect deception, *IEEE Trans. Inform. Theory* **25**, No. 2 (March, 1979), 233–234.
9. R. FAGIN, J. NIEVERGELT, N. PIPPENGER, AND H. R. STRONG, Extendible hashing—A fast access method for dynamic files, *ACM Transactions Database Systems* **4**, No. 3 (1979), 315–344.
10. E. N. GILBERT, F. J. MACWILLIAMS, AND N. J. A. SLOANE, Codes which detect deception, *Bell System Tech. J.* (March 1947), 405–424.

11. J. GILL, "Probabilistic Turing Machines and Complexity of Computation," Ph. D. dissertation, Department of Mathematics, University of California at Berkeley, June, 1972.
12. R. RIVEST, A. SHAMIR, AND L. ADLEMAN, "On Digital Signatures and Public-key Cryptosystems," MIT/LCS/TM-82.
13. J. T. SCHWARTZ, private communication (implemented in 1973).
14. A. YAO, Some complexity questions related to distributive computing, *in* "Proceedings, Eleventh Annual ACM Symposium on Theory of Computing, May 1979," pp. 209–213.