Bucket Hashing and its Application to Fast Message Authentication

Phillip Rogaway

Department of Computer Science, University of California, Davis, CA 95616, USA. rogaway@cs.ucdavis.edu

Abstract. We introduce a new technique for generating a message authentication code (MAC). At its center is a simple metaphor: to (non-cryptographically) hash a string x, cast each of its words into a small number of *buckets*; xor the contents of each bucket; then collect up all the buckets' contents. Used in the context of Wegman–Carter authentication, this style of hash function provides the fastest known approach to software message authentication.

1 Introduction

MESSAGE AUTHENTICATION. Message authentication is one of the most common cryptographic aims. The setting is that two parties, a signer S and verifier V, share a (short, random, secret) key, a. When S wants to send V a message, x, S computes for it a *message authentication code* (MAC), $\mu \leftarrow \mathsf{MAC}_a(x)$, and S sends V the pair (x, μ). On receipt of (x', μ'), verifier V checks that $\mathsf{MACV}_a(x', \mu') = 1$.

To describe the security of a message authentication scheme an adversary is given an oracle for $\mathsf{MAC}_a(\cdot)$. Following [11], she is declared *successful* if she outputs an (x^*, t^*) such that $\mathsf{MACV}_a(x^*, t^*) = 1$ but x^* was never asked of the $\mathsf{MAC}_a(\cdot)$ oracle. For a scheme to be "good," reasonable adversaries should rarely succeed.

SOFTWARE-EFFICIENT MACS. In the current computing environment it is often necessary to compute MACs frequently and over strings which are commonly hundreds to thousands of bytes long. Despite this, there will usually be no special-purpose hardware to help out: MAC generation and verification will need to be done in software on a conventional workstation or personal computer. So to reduce the impact of message authentication on the machine's overall performance, and to facilitate more pervasive use of message authentication, we need substantially faster techniques. That is what this paper provides.

TWO APPROACHES TO MESSAGE AUTHENTICATION. The fastest software MACs in common use today are exemplified by $\mathsf{MAC}_a(x) = h(x\|a)$, with h a (software-efficient) cryptographic hash function, such as h =MD5. Such methods are described in [22]. A scheme like this might seem to be about as software-efficient as one might realistically hope for: after all, we are computing one of the fastest types of cryptographic primitives over a string nearly identical in length to that which we want to authenticate. But it is well-known that this reasoning is specious: in particular, Wegman and Carter [23] showed back in 1981 that we do not have to "cryptographically" transform the entire string x.

In the Wegman–Carter approach communicating parties S and V share a secret key a which specifies both a random pad p and a hash function h drawn randomly from a strongly universal$_2$ family of hash functions $\mathcal{H}$. (Recall that $\mathcal{H}$ is strongly universal$_2$ if for all $x_0 \neq x_1$, the random variable $h(x_0) \parallel h(x_1)$ is uniformly distributed.) To authenticate a message x, the sender transmits $h(x)$ xor-ed with the next piece of the pad p. The thing to notice is that x is transformed first by a non-cryptographic operation (universal hashing) and only then is it subjected to a cryptographic operation (encryption)—now applied to a much shorter string.

As it turns out, to make a good MAC you don't need to start from a strongly universal$_2$ family. Carter and Wegman [7] also introduced the notion of an almost universal$_2$ family, $\mathcal{H}$. This must satisfy the weaker condition that $\Pr_{h\in\mathcal{H}}[h(x_0) \neq h(x_1)]$ is small for all $x_0 \neq x_1$. As observed by Stinson [19], an almost universal$_2$ family can easily be turned into an almost strongly universal$_2$ family (which can, in turn, be used to authenticate ones messages). In this manner the problem of finding an efficient MAC has effectively been reduced to that of finding an efficient almost universal$_2$ family of hash functions.

OUR CONTRIBUTION. This paper provides a novel almost universal$_2$ family of hash functions. We call our hash family *bucket hashing*. It is distinguished by its member functions being extremely fast to compute—as few as 6 elementary machine instructions per word (independent of word size) for the version of bucket hashing we concentrate on in this paper. Putting such a family of hash functions to work in the framework of known constructions gives rise to the most efficient software MACs now known. For example, we estimate that a MAC so constructed can authenticate (reasonably long) messages in about 10–15 instructions per 32-bit word. For comparison, authenticating messages using an MD5-based technique requires some 40–50 instructions per word [21].

A bucket hash MAC has advantages in addition to speed. Bucket hashing is a *linear* function —it is a special case of matrix multiplication over GF(2)— and this linearity yields many pleasant characteristics for a bucket hash MAC. In particular, bucket hashing is *parallelizable,* since each word of the hash is just the xor of certain words of the message. Bucket hashing is *incremental* in the sense of [2] with respect to both **append** and **substitute** operations. Finally, the only processor instructions a bucket hash needs are word-aligned **load**, **store**, and **xor**; thus a bucket hash MAC is essentially endian-indifferent.

One might worry that the linearity of bucket hashing might give rise to some "weakness" in a MAC which exploits it. But it does not. A bucket hash MAC, like any MAC which follows the Wegman–Carter paradigm, enjoys the assurance advantages of provable security. Moreover, this provable security is achieved under extremely "tight" reductions, so that an adversary who can successfully break the MAC can break the underlying cryptographic primitive (e.g., DES) with essentially identical efficacy. In contrast, a scheme like $\mathsf{MAC}_a(x) = h(x\|a)$

is not known to be provably secure under *any* standard assumption on h.

RELATED WORK. The general theory of unconditional authentication was developed by Simmons; see [18] for a survey. As we have already explained, the universal-hash-and-then-encrypt paradigm is due to Wegman and Carter [23]. The idea springs from their highly influential [7].

In Wegman–Carter authentication the size of the hash family corresponds to the number of bits of shared key—one reason to find smaller families of universal hash functions than those of [7, 23]. Stinson does this in [19], and also gives general results on the construction of universal hash functions. We exploit some of these ideas. Subsequent improvements (rooted in coding theory) came from Bierbrauer, Johansson, Kabatianskii and Smeets [6], and Gemmell and Naor [9].

The above work concentrates on unconditionally-secure authentication. Brassard [5] first connects the Wegman–Carter approach to the complexity-theoretic case. The complexity-theoretic notion for a secure MAC is a straightforward adaptation of the definition of a digital signature due to Goldwasser, Micali and Rivest [11]. Their notion of an adaptive chosen message attack is equally at home for defining an unconditionally-secure MAC. Thus we view work like ours as making statements about unconditionally-secure authentication which give rise to corresponding statements and concrete schemes in the complexity-theoretic tradition. To make this translation we regard a finite pseudorandom function (PRF) as the most appropriate tool. Bellare, Kilian and Rogaway [3] were the first to formalize such objects, investigate their usage in the construction of efficient MACs, and suggest them as a desirable starting point for practical, provably-good constructions. Finite PRFs are a refinement of the PRF notion of Goldreich, Goldwasser and Micali [10] to take account of the fixed lengths of inputs and outputs in the efficient primitives of cryptographic practice.

Zobrist [25] gives a hashing technique which predates [7] and which, in implementation, essentially coincides with the scheme $\mathcal{H}_M$ described in Section 2 and due to [7]. Arnold and Coppersmith [1] give an interesting hashing technique which allows one to map a set of keys k_i into a set of corresponding values v_i using a table only slightly bigger than $\sum_i v_i$. The proof of our main technical result is somewhat reminiscent of the analysis in [1].

Lai, Rueppel and Woolven [14], Taylor [20], and Krawczyk [12] have all been interested in computationally efficient MACs. The last two works basically follow the Wegman–Carter paradigm. In particular, Krawczyk obtains efficient message authentication codes from hash families which resemble traditional cyclic redundancy codes (CRCs), and matrix multiplication using Toeplitz matrices. Though originally intended for hardware, these techniques are fast in software, too. We recall Krawczyk's CRC-like hash in Section 2.

2 Preliminaries

This section provides background drawn from Carter and Wegman [7, 23], Stinson [19], and Krawczyk [12]. The only new material is the (simple) scheme $\mathcal{H}_N$ and the statement of Theorem 7. Proofs are omitted.

A *family of hash functions* is a finite multiset $\mathcal{H}$ of string-valued functions, each $h \in \mathcal{H}$ having the same nonempty domain $A \subseteq \{0,1\}^*$ and range $B \subseteq \{0,1\}^*$.

Definition 1. [7] A family of hash functions $\mathcal{H} = \{h : A \to B\}$ is **ϵ-almost universal$_2$**, written ϵ-AU$_2$, if for all $x_0, x_1 \in A$, $x_0 \neq x_1$, $\Pr_{h \in \mathcal{H}}[h(x_0) = h(x_1)] \leq \epsilon$. Family $\mathcal{H}$ **ϵ-almost XOR universal$_2$**, written ϵ-AXU$_2$, if for all $x_0, x_1 \in A$, $y \in B$, $x_0 \neq x_1$, $\Pr_{h \in \mathcal{H}}[h(x_0) \oplus h(x_1) = y] \leq \epsilon$.

The value of $\epsilon^* = \max_{x_0 \neq x_1}\{\Pr_h[h(x_0) = h(x_1)]\}$ is called the *collision probability*. For us, the principle measures of the worth of an AU$_2$ hash family are how small is ϵ^* and how fast can one compute its functions.

To make a fast MAC we will want to "glue together" various universal hash families. The following are our the basic methods for doing this.

First we need a way to make the domain of a hash family bigger. Let $\mathcal{H} = \{h : \{0,1\}^a \to \{0,1\}^b\}$. By $\mathcal{H}^m = \{h : \{0,1\}^{am} \to \{0,1\}^{bm}\}$ we denote the family of hash functions whose elements are the same as in $\mathcal{H}$ but where $h(x_1 x_2 \cdots x_m)$, for $|x_i| = a$, is defined by $h(x_1) \parallel h(x_2) \parallel \cdots \parallel H(x_m)$.

Proposition 2. [19] If $\mathcal{H}$ is ϵ-AU$_2$ then $\mathcal{H}^m$ is ϵ-AU$_2$.

Sometimes one needs a way to make the collision probability smaller. Let $\mathcal{H}_1 = \{h : \{0,1\}^a \to \{0,1\}^{b_1}\}$ and $\mathcal{H}_2 = \{h : \{0,1\}^a \to \{0,1\}^{b_2}\}$ be families of hash functions. By $\mathcal{H}_1 \& \mathcal{H}_2 = \{h : \{0,1\}^a \to \{0,1\}^{b_1+b_2}\}$ we mean the family of hash functions whose elements are pairs of functions in $\mathcal{H}_1$ and $\mathcal{H}_2$ and where $(h_1, h_2)(x)$ is defined as $h_1(x) \parallel h_2(x)$.

Proposition 3. If $\mathcal{H}_1$ is ϵ_1-AU$_2$ and $\mathcal{H}_2$ is ϵ_2-AU$_2$ then $\mathcal{H}_1 \& \mathcal{H}_2$ is $\epsilon_1\epsilon_2$-AU$_2$.

Next is a way to make the image of a hash function shorter. Let $\mathcal{H}_1 = \{h : \{0,1\}^a \to \{0,1\}^b\}$ and $\mathcal{H}_2 = \{h : \{0,1\}^b \to \{0,1\}^c\}$ be families of hash functions. Then by $\mathcal{H}_2 \circ \mathcal{H}_1 = \{h : \{0,1\}^a \to \{0,1\}^c\}$ we mean the family of hash function whose elements are pairs of functions in $\mathcal{H}_1$ and $\mathcal{H}_2$, and where $(h_1, h_2)(x)$ is defined as $h_2(h_1(x))$.

Proposition 4. [19] If $\mathcal{H}_1$ is ϵ_1-AU$_2$ and $\mathcal{H}_2$ is ϵ_2-AU$_2$ then $\mathcal{H}_2 \circ \mathcal{H}_1$ is $(\epsilon_1 + \epsilon_2)$-AU$_2$.

Composition can also be used to turn an AU$_2$ family into AXU$_2$ family:

Proposition 5. [19] Suppose $\mathcal{H}_1 = \{h : A \to B\}$ is ϵ_1-AU$_2$, and $\mathcal{H}_2 = \{h : B \to C\}$ is ϵ_2-AXU$_2$. Then $\mathcal{H}_2 \circ \mathcal{H}_1 = \{h : A \to C\}$ is $(\epsilon_1 + \epsilon_2)$-AXU$_2$.

Now given a family of hash functions $\mathcal{H} = \{A \to \{0,1\}^b\}$ we can construct from it a MAC. In the scheme we denote WC[$\mathcal{H}$], the signer S and verifier V share a random element $h \in \mathcal{H}$, as well as an infinite random string $p = p_0 p_1 p_2 \cdots$, where $|p_i| = b$. Together, h and p comprise the shared secret. The signer maintains a counter, cnt, which is initially 0. We let the MAC of x under key (h,p) be given by $(\text{cnt}, p_{\text{cnt}} \oplus h(x))$. The scheme is stateful: after the MAC of x is computed, the number cnt is incremented. The following theorem says that it is impossible (regardless of time, number of queries, or amount of MACed text) to forge with probability exceeding the collision probability (see Appendix B for definitions):

Theorem 6. **[23, 12]** Let $\mathcal{H}$ be ϵ-AXU_2 and suppose E (t,q,μ,δ)-breaks $\mathrm{WC}[\mathcal{H}]$. Then $\delta \leq \epsilon$.

A natural complexity-theoretic variant is to use, instead of the random pad p, an index $a \in \{0,1\}^{\kappa}$ into a finite PRF F. The signer S maintains a counter $\text{cnt} \in \{0,1\}^{l}$. Function F_a maps l-bit strings to b-bit ones. Now S and V share a random $a \in \{0,1\}^{k}$ and a random $h \in \mathcal{H}$. The cnt^{th} MAC of x under key (h,a) is given by $(\text{cnt}, F_a(\langle\text{cnt}\rangle_l) \oplus h(x))$. At most 2^l messages may be MACed before the key must be changed. We call the scheme just described $\mathrm{WC}[\mathcal{H},F]$. Its security is described by the following:

Theorem 7. Let $\mathcal{H} = \{h : A \to \{0,1\}^b\}$ be an ϵ-AXU_2 family of hash functions, let $F : \{0,1\}^{\kappa} \times \{0,1\}^{l} \to \{0,1\}^{b}$ be a finite PRF, and let E be an adversary which (t,q,μ,δ)-breaks $\mathrm{WC}[\mathcal{H},F]$. Suppose one can in time $T_{\mathcal{H}}$ compute a representation of a random element $h \in \mathcal{H}$, and from this representation one can compute h-values on q strings totalling μ bits in $T_h(q,\mu)$ time. Then there is an algorithm D which $(t+\Delta t,\ q+1,\ \delta-\epsilon)$-breaks F, where $\Delta t = O(T_h(q,\mu) + T_{\mathcal{H}} + ql + qb)$.

The value of Δt would usually be insignificant compared to t.

In the above two theorems the forging probability is independent of the number of queries (q) and the length of the queried messages (μ). This is a significant advantage compared with constructions based on the iterated application of a finite PRF.

We emphasize that the signer is stateful in both $\mathrm{WC}[\mathcal{H}]$ and $\mathrm{WC}[\mathcal{H},F]$. This improves these schemes' security (compared with using a random value) at little practical cost. Note that the verifier is not stateful, as we have chosen a definition of security (see Appendix B) which does not address "replay attacks."

We end this section with some constructions for software-efficient hash families. In the first $A = \{0,1\}^a$ is the strings we want to hash and $B = \{0,1\}^b$ is the space we want to hash our strings into. An element of $\mathcal{H}_M[a,b]$ is described by a $b \times a$ binary matrix. If $h \in \mathcal{H}_M$ is such a matrix we define $h(x)$ as the product hx of matrix h and column vector x.

Proposition 8. [7] $\mathcal{H}_M[a,b]$ is 2^{-b}-AXU_2.

We can modify this family to trade ϵ-AXU_2 for ϵ-AU_2 and some speed. Let $A = \{0,1\}^{a+b}$ be the strings we want to hash and let $B = \{0,1\}^b$ be the space we want to hash into. Each $h \in \mathcal{H}_N[a+b,b]$ is described by a $b \times a$ binary matrix. If $h \in \mathcal{H}_N[a+b,b]$ is such matrix, we define $h(x_1 e_0)$, where $|x_1| = a$ and $|x_0| = b$, by $hx_1 \oplus x_0$—the product of matrix h and column vector x_1, xor-ed with x_0.

Proposition 9. $\mathcal{H}_N[a+b,b]$ is $2^{-b}-\mathrm{AU}_2$.

Here's a final construction, this one from [12]. An element of the hash family $\mathcal{H}_K[a,b] = \{h : \{0,1\}^a \to \{0,1\}^b\}$ is described by an irreducible polynomial over GF[2] of degree b. Given such a polynomial h, the value of $h(M)$, where $M \in \{0,1\}^a$, is the coefficients of $M(X)X^b \bmod h(X)$, where X is a formal variable and $M(X)$ is the polynomial over X whose coefficients are given by M.

Theorem 10. [12] $\mathcal{H}_K[a,b]$ is $\frac{a+b}{2^b-1}$-AXU_2.

3 Bucket Hashing

In software, hash families such as $\mathcal{H}_N$, $\mathcal{H}_M$ and $\mathcal{H}_K$ are all reasonably efficient. Still, for reasonably long message, we can do quite a bit better.

3.1 Defining $\mathcal{H}_B[w,n,N]$

Fix a "word size" $w \geq 1$. For some particular $n \geq 1$ and $N \geq 3$ we will be hashing from $A = \{0,1\}^{wn}$ to $B = \{0,1\}^{wN}$. We call N the "number of buckets" for reasons soon to be clear. As a typical example, take $w = 32$, $n = 1024$, and $N = 140$. We require that $\binom{N}{3} \geq n$.

Each $h \in \mathcal{H}_B[w,n,N]$ is specified by a length-n list of cardinality-3 subsets of $[0..N-1]$. We denote this list by $h = (h_0, \ldots, h_{n-1})$. We denote the three elements of h_i by $h_i = \{h_i^1, h_i^2, h_i^3\}$.

Choosing a random h from $\mathcal{H}_B[w,n,N]$ means choosing a random length-n list of three-element subsets of $[0..N-1]$ subject to the constraint that no two of these sets are the same. That is, for all $i \neq j$, we demand that $h_i \neq h_j$.

Let $h = (h_0, \ldots, h_{n-1}) \in \mathcal{H}_B[w,n,N]$ be as above. Then $h(x)$ is defined by the following algorithm. Let $x = x_0 \cdots x_{n-1} \in A$, with each $|x_i| = w$. First, initialize y_j to 0^w for each $j \in [0..N-1]$. Then, for each $i \in [0..n-1]$ and $k \in h_i$, replace y_k by $y_k \oplus x_i$. When done, set $h(x) = y_0 \,\|\, y_1 \,\|\, \cdots \,\|\, y_{N-1}$. In pseudocode we have:

```
**for** $j \leftarrow 0$ **to** $N-1$ **do** $y_j \leftarrow 0^w$
**for** $i \leftarrow 0$ **to** $n-1$ **do**
    $y_{h_i^1} \leftarrow y_{h_i^1} \oplus x_i$
    $y_{h_i^2} \leftarrow y_{h_i^2} \oplus x_i$
    $y_{h_i^3} \leftarrow y_{h_i^3} \oplus x_i$
**return** $y_0 \,\|\, y_1 \,\|\, \cdots \,\|\, y_{N-1}$
```

The computation of a $h(x)$ can be envisioned as follows. We have N buckets, each initially empty. The first word of x is thrown into the three buckets specified by h_0. The second word of x is thrown into the three buckets specified by h_1. And so on, with the last word of x being thrown into the three buckets specified by h_{n-1}. Our N buckets now contain a total of $3n$ words. Compute the xor of the words in each of the buckets (with the xor of no words being defined as 0^w). The hash of x, $h(x)$, is the concatenation of the final contents of the buckets.

3.2 Collision probability of $\mathcal{H}_B[w,n,N]$

The following theorem shows that $\mathcal{H}_B[w,n,N]$ is $\Theta(N^{-6})$-good. For large N and n, where $N \ll n$, the bound approaches $3348N^{-6}$.

Theorem 11. [Bucket hash bound] Assume $w \geq 1$, $N \geq 20$ and $n \leq \binom{N}{3}$. Then $\mathcal{H}_B[w,n,N]$ is $(\lambda_{n,N} \cdot \alpha(N))$-AU$_2$, where $\lambda_{n,N} = \dfrac{1}{1 - n/\binom{N}{3}}$ and $\alpha(n) =$

$$\frac{720(N-3)(N-4)(N-5)+1944(N-3)(N-4)^2+648(N-2)(N-3)^2+36N(N-1)(N-2)}{N^3(N-1)^3(N-2)^3}.$$

The proof (given in Appendix A) involves a tedious calculation on a Markov chain. For intuition, it is not hard to guess that $\max_{x_0 \neq x_1} \Pr[h(x_0) = h(x_1)]$ is achieved on strings x_0 and x_1 which differ in exactly four (appropriately selected) words. Bounding the collision probability for this case gives the formula of above. Most of the effort is showing that x_0 and x_1 differing by four words really *is* the case which which maximizes the collision probability.

By way of example, suppose we use $\mathcal{H}_B[w,n,N]$ to hash $n \in \{256, 1024, 4096\}$ words down to $N \in \{20, 40, \cdots, 200\}$ words. From Theorem 11, upper bounds on the probability that distinct but equal length strings collide are given by:

	20	40	60	80	100	120	140	160	180	200
256	$2^{-14.01}$	$2^{-20.25}$	$2^{-23.76}$	$2^{-26.24}$	$2^{-28.17}$	$2^{-29.75}$	$2^{-31.08}$	$2^{-32.23}$	$2^{-33.25}$	$2^{-34.16}$
1024	$2^{-11.08}$	$2^{-20.13}$	$2^{-23.73}$	$2^{-26.23}$	$2^{-28.16}$	$2^{-29.74}$	$2^{-31.08}$	$2^{-32.23}$	$2^{-33.25}$	$2^{-34.16}$
4096		$2^{-19.51}$	$2^{-23.59}$	$2^{-26.17}$	$2^{-28.14}$	$2^{-29.73}$	$2^{-31.07}$	$2^{-32.22}$	$2^{-33.24}$	$2^{-34.16}$

The first entry of the last row is missing because it does not satisfy the condition of Theorem 11: there are not enough distinct triples of 20 buckets to accommodate 4096 words.

4 The Efficiency of Universal Hash Methods

To make a practical MAC we want a fast-to-compute ϵ-AU$_2$ hash family $\mathcal{H} = \{h : \{0,1\}^{\leq a} \to \{0,1\}^b\}$ where, for example, $a \approx 2^{30}$, $b \approx 64$, and $\epsilon \approx 2^{-30}$. This section compares the efficiency of various universal hash families useful to construct such an $\mathcal{H}$. Efficiency comparisons are given in a very concrete way: we count the machine instructions per word of x to compute $h(x)$. We assume a contemporary 32-bit architecture. Though instruction counting is only a crude predictor of speed, an analysis like this is still the best implementation-independent way to get a feel for our methods' efficiency.

EFFICIENCY OF $\mathcal{H}_B$. From Section 3.2 and the above preamble it is apparent that we need more buckets than can be accommodated by a typical machine's register set. So there are two natural strategies to hash $x = x_0 \ldots x_{n-1}$:

- **Method 1** – Process words $x_0, \ldots, x_{n-1}$. We can read each x_i from memory and then, three times: (1) read the value y_j of some bucket j from memory; (2) compute $x_i \oplus y_j$; (3) write the result back to bucket j. Total instruction count is 10 instructions per word.

- **Method 2** – Fill buckets $y_0, \ldots, y_{N-1}$. We can xor together all words that should wind up in bucket 0; then xor all words that go in bucket 1; and so forth, for each of the N buckets. A total of $3n$ reads into $x_0, \ldots, x_{n-1}$ will be needed; plus a total of $3n - N$ xor operations; and (possibly) another N writes back to memory. Total instruction count is 6 instructions per word.

Achieving the stated instruction counts requires the use of self-modifying code ("sm code"); above, it is implicitly assumed that the representation of $h \in \mathcal{H}_B$ is the piece of executable code which computes it. In implementation, this can be tricky. If we must spend the time to load the bucket location (Method 1) or word location (Method 2) out of memory ("kiss code"), these loads comprise extra overhead. For Method 2, a refusal to use self-modifying code will further increase the instruction count because of overhead to control the looping: it is key-dependent how many words fall into a given bucket.

method	**implementation**	**table size (Bytes)**	**$\approx$ instrs/wd**
$\mathcal{H}_B[32, n, N]$	Method 1 (sm code)	code – no table	10
$\mathcal{H}_B[32, n, N]$	Method 2 (sm code)	code – no table	6
$\mathcal{H}_B[32, n, N]$	Method 1 (kiss code)	$3n$ or $12n$	13
$\mathcal{H}_B[32, n, N]$	Method 2 (kiss code)	$6n$ or $12n$	9+

For Method 2, n should be at most a few thousand to retain reasonable cache performance.

EFFICIENCY OF $\mathcal{H}_N$ AND $\mathcal{H}_M$. The software-efficient implementation of $\mathcal{H}_N$ relies on a table of pre-computed inner products. There is a tradeoff between the size of this table and the number of instructions. Consider $\mathcal{H}_N[64, 32]$ and assume we partition words as 8-bit bytes to look up inner products in a pre-computed table TN. The table size is then 4×2^8 words, or 4 KBytes. To hash the two-word string $x_0x_1x_2x_3\ x_4x_5x_6x_7$ to the one-word string $y_0y_1y_2y_3$ we must isolate the four bytes x_0, x_1, x_2, x_3; lookup in TN the entries these bytes index; then xor what we get out of the table. We need 3–5 instructions per table lookup, plus an extra couple of instructions to read $x_4x_5x_6x_7$ and xor it with what we have already. Other examples:

method	**cmprsn**	**coll prob**	**table size (Bytes)**	**$\approx$ instrs/wd**
$\mathcal{H}_2 = \mathcal{H}_N[2 \times 32, 32]$	2	2^{-32}	4 K	7–11
$\mathcal{H}_N[8 \times 32, 32]$	8	2^{-32}	28 K	7–11
$\mathcal{H}_2 \circ \mathcal{H}_2 \circ \mathcal{H}_2$	8	$2^{-30.4}$	12 K	13.5–19.5

Overall compression is poor unless the table size is quite large, leading one towards multiple applications of concatenation and composition (using Propositions 2 and 4) as in the third row of the table.

The instruction count and table size of $\mathcal{H}_M$ are worse than $\mathcal{H}_N$—for example, 12–20 instructions with an 8 KByte table for $\mathcal{H}_M[2 \times 32, 32]$.

EFFICIENCY OF $\mathcal{H}_K[n, 64]$. The collision probability of $\mathcal{H}_K[n, 32]$ is inadequate for the stated goal, and there is no apparent way to implement $\mathcal{H}_K[n, w]$ for

$w \in [33..64]$ any faster than implementing $\mathcal{H}_K[n,64]$. The software-efficient implementation of $\mathcal{H}_K[n,64]$ relies on a table TK of pre-computed quotients. Let us assume that we index into TK in 8-bit bytes. Computing $h \in \mathcal{H}_K[n,64]$ on $x_0 \cdots x_{n-1}$, where $|x_i| = 8$, could be done as in

```
crc ← $0^{64}$
for $i = 0$ to $n/8 - 1$ do
        index ← $x_i \oplus$ (crc & 0xFF);   crc ← crc ⟩⟩⟩ 8;   crc ← crc $\oplus$ TK[index]
return crc
```

which, coded in the natural manner on a 32-bit machine gives, gives 44–48 instrs/wd with a 2 KByte table.

5 Towards a Fully-Specified Scheme

AN EXAMPLE. This section provides an illustrative example of a concrete MAC based on the ideas presented so far. This is only a "toy" example; doing a good job at specifying a software-optimized bucket hash MAC will involve further design, experimental, and theoretical work.

To keep things simple, first suppose we want to MAC strings which are always 4096 bytes (1024 words). Let $F : \{0,1\}^\kappa \times \{0,1\}^{64} \to \{0,1\}^{64}$ be a (candidate) finite PRF. Here's how to MAC the cntth message $x \in \{0,1\}^{32\times 1024}$. Recall the $\mathcal{H}^i$ notation from Section 2.

1. Hash x with $h_1 \in \mathcal{H}_B[32,1024,144]$ to get a 144-word result, $y_1 \leftarrow h_1(x)$.
2. Hash y_1 with $h_2 \in (\mathcal{H}_N[64,32])^{72}$ to get a 72-word result, $y_2 \leftarrow h_2(y_1)$.
3. Hash y_2 with $h_3 \in (\mathcal{H}_N[64,32])^{36}$ to get a 36-word result, $y_3 \leftarrow h_3(y_2)$.
4. Hash y_3 with $h_4 \in (\mathcal{H}_N[64,32])^{18}$ to get an 18-word result, $y_4 \leftarrow h_4(y_3)$.
5. Hash y_4 with $h_5 \in \mathcal{H}_K[576,64]$ to get a 2-word result, $y_5 \leftarrow h_5(y_4)$.
6. Compute $p = F_a(\langle \text{cnt} \rangle_{64})$.
7. Return (cnt, $p \oplus y_5$).

Conceptually, the MAC key is $(a, h_1, h_2, h_3, h_4, h_5)$; in practice, it would be the seed used by a pseudorandom generator to generate such a tuple.

Let us estimate the speed of the above scheme, TOY-MAC. Assume the most pessimistic instruction counts in Section 4, and assume 200 instructions for the computation of F_a. With bucket hashing achieved by Method 2 (sm code), we get 6+11(144/1024)+11(72/1024)+11(36/1024)+48(18/1024)+200(2/1024) $\approx$ 10.3 instructions per word. The tables add up to 14 KBytes and the total collision probability (using Propositions 2,4,5,9; Theorems 10,11) is $\leq 2^{-31.32} + 3 \times 2^{-32} + 2^{-53} \approx 2^{-29.8}$. For comparison, recall that authenticating messages using an MD5-based technique requires some 40–50 instructions per word [21].

Notice that the "cryptographic" contribution (Step 6) of TOY-MAC takes just 200(2/1024) $\approx$ 0.4 instructions, which is less than 5% of the total work. In a Wegman-Carter MAC one is afforded the luxury of conservative (slow) cryptography even in an aggressive (fast) MAC, since one arranges that the time complexity for the MAC is dominated by the non-cryptographic work.

Schemes simpler than TOY-MAC still perform well. Replacing steps (1)–(5) by $y_4 \leftarrow h_5(h_1(x))$ gives $6 + (144/1024)48 + (2/1024)200 \approx 13.2$ instructions/word.

Alternatively, since MD5 is about the same speed as $\mathcal{H}_K$, a similar estimate would hold for a scheme like $\text{MAC}_{a,\text{cnt}}(x) = \langle \text{cnt}, \text{MD5}(a \,.\, \text{cnt} \,.\, |x| \,.\, h_1(x))\rangle$.

Our instruction counts are attractive even if we do not use self-modifying code: naive implementations would take fewer than 20 instrs/wd.

Short strings. What if the strings we are MACing haves lengths which are a constant substantially less than 4096 bytes? Using the layers of hash functions chosen for TOY-MAC makes no sense if $|x|$ is too short. In general, the hash method which will be fastest for a string depends strongly on its length. For a software-optimized MAC to be most useful it should be as fast as possible for each input length. We thus suggest designing a Wegman–Carter MAC using multiple AU$_2$-hash functions, choosing the best sequence of these to apply to x based on $|x|$. While this may seem complex, the user of a MAC typically cares about performance and security, not definitional simplicity.

While instruction counts for fast AU$_2$-hashing are best (per word) when the strings being hashed are long, even strings of a few words benefit from having their lengths reduced (e.g. by $\mathcal{H}_N$) before being cryptographically acted on.

Long strings. Modifying TOY-MAC to deal with long strings is easy: for example, if x is $1024k$ words then one can replace Steps 1–4 by $y_4 \leftarrow (h_4 \circ h_3 \circ h_2 \circ h_1)^k$.

Length variability. TOY-MAC is only correct when the strings to MAC have some fixed length. The reason is that $\mathcal{H}_B$ is ϵ-AU$_2$ only when restricted to strings of a fixed length. For example, $h(x) = h(x0^w)$ for any $h \in \mathcal{H}_B$. Fortunately, eliminating the fixed-length restriction is easy. One solution is to include $|x|$ in the scope of $F_a(\cdot)$, which ensures that $\text{WC}[\mathcal{H}, F]$ is secure across variable-length strings as long as $\mathcal{H}$ is ϵ-AXU$_2$ on equal-length ones. The bound in Theorem 7 is unchanged. A second approach is to use constructions like $h_f(\langle |x| \rangle \,.\, h_0(x))$. Neither approach requires $|x|$ be known before x is hashed.

6 Extensions and Directions

Generalizing $\mathcal{H}_B$ we call by "bucket hashing" any scheme in which the hash function h is a given by a list $(h_0, \ldots, h_{n-1})$ of "small" subsets of $[0..N-1]$ and the hash of $x = x_0 \cdots x_{n-1}$, where $|x_i| = w$, is:

```
for $j \leftarrow 0$ to $N-1$ do $y_j \leftarrow 0^w$
for $i \leftarrow 0$ to $n-1$ do
    for each $k \in h_i$ do
        $y_k \leftarrow y_k \oplus x_i$
return $y_0 \,\|\, y_1 \,\|\, \cdots \,\|\, y_{N-1}$
```

In the general case the distribution on h-values is arbitrary. So $\mathcal{H}_B[w, n, N]$ is the special case in which we use the uniform distribution on three-elements subsets of distinct triples in $[0..N-1]$.

One could imagine many alternate distributions, some of which will give rise to faster-to-compute hash functions or better bounds on the collision probability. As an example, suppose $\mathcal{H}$ is given by choosing $h_0, h_1, \ldots,$ in sequence, where h_i is a random triple of $[0..N-1]$ subject to the constraint that among

$\{h_0, \ldots, h_i\}$ there are no two *and no four* of the h_j's such that the multiset $\cup h_j$ has an even number of each point $0, 1, \ldots, N-1$. (Dropping the italicized words we would recover the definition of $\mathcal{H}_B[w,n,N]$.) This new family of hash functions, $\mathcal{H}'_B[w,n,N]$, may have substantially smaller collision probability than $\mathcal{H}_B[w,n,N]$, allowing one to choose a smaller value of N.

THE BUCKET HASH SCHEME OF A GRAPH. Hash family $\mathcal{H}_B$ would have been more efficient had each word gone into two buckets instead of three. One way to specify such a scheme is with a graph G whose N vertices comprise the N buckets and whose m edges $[0..m-1]$ indicate the pairs of buckets into which a word may fall. A random hash function from the family is given by a random permutation π on $[0..m-1]$. To hash a string $x_0 \ldots x_{n-1}$ using π, where $|x_i| = w$ and $n \leq m$, each word x_i is dropped into the two buckets at the endpoints of edge $\pi(i)$. As before, we xor the contents of each bucket and output their concatenation in some canonical order. We call the above scheme $\mathcal{H}_G[w,n,N]$.

Finding a "good" $\mathcal{H}_G[w,n,N]$ amounts to finding a graph G on a small number of vertices N, a large number of edges m, and such that for all $1 \leq k \leq n \leq m$, if k distinct edges are selected at random from G, then the probability that their union (with multiplicities) comprises a union of cycles is at most some small number ϵ

One interesting set of graphs in this regard are the (d,g)*-cages* (see [4]). A (d,g)-cage is a smallest d-regular graph whose shortest cycle has g edges. These graphs have been explicitly constructed for various values of (d,g). Though (d,g)-cages are rather large (for even g they have at least $(2(d-1)^{g/2}-2)/(d-2)$ nodes) some (d,g)-cages may give rise to useful hash families. The following is an example.

For $d-1$ a prime power, let $C[d,6]$ be the $(d,6)$-cage. This is the the point-line incidence graph of the projective plane of order $d-1$. I conjecture that $\mathcal{H}_{C[d,6]}[w, d^3-d^2+d-1, 2d^2-2d+2]$ is ϵ-AU$_2$ for $\epsilon = \binom{d^2-d+1}{3} / \binom{d(d^2-d+1)}{6}$. Assuming this, $\mathcal{H}_{C[10,6]}[w, 909, 182]$ achieves compression ≈ 5, collision probability $2^{-32.572}$, and Method 2 cost of 4 (sm code) or 6 (kiss code) instrs/wd.

OPEN QUESTIONS. The generalized notion of bucket hashing amounts to saying that hashing is achieved for each bit position $1 \ldots w$ by matrix multiplication with a sparse Boolean matrix H. Expressing the method in this generality raises questions like the following: for all distributions $\mathcal{D}[N,n]$ of binary $N \times n$ matrices H having k ones per column (e.g., $k = 2,3,4$) for which is $\max_{x \in \{0,1\}^n - \{0^n\}} \Pr_{H \in \mathcal{D}[N,n]} \left[Hx = 0^N\right]$ minimized? What if we also demand that each pair of rows have the same number of ones? (This is true in the matrix of $C[d,g]$, and having this property eliminates the Method 2 (kiss code) overhead mentioned in Section 4.) What if we demand density $\leq \rho$ but make no further restriction on H? Answers will lead to still faster bucket hash MACs.

Acknowledgments

The author thanks Mihir Bellare, Don Coppersmith, Hugo Krawczyk, David Zuckerman, and an anonymous reviewer for their comments and suggestions.

References

1. R. Arnold and D. Coppersmith, "An alternative to perfect hashing." IBM RC 10332 (1984).
2. M. Bellare, O. Goldreich and S. Goldwasser. "Incremental cryptography: The case of hashing and signing." *Advances in Cryptology - CRYPTO '94 Proceedings*, Springer-Verlag (1994).
3. M. Bellare, J. Kilian and P. Rogaway, "The security of cipher block chaining." *Advances in Cryptology - CRYPTO '94 Proceedings*, 341–358 (1994).
4. J. Bondy and U. Murty, *Graph theory with Applications.* North Holland (1976).
5. G. Brassard, "On computationally secure authentication tags requiring short secret shared keys." *Advances in Cryptology - CRYPTO '82 Proceedings*, 79–86 (1983).
6. J. Bierbrauer, T. Johansson, G. Kabatianskii and B. Smeets, "On families of hash functions via geometric codes and concatenation." *Advances in Cryptology -CRYPTO '93 Proceedings*, Springer-Verlag, 331–342 (1994).
7. L. Carter and M. Wegman, "Universal hash functions," *J. of Computer and System Sciences* 18, 143–154 (1979).
8. Y. Desmedt, "Unconditionally secure authentication schemes and practical and theoretical consequences." *Advances in Cryptology - CRYPTO '85* Proceedings, Springer-Verlag, 42–45 (1985).
9. P. Gemmell and M. Naor, "Codes for interactive authentication." *Advances in Cryptology - CRYPTO '93* Proceedings, Springer-Verlag, 355–367 (1994).
10. O. Goldreich, S. Goldwasser and S. Micali, "How to construct random functions." *Journal of the ACM,* Vol. 33, No. 4, 210–217 (1986).
11. S. Goldwasser, S. Micali and R. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal of Computing*, 17(2):281–308, April 1988.
12. H. Krawczyk, "LFSR-based hashing and authentication." *Advances in Cryptology - CRYPTO '94 Proceedings*, Springer-Verlag, 129–139 (1994).
13. M. Luby and C. Rackoff, "How to construct pseudorandom permutations from pseudorandom functions," *SIAM J. Comput*, Vol. 17, No. 2, April 1988.
14. X. Lai, R. Rueppel and J. Woollven, "A fast cryptographic checksum algorithm based on stream ciphers." *Advances in Cryptology, Proceedings of AUSCRYPT 92.* Springer-Verlag (1992).
15. P. Pearson, "Fast hashing of variable-length text strings." *Communications of the ACM,* 33 (6), 677–680 (1990).
16. R. Rivest, "The MD5 message digest algorithm." IETF RFC-1321 (1992).
17. P. Rogaway, "Bucket hashing and its application to fast message authentication." (Full version of this paper.) Available from the author or out of `http://www.cs.ucdavis.edu/~rogaway/`
18. G. Simmons, "A survey of information authentication." In *Contemporary Cryptography, The Science of Information Integrity,* G. Simmons, editor. IEEE Press, New York (1992).
19. D. Stinson, "Universal hashing and authentication codes." *Designs, Codes and Cryptography,* vol. 4, 369–380 (1994). Earlier version in *Advances in Cryptology - CRYPTO '91 Proceedings*, Springer-Verlag, 74–85 (1991).
20. R. Taylor, "An integrity check value algorithm for stream ciphers." *Advances in Cryptology - CRYPTO '93 Proceedings*, Springer-Verlag, 40–48 (1994).

21. J. TOUCH, "Performance analysis of MD 5." Manuscript, February 1995.
22. G. TSUDIK, "Message authentication with one-way hash functions." *Proceedings of Infocom 92,* IEEE Press (1992)
23. M. WEGMAN AND L. CARTER, "New hash functions and their use in authentication and set equality." *J. of Computer and System Sciences* 22, 265–279 (1981).
24. D. WHEELER, "A bulk data encryption algorithm." *Fast Software Encryption, Cambridge Security Workshop, 1993 Proceedings,* R. Anderson, ed., 127–134. Lecture Notes in Computer Science, vol. 809, Springer-Verlag (1994).
25. A. ZOBRIST, "A new hashing method with applications for game playing." University of Wisconsin, Dept. of Computer Science, TR #88 (April 1970).

A Proof of Theorem 11

Due to page limits we give only the briefest sketch. The complete proof is in [17].

We argue first that wlog we may assume $w = 1$. Then we observe that $h(x)$ is a product Hx over GF[2] for an $N \times n$ matrix H and so h is linear and the collision probability is $\epsilon^* = \max_{x \in \{0,1\}^n - \{0^n\}} \Pr_{h \in \mathcal{H}_B}[h(x) = 0^N]$. This depends only on the number of ones in x. Thus our problem has been reduced to deciding which string $x_t = 1^t 0^{n-t}$, $t \geq 1$, maximizes $\epsilon_t^* = \Pr_{h \in \mathcal{H}_B}[h(x_t) = 0^N]$. It is easy to see that it's not any odd-indexed x_o. Furthermore, $\epsilon_2^* = 0$ because of our exclusion of $h_i = h_j$ for $i \neq j$. So $\epsilon^* = \max\{\epsilon_4^*, \epsilon_6^*, \epsilon_8^*, \ldots\}$.

Upperbounding ϵ^* is facilitated by looking not at $\mathcal{H}_B$, where no $h_i = h_j$ for $i \neq j$, but at the related hash family $\mathcal{H}'_B$ which removes this constraint. Let ϵ'_t be defined by $\Pr_{h \in \mathcal{H}'_B}[h(x_t) = 0^N]$ and let $\epsilon' = \max_{t=4,6,8,\ldots}\{\epsilon'_t\}$.

To analyze $\mathcal{H}'_B$ we construct an $(N+1)$-state Markov chain $\overline{\mathcal{M}}$. States are numbered $0, 1, \ldots, N$. The e-th state models that e buckets have an even number of ones while $N - e$ buckets have an odd number of ones. So N is the start state. Let $\mathcal{N} = N(N-1)(N-2)$. For $3 \leq i \leq N-3$ chain $\mathcal{M}$ has transition probabilities $p_{i,j}$ given by $p_{i,i-3} = i(i-1)(i-2)/\mathcal{N}$, $p_{i,i-1} = 3i(i-1)(N-i)/\mathcal{N}$, $p_{i,i+1} = 3i(N-i)(N-i-1)/\mathcal{N}$ and $p_{i,i+3} = (N-i)(N-i-1)(N-i-2)/\mathcal{N}$, for all $p_{i,j}$ where both i and j are in $[1..N]$ and they differ by ± 1 or ± 3. For $0 \leq i \leq 2$ and $N-2 \leq i \leq N$ we have $p_{0,3} = p_{N,N-3} = 1$, $p_{1,2} = p_{N-1,N-2} = 3/N$, $p_{1,4} = p_{N-1,N-4} = N-3/N$, $p_{2,1} = p_{N-2,N-1} = 6/N(N-1)$, $p_{2,3} = p_{N-2,N-3} = 6(N-3)/N(N-1)$, and $p_{2,5} = p_{N-2,N-5} = (N-3)(N-4)/N(N-1)$. For all other transition probabilities $p_{i,j}$ we have $p_{i,j} = 0$. A simple calculation now shows that $\epsilon'_4 = \alpha(N)$. Using that $N \geq 20$ we get that $\alpha(N) \geq 1997/\mathcal{N}^2$ and so the following proves that $\epsilon' = \epsilon'_4$.

Lemma 12. Suppose $N \geq 20$. If $t \geq 6$ then $\epsilon'_t \leq 1788/\mathcal{N}^2$.

To prove this we bound selected probability masses $\pi'_i(t)$ for $i \in [1..N]$ in $\mathcal{M}'$ by considering the 7-state process $\overline{\mathcal{M}}$ which we get by collapsing a certain $N-6$ set of states of $\mathcal{M}'$. Process $\overline{\mathcal{M}}$ has states which we shall call N, $N-1$, $N-2$, $N-3$, $N-4$, $N-6$, and R. The first six represent the corresponding states in $\mathcal{M}'$ while state R represents the remaining states, combined. One can verify that the nonzero transition probabilities $\bar{p}_{i,j}$ of $\overline{\mathcal{M}}$ are bounded by $\bar{p}_{N,N-3} = 1$, $\bar{p}_{N-1,N-4} \leq 1$, $\bar{p}_{N-1,N-2} = 3/N$, $\bar{p}_{N-2,R} \leq 1$, $\bar{p}_{N-2,N-3} = 6(N-2)(N-3)/\mathcal{N}$,

$\bar{p}_{N-2,N-1} = 6(N-2)/\mathcal{N}$, $\bar{p}_{N-3,N} = 6/\mathcal{N}$, $\bar{p}_{N-3,N-6} \le 1$, $\bar{p}_{N-3,N-4} = 9(N-3)(N-4)/\mathcal{N}$, $\bar{p}_{N-3,N-2} = 18(N-3)/\mathcal{N}$, $\bar{p}_{N-4,R} \le 1$, $\bar{p}_{N-4,N-3} = 36(N-4)/\mathcal{N}$, $\bar{p}_{N-4,N-1} = 24/\mathcal{N}$, $\bar{p}_{N-6,N-3} = 120/\mathcal{N}$, $\bar{p}_{N-6,R} \le 1$, $\bar{p}_{R,R} \le 1$, $\bar{p}_{R,N-6} \le 15(N-5)(N-6)/\mathcal{N}$, $\bar{p}_{R,N-4} \le 60(N-5)/\mathcal{N}$ $\bar{p}_{R,N-2} = 60/\mathcal{N}$, We show that $\bar{\pi}_N(6) \le 1788/\mathcal{N}^2$ by induction on t. The basis and induction are obtained by calculations using the above bounds.

We conclude the theorem by showing that $\gamma_{n,N}$ adequately compensates for the error we have induced by examining $\mathcal{H}'_B$ instead of $\mathcal{H}_B$. This follows from a lemma asserting that if p is the probability that a random set of n distinct random triples of elements drawn from $[0..N-1]$ contains no repeated triple, then $\epsilon^* \le \epsilon'/(1-p)$.

B Definitions of Security

Security of a MAC. We define (deterministic, counter-based) message authentication schemes. In this case a MAC scheme $\mathcal{M}$ specifies a set $\mathsf{Messages} = \{0,1\}^{\le L}$ that can be authenticated; a finite set $\mathsf{Keys} \subseteq \{0,1\}^*$ of keys; a set $\mathsf{Tags} \subseteq \{0,1\}^*$; a number $\mathsf{MAX} = 2^r$ which is the number of messages that can be authenticated; and a pair of functions $(\mathsf{MAC}, \mathsf{MACV})$ where $\mathsf{MAC} : \mathsf{Messages} \times \mathsf{Keys} \times [0..\mathsf{MAX}-1] \rightarrow \mathsf{Tags}$, and $\mathsf{MACV} : \mathsf{Messages} \times \mathsf{Keys} \times \mathsf{Tags} \rightarrow \{0,1\}$. We write $\mathsf{MAC}_a^{\mathrm{cnt}}(x)$ for $\mathsf{MAC}(x, a, \mathrm{cnt})$, $\mathsf{MACV}_a(x,t)$ for $\mathsf{MACV}(x,a,t)$, and $\mathsf{MAC}_a(x)$ for $\mathsf{MAC}_a^0(x)$, We demand that for any $x \in \mathsf{Messages}$, $a \in \mathsf{Keys}$, and $\mathrm{cnt} \in [0..\mathsf{MAX}-1]$, $\mathsf{MACV}_a(x, \mathsf{MAC}_a^{\mathrm{cnt}}(x)) = 1$.

Let $\mathcal{M}$ be a message authentication scheme. A *MAC oracle* $\mathrm{MAC}_a(\cdot)$ for $\mathcal{M}$ behaves as follows: it answers its first query $x_0 \in \mathsf{Messages}$ with the string $\mathsf{MAC}_a^0(x_0)$; it answers its second query $x_1 \in \mathsf{Messages}$ with the string $\mathsf{MAC}_a^1(x_1)$; and so forth.

An *adversary* E for a message authentication scheme $\mathcal{M}$ is an algorithm equipped with a MAC oracle. Adversary E is said to *succeed* on a particular execution having MAC oracle $\mathrm{MAC}_a(\cdot)$ if E outputs a string (x^*, t^*) where $\mathsf{MACV}_a(x^*, t^*) = 1$ yet E made no earlier query of x^*.

Adversary E is said to be $[t, q, m, \epsilon]$*-break* $\mathcal{M}$, where $0 \le q < \mathsf{MAX}$ and $0 \le \epsilon \le 1$, if E makes at most q queries of its oracle, asks its oracle a total of m bits, runs in at most t time, and $\epsilon \ge \Pr\left[a \leftarrow \mathsf{Keys} : E^{\mathrm{MAC}_a(\cdot)} \text{ succeeds}\right]$.

Security of a finite PRF. A finite PRF is a map $F : \{0,1\}^\kappa \times \{0,1\}^l \rightarrow \{0,1\}^L$. We write $F_a(x)$ in place of $F(a,x)$. We let $R_{l,L}$ be the set of all functions mapping $\{0,1\}^l$ to $\{0,1\}^L$. Following [10], a *distinguisher* is an oracle algorithm D. We say that D $[t, q, \epsilon]$-breaks F if D runs in at most t steps, makes at most q oracle queries, and $\Pr_{a \leftarrow \{0,1\}^k}\left[D^{F_a(\cdot)} = 1\right] - \Pr_{g \leftarrow R_{l,L}}\left[D^g = 1\right] \ge \epsilon$. Complexity is measured in a standard RAM model of computation, with oracle queries counting as one step.