Embedded Security for Network-Attached Storage

Howard Gobioff[1], David Nagle[2], Garth Gibson[1]

June 1999
CMU-CS-99-154

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Contact: David Nagle (bassoon@cs.cmu.edu)
Office: 412-268-3898
Fax: 412-268-6353

Abstract

As storage interconnects evolve from single-host small-scale systems, such as traditional SCSI, to the multi-host Internet-based systems of Network-attached Secure Disks (NASD), protecting the integrity of data transfers between client and storage becomes essential. However, it is also computationally expensive and can impose significant performance penalties on storage systems. This paper explores several techniques that can protect the communications integrity of storage requests and data transfers, imposing very little performance penalty and significantly reducing the amount of required cryptography.

Central to this work is an alternative cryptographic approach, called "Hash and MAC", that reduces the cost of protecting the integrity of read traffic in storage devices that are unable to generate a message authentication code at full data transfers rates. Hash and MAC does this by precomputing security information, using and reusing the precomputed information on subsequent read requests. We also present a refined "Hash and MAC" approach that uses incremental hash functions to improve the performance of small read and write operations as well as non-block-aligned operations.

1. School of Computer Science, can be reached via email at *{hgobioff,garth}@cs.cmu.edu*
2. Department of Electrical and Computer Engineering, can be reached via email at *bassoon@cs.cmu.edu*

This research is sponsored by DARPA/ITO through DARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-0002. Additional support was provided by the member companies of the Parallel Data Consortium, including: Hewlett-Packard Laboratories, Hitachi, IBM, Intel, Quantum, Seagate Technology, Siemens, Storage Technology, Wind River Systems, 3Com Corporation, Compaq, Data General/Clariion, and LSI Logic.

ACM Computing Reviews Keywords: D.4.3 File systems management, C.3.0 Special-purpose and application-based systems, C.4 Design study, D.4.6 Cryptographic controls. D.4.4 Network communication

1 Introduction

Traditionally, disk drives and storage systems have been bound to single server machines that assume responsibility for most aspects of data integrity and security. However, the demand for greater scalability has forced storage to adopt a decentralized architecture where either: 1) multiple servers manage a shared set of disk drives [Soltis96]] or; 2) clients communicate directly with storage [Gibson97]. These systems achieve their scalability by eliminating the single-server bottleneck, possibly providing multiple access paths between storage and servers/clients, and by requiring storage to help manage the integrity of its data.

This storage-based integrity requirement is a fundamental change to storage systems and poses a number of system-level issues focused on how to efficiently implement integrity in storage. The most obvious issue is the computational cost of integrity, especially for the cost-constrained embedded environments of storage systems (i.e., disk drives or RAID controllers). Software solutions cannot support integrity, completely saturating a drive's CPU before reaching storage's 1-Gigabit/sec network transfer rate. Hardware solutions could provide 1-Gigabit/sec bandwidth, but are not available in low-cost commodity implementations [Eberle92].

To address these issues, this paper examines how to efficiently implement drive-embedded security. Our focus is on integrity because integrity is essential to the correct functioning in storage systems while applications are able to provide privacy without support from storage. We perform this study in the context of the Network-attached Secure Disk (NASD) architecture, using both a NASD prototype and simulation to understand the needs of security. Section 2 begins with a description of our Network-attached Secure Disk (NASD) architecture and prototype, measuring the performance impact traditional integrity mechanisms can have on storage's bandwidth and latency. In Section 3 we explore several approaches to minimizing the cost of integrity. By exploiting storage characteristics we develop two schemes that reduce integrity's computational cost, allowing a drive to provide only 33% of the peak cryptographic bandwidth while increasing latency by less than 10%. Finally, we study how these schemes interact with real distributed file system workloads, uncovering several potential problems and presenting solutions that allow storage to cost-effectively deliver drive-embedded security.

While this study focuses on embedded-security for the Network-attached Secure Disk architecture, many of the behaviors and results in this study are applicable to other storage systems including traditional single-server based systems running NFS or the WWW. For example, distributed file systems implemented on top of IPsec can significantly improve the scalability and performance of IPsec by providing using a similar MAC structure based on pre-computed hash function as described in Section 3. This is not only important, but essential to server machines that will quickly bottleneck under the load of data movement and cryptography. Of course, this does require some integration of security and the file system, but many of the most important performance improvements have come from these types of integration.

2 Network-attached Storage

In traditional distributed filesystems (Figure 1a), storage is protected behind a fileserver that screens all requests and transfers data between an internal I/O bus and a general-purpose network. Often, the fileserver becomes the system bottleneck because of the store and forward copying through the fileserver's memory. One solution is to avoid the server bottleneck by using clusters of trusted clients that issue unchecked commands to shared storage. However, few environments can tolerate such weak integrity and security guarantees. Even if only for accident prevention, file

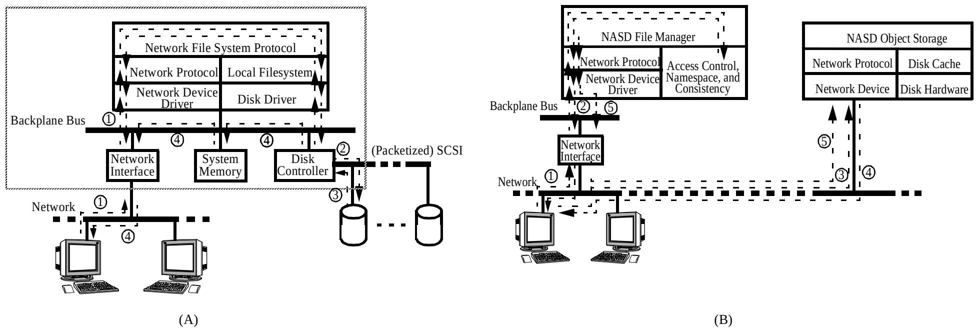


Figure 1: SAD vs. NASD. Server Attached Disks (SAD) interpose a server between the storage and the network that copies data from the peripheral network onto the client network. In the SAD case, a client wanting data from storage sends a message to the file server (1), which sends a message to storage (2), which accesses the data and sends it back to the file server (3), which finally sends the requested data back to the client (4). In the NASD case, prior to reading a file, the client requests access to a file from the file manager (1), which delivers access credentials to the authorized client (2). So equipped, the client makes repeated accesses to the different regions of the file (3,4) without contacting the file manager again unless the filemanager choose to revoke the clients rights (5).The server also protects communication with the storage. NASD removes the fileserver from the data path. NASD clients infrequently consult a filemanager and, in the common case, go directly to the storage device thus avoiding the store-and-forward through the fileserver.

protections and data/metadata boundaries should be checked by a small number of administrator-controlled file manager machines.

Another solution, Network Attached Secure Disks (NASD) (Figure 1b) avoids both the bottleneck and security problems by promoting simple storage devices (e.g. disk drives) to first-class network entities and enabling secure communication directly with clients [Gibson97]. Using NASD's high-level command interface, clients and drives communicate directly for common operations (e.g., reads and writes), while infrequent operations which depend on application specific semantics (e.g., namespace and access control manipulations) go to a file manager. Essential to the scalability of NASD is the division of authorization, which is performed asynchronously by a file manager, from enforcement, which is performed synchronously by the drive. Authorization, in the form of a time-limited access credential applicable to a given file (or set of files) is provided by the file manager upon an initial client request. Then, when the client makes a request to the drive, the drive enforces the access control decision previously specified by the file manager and encoded in the access credential. This allows the application specific semantics to reside in the file-manager. The filemanager encodes policy decisions in NASD-specific terms within the access credential that is passed to the drive via the client.

To experiment with the performance and scalability of NASD, we designed and implemented a prototype NASD storage interface, ported two popular distributed file systems (AFS and NFS) to use this interface, and implemented a striped version of NFS on top of this interface [Gibson97b]. The NASD interface offers variable length objects with size, time, security, clustering, cloning, and uninterpreted attributes. Access control is enforced by cryptographic access credentials authenticating the arguments of each request to a file manager/drive secret through the use of a digest. Using our implementations[1], we compared NASD/NFS performance against the traditional Server-Attached Disk (SAD) implementations of NFS. Our load-balanced large-read benchmark (512K chunks) showed that NASD is

1. The experimental testbed contained four NASD drives, each one a DEC Alpha 3000/400 (133MHz, 64 MB, Digital UNIX 3.2g-3) with a single 1.0 GB HP C2247 disk. We used four Alpha 3000/400's as clients. All were connected by a 155 Mb/s OC-3 ATM network (DEC Gigaswitch/ATM).

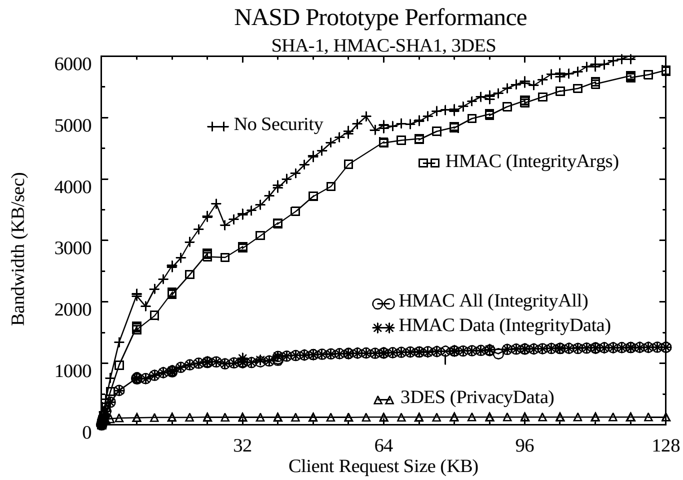


Figure 2: NASD Prototype Performance with and without Security. Protecting the integrity of arguments and return codes imposes a small performance penalty over a system with no security while protecting all data quickly saturates the CPU. Each point represents read throughput over 3^+ seconds of continuous read requests and a minimum of 100 requests by a single client reading data from an in-memory object This eliminates media access time from the measurements. The large irregularities in the graph are an artifact of the DCE/RPC communications layer. The data was taken from our NASD prototype, using DEC Alpha workstations (133 MHz 21064 processors) for NASD drives [Gibson98], 233 MHz Alpha workstations for clients and an OC-3 ATM DEC Gigaswitch for networking. HMAC-SHA1 implemented integrity and 3DES implemented privacy, with all security algorithms implemented in software.

able to scale linearly, up to the drive's aggregate transfer bandwidth, while SAD NFS and a parallelized version of NFS are limited by the data throughput of the server to just three drives [Gibson98].

2.1 Security Costs for Storage

Implementing efficient and cost-effect cryptography in a NASD is a challenging problem. Current processors cannot support software-based cryptography due to insufficient CPU cycles while hardware-based solutions are costly, especially for commodity storage. Figure 2 shows the performance of our NASD prototype across a range of software-implemented security options. With no security, raw performance peaks at ~6 MB/second. Activating integrity on a command's arguments (**IntegrityArgs**), which protects nonces and request arguments (e.g., object identifier, byte-range, operation-specific fields, return codes) using HMAC-SHA1 [Bellare96a], lowers performance by a fixed number of cycles, reducing 1-KB read performance by 30% and 128-KB read performance by 7%. By also protecting the data (**IntegrityAll**), the system provides improved protection but imposes a high overhead per byte. Figure 2 shows the maximum throughput is reduced by 46% for 1-KB reads and over 65% for 8-KB reads, where the CPU saturates due to cryptography. Finally, providing privacy for all of the data, **PrivacyData,** reduces performance by an even larger margin, to 126-KB/sec regardless whether or not any integrity protection is used.

These results demonstrate that software-based security on a low-cost microprocessor cannot meet the demands of storage. Other algorithms may be faster, but not by the factor of 240 necessary to meet modern 30 MB/sec disk drive media rates, nor the Gigabit/second storage network rates. Hardware-based cryptography would provide higher performance, but at significant cost, especially for 1-Gbit/sec security necessary for FibreChannel-based disk drives. Therefore, we turn to other approaches that enable high-performance and security from a low-cost storage device.

3 Optimizing Security Performance for Storage

3.1 Storage Characteristics

Storage has numerous characteristics that can be exploited to better optimize the performance of security. These include:

- repeated reading of the same data
- storage is non-volatile
- long access latencies (1 - 20 msecs)
- storage transfer rates (30 MBytes/sec) << interconnect transfer rates (100^{+}-Mbytes/sec)
- relatively slow CPUs << server class or workstation CPUs
- applications/operating systems allow many writes to happen lazily (e.g., Unix file cache sync)
- most data moved in large (bulk) transfers)
- most messages are small (i.e., commands)

Moreover, many applications read more data than they write [Baker91]. Good examples are executable files, data mining databases, mail files, directories, and news files with Berkeley NFS traces [Dahlin94] showing a read to write request ratio of 4.8 to 1. These infrequent changes enable resuse of both the raw data and any computation done over the data. For example, storing the network checksum with a set of data blocks allows subsequent reads to reuse the checksum, avoiding the repeated cost of on-the-fly checksum computation. Previous research has shown that web server support that stores the network checksum can improve throughput by more than 2X [Kaashoek96]. In the following section we discuss how to apply this technique and other storage characteristics to efficiently embed integrity in a disk drive.

3.2 Precomputing Security with Hash and MAC

Precomputation can be used to improve the performance of security. However, if data is shared using different keys, perhaps one per user, precomputation requires significantly more storage space to store the different MACs or does not benefit read requests from different users. However, it is possible to decouple MAC calculation into a keyed and an unkeyed component and *explicitly* delaying the binding of the key to the computation. Based on existing message authentication code and message digest algorithms, this approach, called *Hash and MAC*, does the following:

- When a drive object is written, the drive precomputes a sequence of *unkeyed message digests* over each of the object’s data blocks,
- For each read request, the drive generates a MAC of the concatenation of the unkeyed message digests corresponding to the requested data blocks.

Normal MAC algorithms (Figure 3a) involve the key throughout the entire computation of the message authentication code. In contrast, Hash and MAC removes the key from the per-byte calculation, only using the key in the final step of the calculation (Figure 3b). Because the key is not involved in the per-byte calculations, the results of the per-byte calculation, a set of message digests, can be stored and used for multiple read requests to the same disk block from different clients. Additionally, since no key needs to be identified before a message digest can begin, message digest processing may be simpler for high speed hardware than MAC processing which must delay until the proper key is identified.

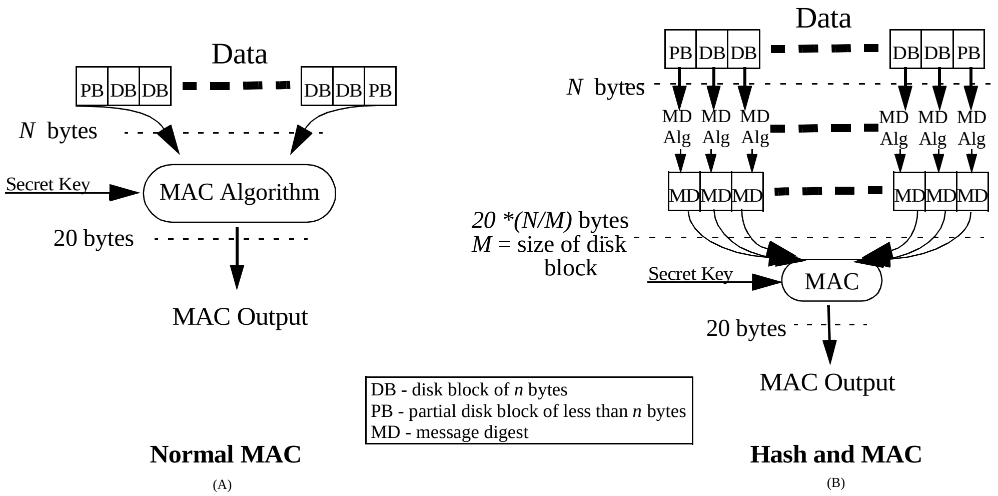


Figure 3: MAC Structures. The Hash and MAC approach reduces the amount of computation that involves the secret key. Each message consists of a sequence of full disk blocks which may be preceded and/or followed by a partial disk block. On the left, most MAC algorithms involve the key in the computation over all the bytes of data and process the data linearly. On the right, Hash and MAC does not involve the key until late in the computation. This enables parallelization and precomputation for increased performance. The labeled dotted lines indicate the amount of data that passes in and out of the message digest or MAC algorithms at that different stages of computation. In the Hash and MAC approach, a calculation over only 20 bytes per disk block involves the key while the rest of the computation can potentially be precomputed without knowledge of the key.

The Hash and MAC approach is very similar to encrypting or signing a message digest. However, it does not provide the non-repudiation that a public key system provides. In this sense, it is more like a normal MAC or encrypting a message digest with a symmetric key system. In contrast to encrypting a digest, a MAC has better defined properties to protect against modification and is not subject to U.S. export restrictions. See Appendix A for a discussion of the security of Hash and MAC.

3.2.1 Performance of Hash and MAC

NASD’s implementation of “Hash and MAC” uses SHA-1 to compute each disk block’s message digest and HMAC-SHA1 for the overall message authentication code. We refer to this specific instantiation of Hash and MAC as *HierMAC*. On a data read command, the pre-computed message digests are read from the drive and used as input to the HMAC-SHA1. If only a partial disk block is read, which only occurs in the first or last disk blocks of a request, a message digest of the partial disk block is computed on the fly.

With a normal MAC, the cryptographic costs are directly proportional to the number of bytes being transmitted. HierMAC reduces the cost to:

$$RequestHdr + (PrefixBytes + SuffixBytes) + NumOfFullDiskBlocks \times DigestSize$$

where *(PrefixBytes + SuffixBytes)* are the bytes in the partial data blocks. In our implementation, a disk block is 8KBytes while a message digest is 20 bytes. Thus, HierMAC performs a MAC operations on 20 bytes per full disk block (8KBytes) transferred, reducing in the asymptotic case the request time to 0.2% of over a normal MAC. This does not change the total number of bytes processed by the MAC algorithm. Instead, we are reordering the work in time and sharing work across multiple commands to reduce the on-the-fly cryptographic load.

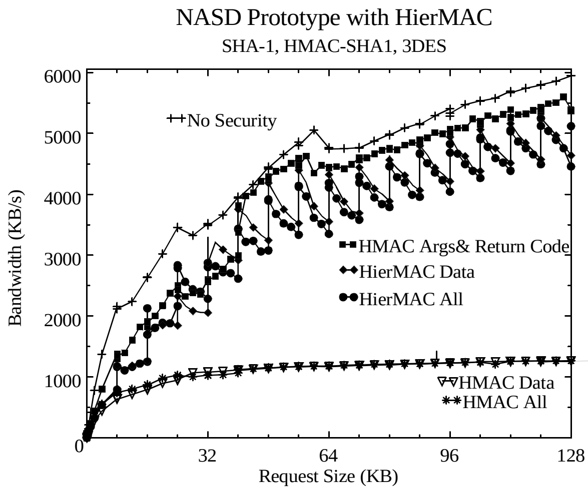


Figure 4: HierMAC Performance. Using HierMAC, based on SHA-1 and HMAC-SHA1, the drive delivers integrity-protected bandwidth close to the prototype's maximum bandwidth. Reuse of stored message digests has substantially improved the read bandwidth for large requests. The saw-tooth behavior occurs as the drive generates an on-the-fly message digest of the partial final data block.

The x-axis is the size of requests, which start at the beginning of an object, while the y-axis is the average read bandwidth seen at the client. Each point represents read throughput for a minimum of 3 seconds of continuous requests and a minimum of 100 requests. Each request is a single client reading data from an in-memory object at the drive beginning at byte 0. HierMAC stores an SHA-1 digest on each 8 KB disk block.

Figure 4 shows that reducing on-the-fly cryptography significantly increases read throughput, allowing performance to closely follow the no-security performance curve. Reuse of stored message digests reduces the total cryptographic costs so much that the cost of protecting the arguments in addition to the data is a noticeable performance difference. Between 8Kbyte block boundaries, performance declines because the drive spends more time processing the prefix and suffix bytes from the partial disk blocks. On 8KByte boundaries, the drive only uses only stored digest (*prefix + suffix* length returns to zero) and the cost of protecting integrity is minimize. This behavior results in the saw-tooth curve. For a uniform distribution of starting and ending bytes within a file, the average number of *prefix + suffix* bytes will be the size of one data block. Thus, the performance at the lowest points of the saw-tooth, 1 byte before hitting a disk block boundary, will represent the expected average performance for a randomly selected read request. Many filesystems attempt to make requests that are aligned on disk block or VM page boundaries which will result in significantly better performance.

3.2.2 Hash and MAC for Attributes

Filesystem attributes can also benefit from the pre-compute optimization. NASD supports both standard attributes[1] (e.g., modification time) and filesystem specific fields that are fairly large (256 bytes). Also, attributes change less frequently than data — our AFS workload shows a ratio of 22:1 of attribute retrieving operations versus attribute modifying operations while NFS shows a ratio of 6:1.

The relatively large size of NASD attributes coupled with their static nature makes them suitable for the same "Hash and MAC" optimization. This reduces the cost of protecting NASD attributes from 336 bytes to its 20 byte digest.

3.3 Efficient Support for Small Requests using an Incremental Hash

Small requests are very common in both distributed file systems [Baker91, Riedel96], databases and persistent object systems [Stamos84]. Therefore, small access and large transfers are important to consider.

1. We can apply this optimization to NASD attributes because, unlike tradition unix attributes, NASD does not maintain a last access time. If NASD maintained a last access time then the attribute would change on every operation and storing pre-computed digests with the attributes would not be advantageous because the digests would need to be updated on every request.

One optimization for small writes is to defer the updates of the stored digest and perhaps amortize the update cost across multiple small writes. However, deferring the update until the next read request can create unpredictable performance. As long as the stored digest is up to date, the client can make reasonable assumptions about expected request service time based on the request size, request alignment, and overall application state. If the digest may need to be recomputed, this introduces another variable that the client can not predict because it depends on the history of a given disk block.

A more promising solution is the incremental hashing paradigm developed by Bellare et al [Bellare94, Bellare97], which describes several hash functions for which the amount of work necessary to update a previously computed digest is proportional to the size of the change. For network attached storage, this enables small writes to be implemented much more efficiently.

3.3.1 Incremental Hashing

Incremental hashing divides a message into a sequence $x_1, x_2, \ldots, x_m$ of fixed sized blocks of size *b*, called incremental blocks. This is the base data unit for a hash function. Each incremental block is concatenated with its block number to generate an *augmented* block, $x_i' = i.x_i$. For each augmented block x_i', a compression function *h* is applied to x_i', generating the hash value $y_i = h(x_i')$. Combining $y_1, y_2, \ldots, y_m$ using a combining operator (∇) generates the final hash value.

More clearly we can express this as:

$$HASH(x_1, \ldots, \mathrm{x}_n) = \nabla_{i = 0, n} h(i.x_i)$$

To replace an incremental block x_i with a new incremental block $x_i^{\dagger}$ in a stored digest, we compute $h(i.x_i)$ and take an inverse of the stored hash and then combine in $h(i.x_i^{\dagger})$ with the stored hash. This is less work than recalculating the entire stored digest.

Another benefit of the incremental digest is the ability to compute the digest of a partial disk block without computing over all the data. Observe that

$$HASH(x_r, \ldots, \mathrm{x}_s) = \langle \nabla_{i = 0, n} h(i.x_i) \rangle \nabla^{-1} \langle \nabla_{i = 0 \ldots (r-1), (s+1) \ldots m} h(i.x_i) \rangle = \nabla_{i = r, s} h(i.x_i)$$

In English, if we read only part of the data covered by a stored digest, the hash for the partial data read is computed by taking the inverse of the stored hash and the hash of the complement of the portion being requested or computed it directly from the data being requested. Using this complementary property, we must compute over, at more, only half of the data to calculate a correct hash.

Bellare et al present two classes of incremental hash functions: MuHash and AdHash where the combining operators are modular multiplication and modular addition, respectively. For NASD, AdHash is more appealing than MuHash for two reasons: the size of digests and computational cost. Since the size of the modulos is equivalent to the size of the digest we must store with each disk block, MuHASH modulos of 512 to 1024 bits is too large, where-as

AdHASH requires ~200 bits[1]. This provides a factor of 2 to 4 space reduction for each disk block in NASD. Further, AdHash's addition operation is faster, enabling faster software and hardware implementations.

3.3.2 Integrating Incremental Digests in NASD

NASD can implement an incremental digest using AdHASH built on SHA-1 by applying the SHA-1 compression function to two sequential message blocks, $c_i = compress(x_i)$ and $c_{i+1} = compress(x_{i+1})$, producing two 160-bit digests which are combined by shifting the first one left 96 bytes and xoring the values together to produce a 256-bit hash:

$$h(x_i x_{i+1}) = (compress(x_i) \ll 96) \oplus compress(x_{i+1})$$

This provides all the collision resistance of the original SHA-1 compression function. An obvious alternative is to simply concatenate the results of the two calls to the compression function and generate a 320 bit output. However, a 320-bit digest requires the drive store 320-bits per disk block which is a larger overhead than storing only 256-bits. If subset sum attacks improve significantly, combining the output of two compression function calls can easily be adapted to produce a 320 bit subset sum problem.

The incremental hash functions described by Bellare et al concatenates the block number i onto data block x_i before hashing in order to prevent reordering of data blocks which can double the number of invocations of the compression function. If the incremental blocks are the same size as the basic block of the compression function, which allows fine grain changes to be done most efficiently, then we need to invoke the compression function twice, once for the data and once for the block id, which doubles the amount of hashing. The obvious solution is to increase the incremental block size to amortize the cost of appending the block number. However, increasing the incremental block size reduces our potential gains from using incremental cryptography because all up dates of a stored digest occur on the granularity of an incremental block.

We propose incorporating the block number into the IV of the compression function to eliminate the extra hash invocation. In an iterated hash function, of which SHA-1 is an example, the IV is used as an initial seed value for the first iteration and a chaining variable between iterations. By placing a value in the IV, the final result of the hash function is dependent on the value. If this were not true, the final output of an iterated hash function would not be dependent on the results of previous iterations.

Does changing the IV make it easier to find a collision? Although the SHA-1 design criteria are not public, we believe it unlikely that changing the IV will make collisions more likely. SHA-1's IV is a simple sequence of bytes with a very regular pattern thus provides little evidence of being a particularly special value. Additionally, SHA-1 is directly derived from MD4 [Rivest91] which was developed openly and has no publicly stated special design criteria for the IV. In the initial MD4 paper, Rivest suggests changing the IV, along with the other constants, and running two-MD4 functions in parallel to generate longer digest values which indicates some flexibility in the IV value. Preneel and Oorschot also modify the IV of arbitrary hash functions in their MDx-MAC construction [Preneel95]. Together, these facts make it unlikely that there is something special about the values employed in SHA-1's IV and it should be safe to effectively incorporate the block id into the message digest using the IV. Furthermore, this technique is applicable to any hash function that does not use special values in the IV.

---

1. Daniele Micciancio, one of the researchers working on incremental cryptography, recommends at least 256 bits for longer term security [Micciancio99].

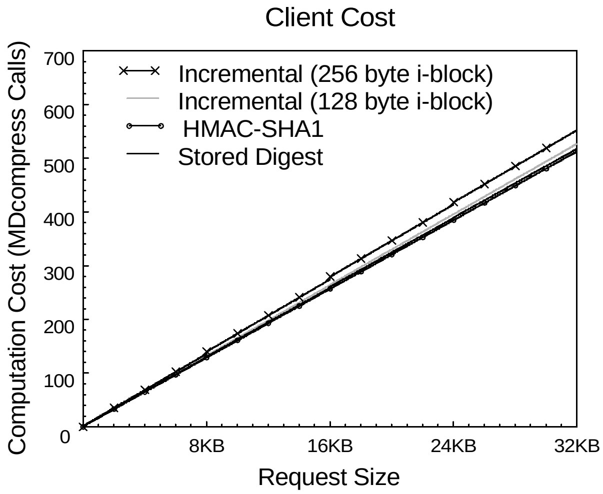


Figure 5: Client Overhead for Integrity. Using the precomputed digest optimizations requires clients to do a small amount of extra work for each disk block transferred. Based on a model of what computation the clients must perform, each line shows how many times a client must call the compression function, or equivalent work in the combine and uncombine operators, for a given amount of data being transferred. The x-axis is the size of the request and the y-axis is the number of invocations of the SHA-1 compression function, the core of SHA-1. The first two lines are approximations of the cost when using incremental cryptography on 256 byte and 128 byte incremental blocks respectively. HMAC-SHA1 is both the baseline for comparison and the minimal achievable amount of computation. Stored digest is the cost for the "Hash and MAC" approach with SHA-1 described in Section 3.2.

3.4 Comparison of Cryptographic Cost

To determine the impact of providing integrity at the drive, this section compares the cost of three approaches:

Basic MAC: The basic MAC scheme where all bytes are MAC'd using HMAC-SHA1.

Stored Digest: The "Hash and MAC" approach using precomputed SHA-1 digests stored with disk blocks as described in Section 3.2.

Incremental Stored Digest: The "Hash and MAC" approach using precomputed digests that are generated using AdHash built on SHA-1 and addition modulo 2^{256} with the block number placed in the IV. The precomputed digests are bound to a key using HMAC-SHA1.

The comparison is in terms of the number of invocations of the SHA-1 compression function which is the "common currency" of cost in the three approaches. The cost of padding out the messages to message digests or message authentication code block sizes is assumed to be zero. This is the cost of filling a buffer with up to 64 bytes of zeros and perhaps the message length which is a inexpensive operation relative to the compression function calls and will only occur once or twice per request. In contrast, the modular addition and subtraction used in the incremental stored digest approach will be used many times in a single request so must be modeled more accurately. The cost of modular addition and subtraction are modeled as 0.10 and 0.07 the cost of an invocation of the SHA-1 compression function. These values are the relative execution cycle counts, measured using DEC's ATOM profiling tools on a 233 MHz Alpha 21064, of simple C-language compiler-optimized implementations of 256 bit addition and subtraction using only 32 bit variables compared to our SHA-1 compression function implementation. The costs are likely higher than a hand optimized assembly implementation but provide a conservative estimate of the cost of the combine and uncombine operations. These costs as well as equations describing the costs of all three approaches were modeled in Mathematica to generate the data presented in Section 3.3.

3.4.1 Integrity Overhead Costs at the Client

Precomputation reduces the drive's work on reads, but forces the client to perform a small amount of extra work to MAC the message digests. Figure 5 shows that both stored digests solutions increase the client's overhead slightly for either a read or a write. The additional computational overhead is due to the combine operator (i.e., modular addition) that is applied to generate a single digest for an entire disk block from the results of the compression function on each incremental block. For example, the smallest possible block size (e.g. 128 bytes) requries the addition of 64 256-bit values per 8K disk block in additional overhead. while 256 byte incremental blocks requires half the overhead. We use 256 byte incremental blocks for the remainder of the evaluation.

3.4.2 Integrity Cost at the Drive for Reads

Figure 6a shows both the stored digests and incremental stored digests approaches perform significantly less cryptographic work for large block aligned reads because they can reuse stored digests. For smaller requests, the complementary property of incremental digests smooths out the saw-tooth curve. In a system where small requests were efficient, we expect this reduction to smooth the saw tooth behavior.

For non-aligned reads (Figure 6b), precomputed digests do not provide any benefit until the drive reads almost 2 full disk blocks. Here, the drive must always compute a digest on all the data in the first partial disk-block because the drive only stores a digest of the entire disk-block. In contrast, the incremental stored digest has a much smaller penalty because the complementary property is largely independent of the offsets. However, both Figure 6a and Figure 6b show a small saw-tooth behavior for the incremental stored digests because the drive may still compute over up to half a disk block before the complementary property is helpful. Just as the stored digest approach must on-the-fly calculate a hash of all bytes in a partially read disk block, the incremental stored approach must on-the-fly calculate a hash of all bytes in a partially read incremental block which creates the fine grained saw-tooth which overlays the coarser saw-tooth from the partial disk blocks. For smaller incremental block sizes, the tooth size will shrink while larger incremental block sizes will increase the size of the teeth. However, reducing the incremental block size increases the overhead.

3.4.3 Integrity Cost for Writes at the Drive

For writes, incremental digests and stored digests increase work by up to one disk block for small or misaligned operations. The drive must verify the received MAC and then update the stored message digest in both the stored and incremental stored approaches. For the stored digest approach, the drive must generate an entire new stored digest for a disk block even if only a single byte is written and this is a substantial penalty for small writes. If a write starts on a disk block boundary then computing of the new stored digest can simply continue from the calculation necessary to verify the digest on the data received from the client because recomputing the hash of the common prefix would be redundant. In this case, the cost is a function of the number of disk blocks touched by the write operation which creates the step-function effect shown in Figure 7a.

If a write begins at some offset into the disk block, shown in Figure 7b, the stored digest approach pays a larger penalty than the incremental stored approach. With the basic stored digest scheme, the drive can no longer continue the calculation used to verify the data received from the client because the received data is no longer a prefix of the disk block. Instead, the drive must first verify the received MAC and then start from scratch to generate the stored digest for the updated disk block which makes small, miss-aligned writes extremely expensive. The incremental approach is largely independent of offset and does not pay these penalties on small writes.

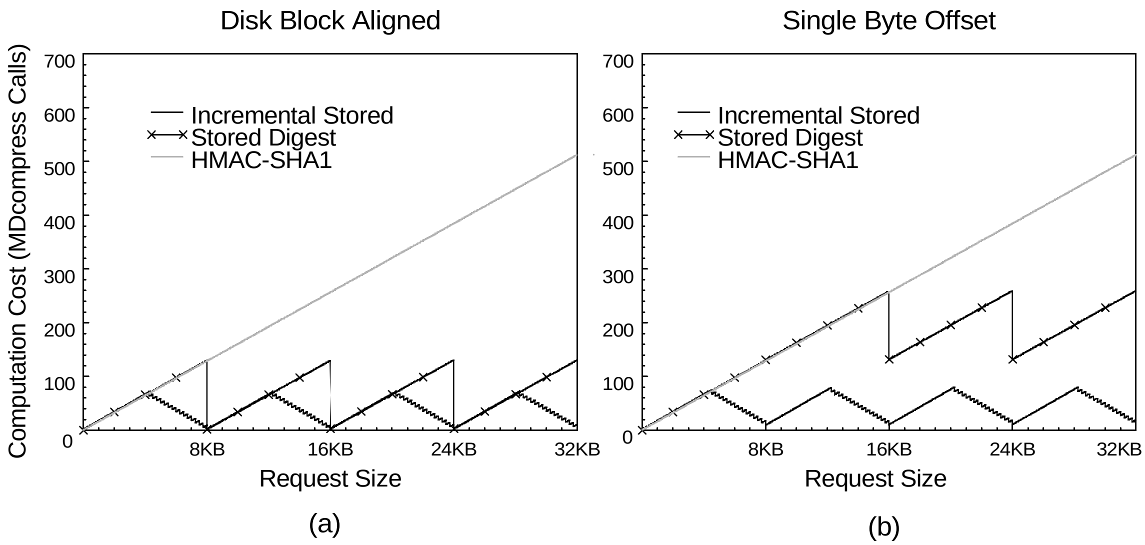


Figure 6: Drive Cryptographic Cost for Integrity on Reads. Both incremental stored and simple stored approaches significantly reduce amount of cryptographic work the drive must perform on a large read request compared to HMAC-SHA1. For misaligned reads, the complementary property of incremental digests allows the digests to be calculated more easily than normal digests. Based on a model of what computation the clients must perform, each line shows how many times a client must call the compression function, or equivalent work in the combine and uncombine operators, for a given amount of data being read from a given offsets. The x-axis is the size of the request and the y-axis is the number of invocations of the SHA-1 compression function, the core of SHA-1. Incremental stored is the incremental scheme described in Section 3.3.2 using 256 byte incremental blocks. Stored digest is the cost for the "Hash and MAC" approach with SHA-1 as described in Section 3.2. HMAC-SHA1 is the most standard way of providing communication integrity and is used as a basis for comparison.

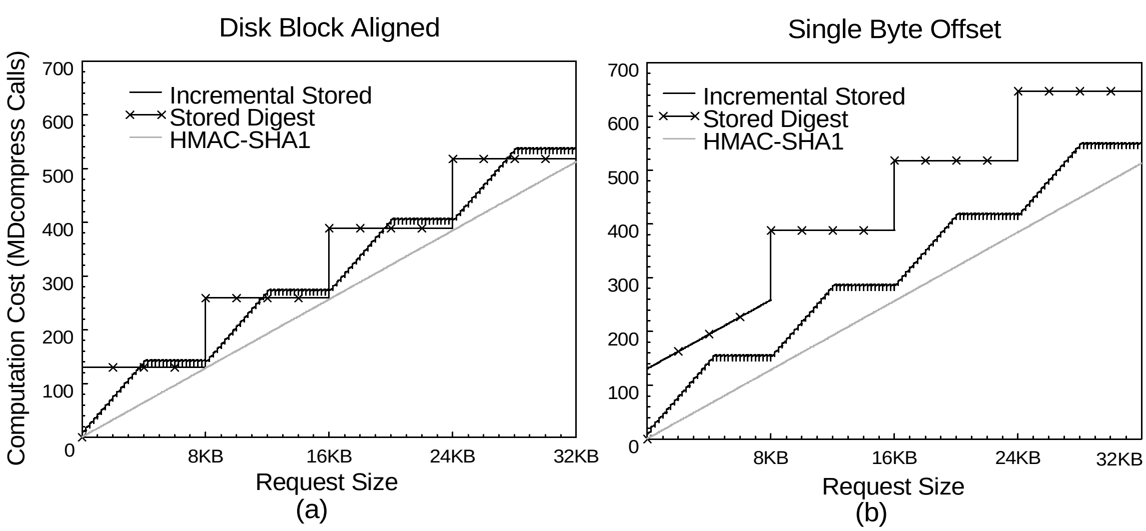


Figure 7: Drive Cryptographic Cost for Integrity on Writes. For write operations, both stored digest approaches pay a penalty for updating partially modified disk blocks. For misaligned operations, this penalty is reduced when incremental digests are used. Based on a model of what computation the clients must perform, each line shows how many times a client must call the compression function, or equivalent work in the combine and uncombine operators, for a given amount of data being written to a given offsets.The x-axis is the size of the request and the y-axis is the number of invocations of the SHA-1 compression function, the core of SHA-1. Incremental stored is the incremental scheme using 256 byte incremental blocks. Stored digest is the cost for the "Hash and MAC" approach with SHA-1.

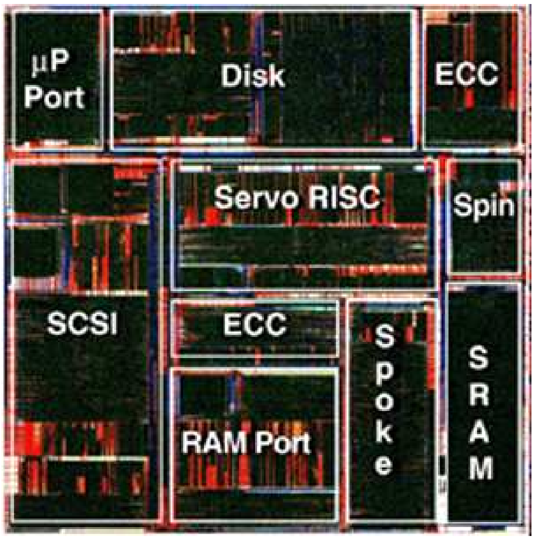


Figure 8: Quantum Trident ASIC. Modern drive ASICs integrate a large amount of functionality onto a single chip. The SCSI controller, servo controller, sequencer, motor controller, error correcting code, and a small amount of SRAM provide the core device functionality while a CPU and DRAM are on other chips. The primary ASIC in the Trident consumes approximately 110 thousand gates and 22 KByte of SRAM in a 74 sq. mm package using 0.68 micron chip technology.

4 Integrating Security Hardware into Network-attached Storage

4.1 Overview

Cryptography capable of sustaining network data rates is the ideal solution for any storage workload that requires security. However, cost considerations can make this difficult to achieve. Optimizations, such as the HierMAC or HierMAC with incremental digests reduce the amount of cryptography required for read traffic, but do not significantly improve small transfers or write-traffic bandwidth — both essential to achieving acceptable storage performance. Fortunately, acceptable storage performance at sub-network cryptographic speeds is possible because: 1) media data rates are significantly lower than high-speed network data rates; 2) storage workloads have periods of idleness. These characteristics provide a range of performance, between network and media data rates, that enables an unique set of trade-offs involving cost, throughput, and latency.

This section explores how this performance range can be exploited to achieve good system performance without implementing full network-speed cryptography. To ground the discussion, we begin with an overview of drive electronics and examine the performance of current software and hardware solutions. Next, we show how integrating security into a NASD impacts the system's latency and discuss the performance issues involved in providing integrity, privacy, or both. The analysis is quantified using real file system traces reveals that a drive, using HierMAC and providing only 33% of a network's full-duplex bandwidth, can successfully services file system requests with less than a 10% increase in latency (over a system with no security).

4.1.1 Integrating Hardware-based Security into Network Attached Secure Disk

The electronics on a modern SCSI disk drive are very similar to a modern computer, and include a microprocessor (~60 MIPS), a SCSI interface, and several megabytes of RAM. In addition, there are numerous very small functional units that manage the drive, including a motor controller, R/W channel, preamp & write driver, error correcting code engine, sequencer, buffer controller, servo controller — most of which are migrating into a single ASIC solution. A network-attached disk requires the same core function as a SCSI drive, replacing the physical SCSI interface with a high-performance network (e.g., Gigabit Ethernet, FibreChannel) while increasing the microprocessor's performance to support NASD's in-drive file system. Adding hardware-based security requires four new functional blocks: key memory, encryption/decryption, message authentication code (which uses SHA-1 in the prototype), and key management logic. Software-based security would require fewer functional blocks, with the encryption/decryption and key management blocks handled by the microprocessor and keys stored on the disk media. However software-based security demands a significantly more powerful processor. Further, because the microprocessor must touch all bytes that are either sent or received, the ASIC's internal datapath, which is currently optimized for minimal data movement through the microprocessor, would also require a fundamental change.

4.1.2 Current Software and Hardware Cryptography

Most cryptographic algorithms are not designed with efficient software implementation as a primary design criteria. For example, current workhorse encryption algorithms such as Triple-DES[1] (3DES), requires 108 clock cycles per byte on a Pentium processor [Schneier97], yielding about ~9MBytes/second on a Gigahertz Pentium Pro. Likely successors to Triple-DES, the Advanced Encryption Standard (AES) candidates [NIST98], all improve on the performance of Triple-DES, but still require 20-69 clock cycles per byte for 8 KB requests with an average penalty of an additional 3 cycles per byte a smaller, 1 KB request [Schneier99].

Hash functions have significantly better software performance than encryption. For example, SHA-1 on a 200 MHz Pentium requires 13 clock cycles per byte (15 MB/second) while RIPE-MD160 hashes at 16 clock cycles per byte (12.5MB/second) [Preneel98]. While better than the fastest AES algorithms, they will still consume most of a 200 MHz Pentium's cycles supporting the media rates of current disk drives. These numbers show that a NASD-class processor, ~200 MIPS (e.g. 200 MHz StrongARM), will be unable to support software-based cryptography, thus requiring hardware-based cryptography.

There are a wide range of hardware cryptographic accelerators. Eberlee at Digital's System Research Center demonstrated an experimental DES chip in 1992 that delivered 1 Gb/s performance [Eberle92]. Currently, you can purchase chips such as the Hi/Fn 7751 [HiFn99] or VLSI's VMS115 [VLSI99] running at 80 MHz which deliver approximately 100 Mb/s and 200 Mb/s performance for both SHA-1 and Triple-DES. These chips, primarily designed to enable IPsec-based virtual private networks in 100Mb/second routers, may not be priced aggressively for commodity devices. Pijinburg Custom Chips' next generation ASIC (500k gates, 0.18 micron) will implement SHA-1, Triple-DES, Safer SK64, and RIPEMD-160 [vanPelt99] and is expected to deliver up to 500 Mb/s performance from each functional unit. Cognitive Designs next generation ASIC, the CDI 3000, will perform Triple-DES at 172 Mb/s and concurrent SHA-1 at 204 Mb/s, priced at approximately $20 in volume [Finley99]. While these cost and performance numbers are difficult to map directly into a NASD, they do provide an intuition of the performance and cost of readily available hardware support.

4.2 Security and the Drive Datapath

Integrating cryptographic hardware into a storage device reduces latency and increase throughput over software-only solutions. Minimizing latency is very important because additional latency increases request service times, which clients are sensitive to on small requests, and increases a drive's internal memory requirements (i.e., larger queues). Similarly, guaranteeing (at least) media-rate bandwidth is important because without sufficient cryptographic throughput, the drive cannot deliver its raw bandwidth to clients.

1. Triple-DES was recently proposed as an updated U.S. government Data Encryption Standard (FIPS 46-3), replacing single DES, so we can expect it to be very relevant for many years to come [NIST99].

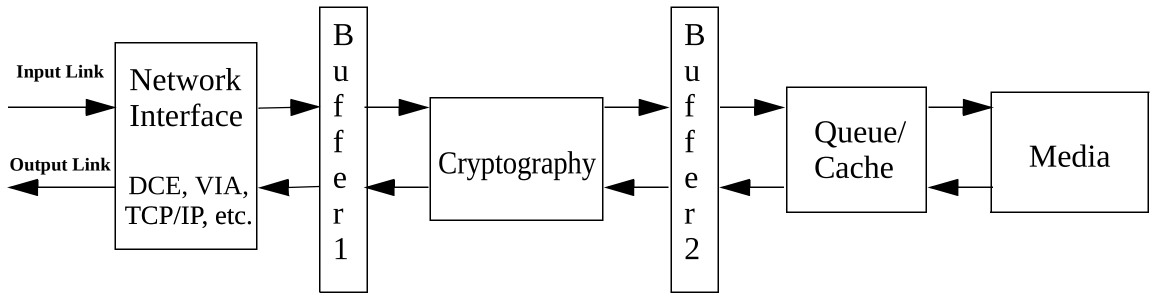


Figure 9: Model of a NASD's internal functional pipeline. When security is introduced into a disk drive, the drive may need to buffer requests both before or after the cryptography in order to maintain correctness or perform speed-matching. The Queue/Cache holds requests queued up at the media and the drive's data cache. The buffers and the media queue/cache may allocated from a single memory pool and illustrate a logical distinction rather than a physical one. If the network runs faster than the security but security is faster than the media, buffer1 will fill on a writes and buffer 2 on a cache read cache hit but both buffers will empty faster than media can drain the queue. If the security is slower than the media rates, buffer1 will fill on every write and buffer2 on every read.

At a functional level, integrating security into a drive architecture adds another stage to request processing, increasing latency and potentially throttling system throughput (Figure 9). Without security, requests arrive on the drive's network interface then they are processed by various levels of communication protocols. Next, they are placed on a work queue and then either serviced from the cache or scheduled for media access. For cached (or buffered) accesses, cryptography slower than the network will force commands or data to queue up between the network interface and crypto unit (buffer1) on incoming datapath and between the crypto unit and the drive electronics (buffer2) on the outgoing datapath. However, for cache misses or large writes, the drive's data rate is ultimately determined by the media.

4.3 Latency

Cryptographic operations impose several ordering dependencies that impact latency. In both directions, the first step is to determine which key should process a request. Because decryption cannot begin before the appropriate key is supplied, key management directly impacts the latency of decryption and/or integrity (using a standard MAC algorithm). Worse, integrity with privacy requires that the MAC wait until decryption is complete. However, the Hash and MAC parallelizes the key access with hash calculations, hiding the key management latency for most requests. Unfortunately, no such optimization is possible for privacy, which must always wait for the proper key.

Cryptographic primitives are the next component of latency. Encryption algorithms normally process 64-bit blocks, for 3DES, or 128-bit blocks for more modern ciphers. These small blocks allow encryption to form a fine-grained pipeline, producing results every 64 or 128 bits. An OceanLogic DES core processing one 64-bit block every 16 clock cycles [OceanLogic99], would implement 3-DES with a 48 cycle latency (*buffer1* and *buffer2* in Figure 9).

Integrity algorithms employ a much larger chunk size, creating a courser-grained pipeline and significantly increasing the latency of requests. For example, our prototype's MAC uses a 64 KB chunk size. With completely serialized operations, a 64KB chunk must be MAC'ed before transmission, then transmitted to the receiver, and finally verified by the receiver by performing another MAC. The process can be parallelized, but the receiver cannot verify the data until the corresponding MAC is received. With drives likely to send data faster than they perform a MAC operation, the time to compute a MAC will determine the minimum latency of an operation.

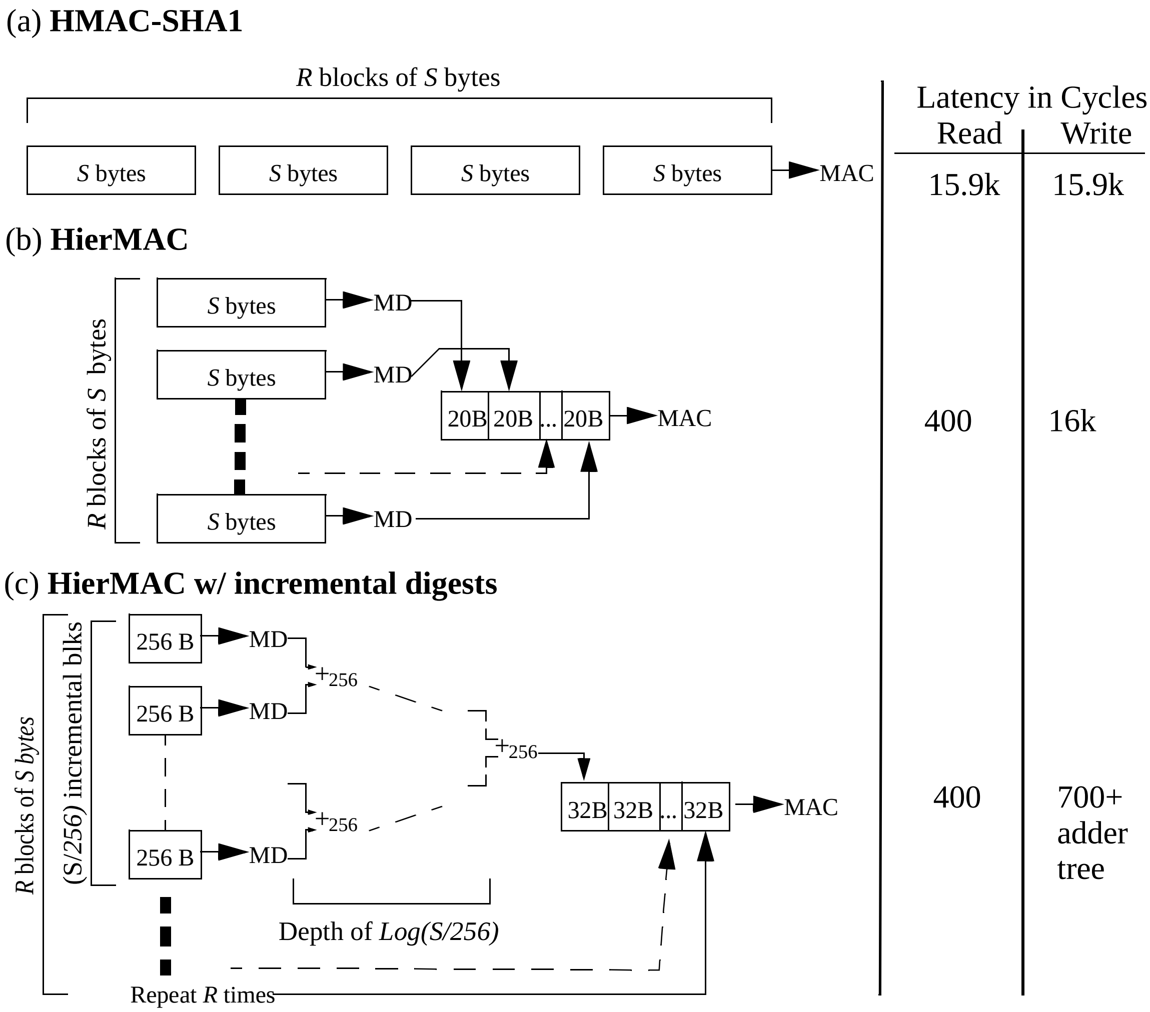


Figure 10: **Comparison of latency for different MAC approaches.** HierMAC has uses precomputation and it has lower latency than HMAC-SHA1 on a read request. On a write request, incremental stored digests also reduce latency because it introduces more parallelism. This figure illustrates the critical path length, i.e. latency, for the three MAC approaches. All three approaches are parameterized by S, the size of the disk block, and R, the maximum number of disk blocks sent before a MAC is inserted. HMAC-SHA1 simply computes over R*S bytes then it produces a result. HierMAC can use precomputed digests on the read and it can compute the digests in parallel on a write (which is the same as HMAC-SHA1 when R=1). HierMAC with incremental digests has more parallelism which benefits small requests and writes as well having the benefits of stored digests.

On the right side of the figure, we list the latency to process a read and write of disk block, ignoring header and key costs. we assume S = 8192 bytes an R=1, which makes HierMAC and HMAC-SHA1 comparable on the write path. For per message digest latency, we estimate 123 cycles, which is the amount of time required per message digest block in an FPGA implementation of the SHA-1 core by built by our research group [Schlosser98].

Integrity latency varies by a factor of 20 or more depending on type of MAC and the size/type of request. Both HierMAC and HierMAC w/incremental digests, improve latency over HMAC-SHA1 by enabling early data processing and both use precomputed digests, reducing latency to a few iterations of the message digest calculation. On writes, both HMAC-SHA1 and HierMAC have longer latencies than HierMAC w/incremental digests. HMAC-SHA1 latency is a function of chunk size while HierMAC depends on digest block size. HierMAC with incremental digests optimization reduces latency by enabling parallel computation over 256 B blocks, followed by the modular arithmetic (combining operators), and a final MAC. This parallelism does, however, require more hardware to process 256 B blocks in parallel.

For small requests, HMAC-SHA1's key generation dependency can create a long critical path. HierMAC avoids this critical path, allowing data computation to proceed without the key, but at the cost of an additional step, one extra iteration of the message digest calculation,

Finally, in addition to the cryptographic operations, the drive must also verify the nonce on a request and check that the an access credentials appropriate for the request. Checking the nonce requires searching for the nonce, probably in a hash table, to confirm that it has not already been received. For capabilities, a simple form of access credentials, the check requires only a few cycles to perform some simple comparisons. These checks can be performed in parallel with the cryptographic processing as long as the request is not irreversibly committed until all checks are completed.

4.4 Throughput

Storage throughput varies widely. Interconnects, such as Fibrechannel provide 2-Gb/sec full duplex while media transfer rates are currently only 28 MB/sec peak (increasing at 40% per year) [Grochowski96]. Clearly, cryptographic throughput at networking data rates provides optimal performance, especially for requests that hit in the on-drive RAM cache. However, limited amounts of drive cache significantly reduces the hit rate, except for sequential accesses that benefit from read-a-head. If cryptography performance exactly matches media rates, the maximum throughput will be media data rates. Cryptographic throughput exceeding media rates reduces cache hit latency and provides greater peak performance when cache hits do occur even though the sustained rate may be substantially lower. Additionally, exceeding the media rate allows requests to queue after cryptography, providing the drive with an opportunity to reorder requests and maximize its use of the media (although it can not exceed media data rates). The exact amount by which cryptographic data rates should exceed media rates will depend on the costs of increasing the data rates, the emphasis on peak bandwidth, and the probability of a cache hit. If a drive were to have a much larger data cache, optimizing for cache hits would be more compelling than the case for a drive with a few megabytes of cache.

We have designed the security of NASD to maximize the parallelism available to the drive in order to improve its throughput. Encryption is highly parallelized because it uses counter-mode rather than more standard modes which have dependencies between encryption blocks. The message authentication code can also be parallelized at the granularity of disk blocks when using HierMAC and at the granularity of incremental blocks when using incremental hashing. The same features that make the computations parallelizable also enable the system to tolerate out-of-order reception while still performing the security processing in stream-like manner.

There is nothing fundamentally preventing a drive from performing its cryptographic operations at full line rate. However, engineering a drive's cryptographic support to meet the peak data rates of the system implies that the drive is over-engineered for most of its workload.

5 Simulation Study of the Impact of Underprovisioned Digest Throughput on Client Latency when Protecting Integrity

The previous section argued that drives need not provide full network bandwidth cryptography. To quantify this hypothesis, we use file system traces to measure the impact of reduced message digest cryptography throughput on the latency of filesystem operations. Because drive-based integrity is necessary for correctness, while applications

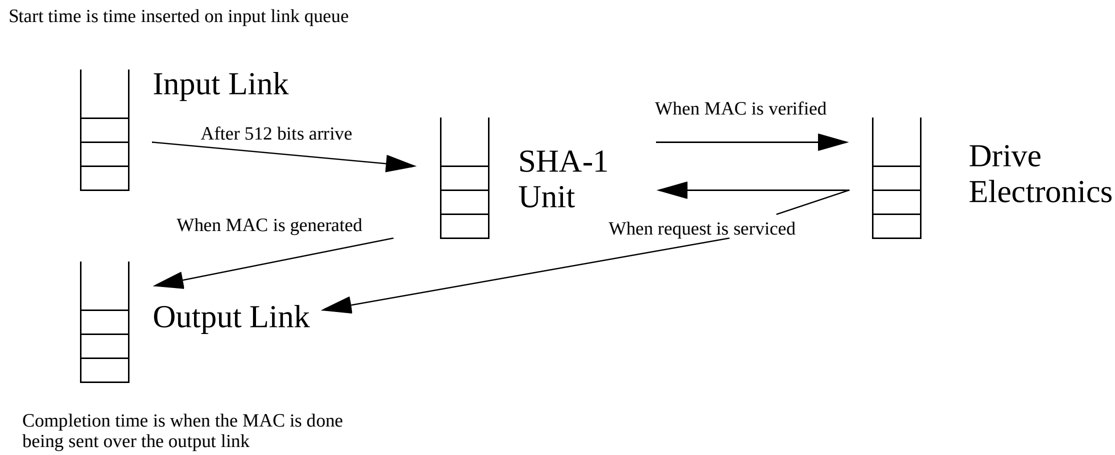


Figure 11: Simulation Queueing Model. These are the four resources modeled in the simulation and the transitions of requests between the queues.When a client sends a request to the drive, the request is first placed on the drive's input link queue. After 512 bits of data have been transferred over the link, the drive has enough data to begin SHA-1 and the request is placed on the SHA-1 queue. When all the required SHA-1 work is complete, the request's MAC has been verified and the request is queue on the drive electronics. After the request is serviced, the result is queue on both the SHA-1 unit and output link to concurrent send and generate the reply MAC. If the data is completely sent to the client before the reply MAC is generated, the MAC will be enqueued separately on the output link when it is complete.

can provide privacy, we focus on full integrity by exploring the minimal amount of cryptography necessary to deliver good performance to the client. The results show that a message digest unit that provides cryptographic bandwidth at only 30% of a full-duplex network link (e.g., Gigabit Ethernet) increases average latency by less than 10% over a system with no security.

5.1 Simulation Environment

Our trace driven simulator uses two sets of traces from: 1) University of California, Berkeley Auspex NFS fileserver [Dahlin94] and; 2) Carnegie Mellon University Parallel Data Lab AFS server traces collected in early 1999 [Gobioff99]. Both the AFS and NFS workloads are an approximation of the workloads offered to a NASD drive.

The simulator maps each NFS or AFS request to one or more NASD requests. Each NASD operation has a fixed NASD header as well as arguments and a result structure), which the simulator models in addition to the networking cost (32 byte header approximating a small UDP/RPC header). Message digest costs are modeled using HMAC-SHA1 to provide integrity.

The simulator uses queuing model of three classes of drive resources: network (Gigabit input and output links), message digest unit (SHA-1), and the drive electronics as shown in Figure 11. Drive requests have a fixed service time and media time is ignored, making the base service time 0.12 milliseconds (the time a Seagate Ultra Wide ST34371W drive takes to process a prefetch hit minus the time spent on the SCSI bus, i.e. the request "think time" [Riedel98]). This limits the drive to a maximum of 8,333 requests per second. By eliminating seek time and internal data transfer times, we bias heavily against reducing message digest capacity because the delay due to slower cryptography is more significant when the slowest portion of the drive is ignored. In some sense, we are modeling a solid-state disk while, for the foreseeable future, most NASD will use magnetic media as the backing store.

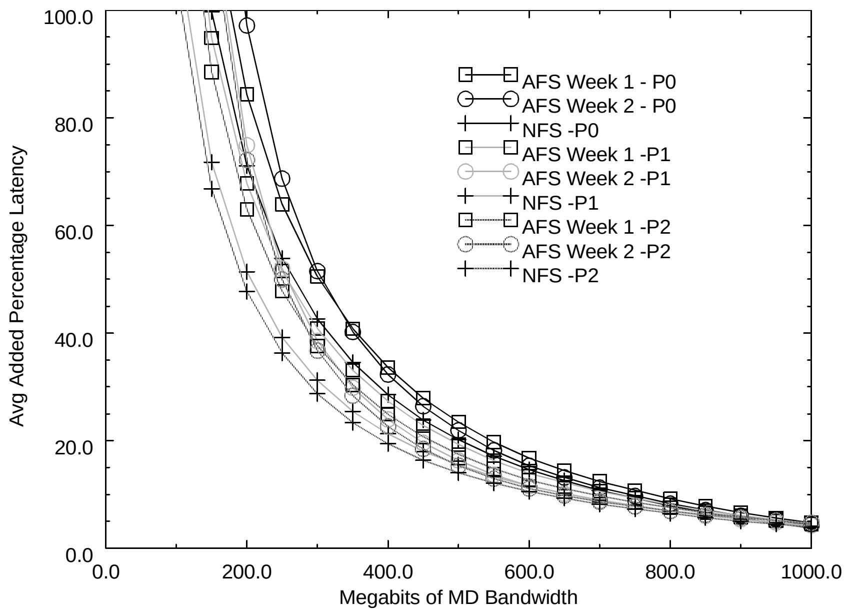


Figure 12: Average additional latency seen by clients. For all workloads, a drive with only 700 Mb/sec of message digest bandwidth adds less than an average of less than 10% additional latency to filesystem requests compared to their latency without security. These simulation result show the impact of having less message digest bandwidth than the full duplex network bandwidth(2Gb/s) for the three sample workloads. The x-axis is the throughput of the SHA-1 unit and the y-axis is the average percentage increase in latency of a filesystem request in comparison to the same request running without security. The additional impact of applying the precompute optimizations is also shown. P0 is using no precompute i.e. all bytes are MAC'd using HMAC-SHA1. P1 is using stored message for each 8K disk block to reduce the computation on a read operation. P2 adds a stored message digest for attributes to reduce computation on GetAttr operations.

5.2 Results

Figure 12 shows the impact of underprovisioned SHA-1 bandwidth, and the impact of precomputing hashing over nothing, disk blocks, and disk block and attributes. Without any optimizations, 600 Mbit/sec SHA-1 only increases latency 20% over a request with no security. Precomputing stored digests and using HierMAC reduces the added latency to about 15% for a 600 Mb/sec MD while precomputation for attributes reduces it by another 1%. Therefore, a NASD system that provides cryptography at only 33% of peak bandwidth will incur a very modest (<20%) increase in latency. Moreover, this is for a drive where all accesses hit in the drive's cache; a drive with accesses to the platters will see an much smaller increase.

Unfortunately, reducing SHA-1 throughput introduces a bottleneck in the drive's queuing system, translating into wider variability in client service time. Figure 13 show that when SHA-1 throughput is badly mismatched to the workload, a large percentage of requests take more than twice as long to service. However, service time quickly converges to less than 1% as security bandwidth increases; for 600 Mb/sec of SHA-1 throughput, less than 0.03% of the requests are outlyers

For each workload, performance can degrade substantially in the worst case (Figure 14). Partially caused when a large request prevents small requests from making progress, the worst case does not change as quickly as the average case because there are brief periods when the entire system is near saturation and substantial queueing can occur. Since the initial simulation processed requests in-order and to completion, one solution is to time slice the SHA-1 resources and prioritize small requests. We simulated this priority scheme using two queues: a high priority queue for

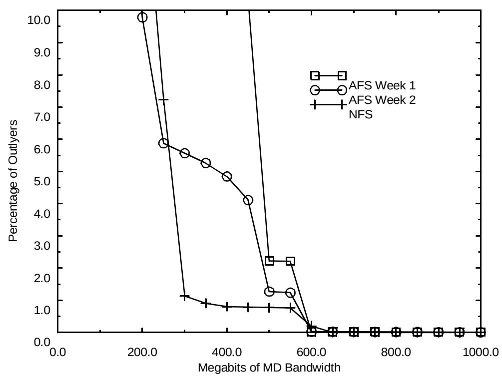


Figure 13: Percentage Outlyers. If message digest bandwidth is less than 500 Mb/s, a large number of requests take twice as long, i.e. outlyers. However, this quickly converges to almost no requests being outlyers. The x-axis is the throughput of the SHA-1 unit and the y-axis the percentage of filesystem level requests where the request service time was twice as long as the time with no-security i.e. requests where the added latency was at least 100%. These are the periods when clients are most likely to noticed the added latency. These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.

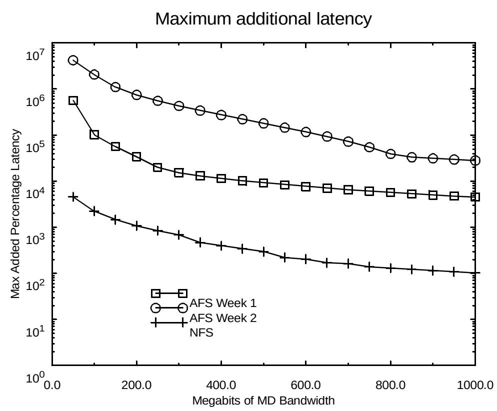


Figure 14: Maximum additional latency seen by clients. Since SHA-1 bandwidth introduces another potential bottleneck, there will always be cases where it introduces queuing and some request takes much longer than normal. The x-axis is the throughput of the SHA-1 unit and the y-axis is the worst case added percentage latencies for each trace in comparison to the non-security version of each request. The worst case is significantly better in NFS because transfers are smaller. The AFS maximums illustrate that the worst scale can vary substantially from trace to trace even in a single filesystem and single user environment.These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.

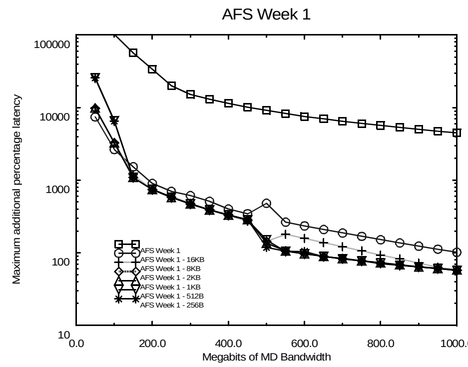
all operations requiring SHA-1 processing on *N* bytes or less and a low priority queue for the rest of the requests. Figure 15 shows that the priority scheme significantly curtails the maximum wait a request may have due to being backed up behind a larger request and reduces the outliers (Figure 16). For AFS workloads, with their larger operations, the time slicing approach significantly improves the worst case since a small request will spend less time stalled behind a large request. However, if the time slicing interval is too small, large requests become starved during periods of heavy activity and the worst case degrades. Time-slicing of the SHA-1 resource does not improve the number of outlyers significantly. In the slow cases, some write operations are now being starved into becoming outlyers. In the fast cases, there are already very few outlyers to the improvement is not very significant.

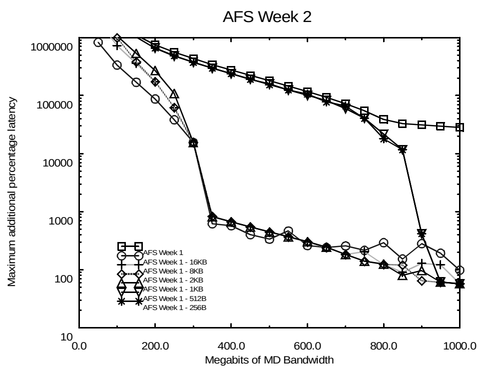
Another solution is for the drive to control its data movement by implementing *pulling* the data from the client on a data write. Logically, pull semantics place the more resource poor drive in control of bandwidth allocation decisions, allowing the drive to schedule data arrivals to meet its buffering constraints and the throughput and availability of its SHA-1 resource. To simulate pull semantics, the drive synchronously handles only the control portion of the write by mapping all writes to 0-byte writes. The drive then pulls the data at its own schedule. Figure 17 shows that for all workloads, employing pull semantics reduced the number of outliers by at least a factor of 4 with 400 Mb/s of SHA-1 bandwidth or more and reduced it to zero for 600 Mb/s or faster systems. This confirms that idea that writes are a major issue for a drive.

6 Summary

This work has shown that by carefully tailoring security to exploit storage's characteristics (e.g., non-volatility, the reuse of data, the wide variation in transfer rates, and the size/types of requests), it is possible to provide high-per-

Figure 15: Impact of time slicing SHA-1 unit on maximum additional latency seen by clients. The simple time slicing approach significantly improves the worst case for AFS workloads. For each of the workloads, we compared the non-slicing case against preempting every 16K, 8K, and 2K bytes to allow smaller jobs to be processed and using the same cutoffs to distinguish between low and high priority operations.The x-axis is the throughput of the SHA-1 unit and the y-axis is the worst case added percentage latencies for each trace in comparison to the non-security version of each request.These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.





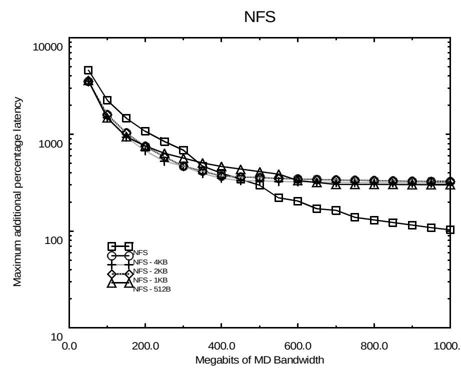
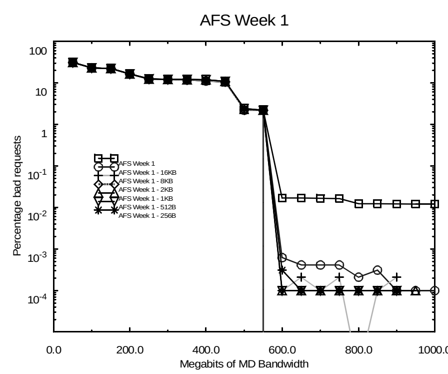
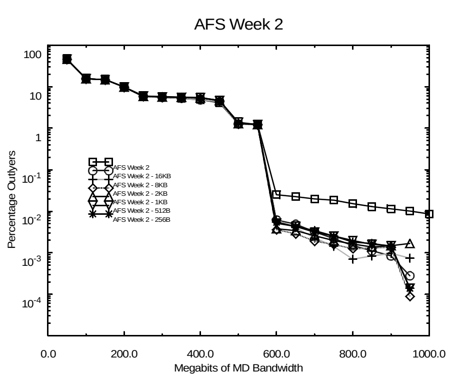
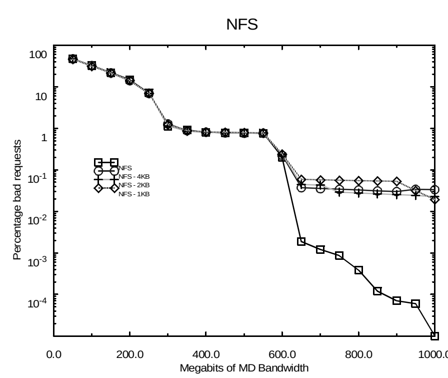


Figure 16: Impact of time slicing SHA-1 unit on percentage of outlyers. Time slicing has a negligible impact on the percentage of outlyers. Note, in contrast to earlier graphs, the Y-axis is in log scale. The x-axis is the throughput of the SHA-1 unit and the y-axis the percentage of filesystem level requests where the request service time was twice as long as the time with no-security i.e. requests where the added latency was at least 100%. For each of the workloads, we compared the non-slicing case against preempting every 16K, 8K, and 2K bytes to allow smaller jobs to be processed and using the same cutoffs to distinguish between low and high priority operations.For each of the workloads, we compared the non-slicing case against at fixed byte intervals to allow smaller jobs to be processed. These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.







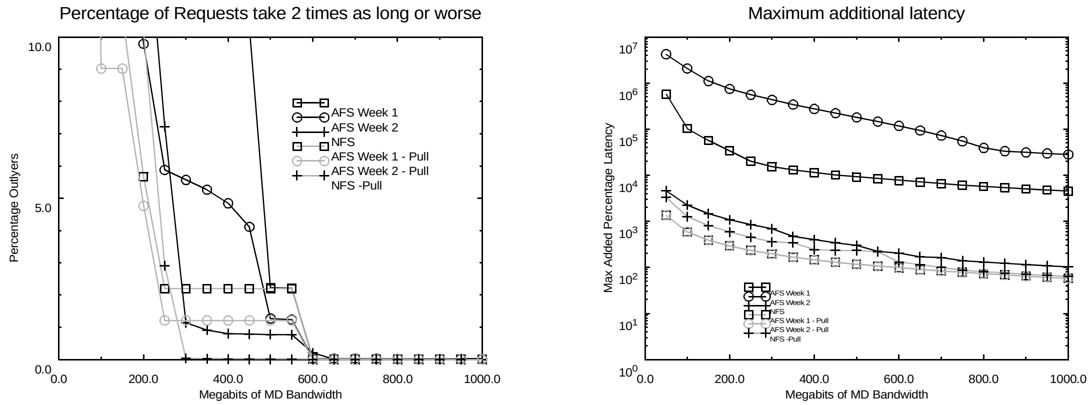


Figure 17: Impact of *pull* semantics on added latency. Pull semantics reduce the percentage of outlyers and improve the worst case for all three workloads. All write operations are mapped to 0-byte writes to approximate the drive synchronously handling the control portion of a write but being able to schedule the data processing to minimize the impact on other requests and efficiently utilize its buffers and limited media bandwidth. On the left, the x-axis is the throughput of the SHA-1 unit and the y-axis the percentage of filesystem level requests where the request service time was twice as long as the time with no-security i.e. requests where the added latency was at least 100%. On the right, the x-axis is the throughput of the SHA-1 unit and the y-axis is the worst case added percentage latencies for each trace in comparison to the non-security version of each request.These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.

formance embedded security at only a fraction of full-speed hardware. Central to this work is the stored digest which allows the drive to precompute and store integrity information that multiple clients can reuse. For aligned, 8-KByte blocks, this reduces the amount of redundant cryptography from 8-KBytes down to only 20 bytes per block. For smaller or unaligned accesses, incremental stored digests can reduce the small-read security penalty by as much as 50% over Hash and MAC.

Modeling these integrity algorithms and the 3DES encryption algorithm reveals how providing integrity, privacy, or both increases storage access latency. Encryption operates on a fine grain pipeline and it has a small impact on latency while message authentication codes produce results at course-grained intervals, significantly increasing latency. HMAC-SHA1 has a very long critical path while HierMAC imposes a much small latency increase on large disk-block aligned reads. Coupling HierMAC with incremental digests allows it to operate on 256 byte blocks in parallel, further reducing the critical path and corresponding latency. Finally, we showed that blind scheduling of data transfers significantly increases worst-case latency by placing small requests behind large, data- and security-intensive commands. By allowing the drive to coordinate data transfers, it is possible to greatly reduce the worst-case latency.

Clearly, faster security hardware is better but faster is also more expensive. These results demonstrate that high-performance drive embedded security can be achieved at a fraction of the peak storage bandwidth, greatly reducing cost while providing a high degree of integrity. While studied in the context of NASD, many of these results are applicable to a wide range of storage architectures, including the popular server-based NFS and WWW. For example, distributed file systems implemented on top of IPsec can significantly improve the scalability and performance of IPsec by using a Hash and MAC style message authentication code. This is not only important, but essential to server machines that will quickly bottleneck under the load of data movement and cryptography. Of course, this does require some integration of the security and file system, but many of the most important performance improvements have come from these types of integration.

Bibliography

[Baker91] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., Ousterhout, J. K., “Measurements of a Distributed File System”, 13th Symposium on Operating Systems Principles, October 1991, pp. 198-212.

[Bellare94] Bellare, M., Goldreich, O., and Goldwasser, S., “Incremental cryptography: the case of hashing and signing”, *Advances in Cryptology - Proceedings of Crypto 94, Lecture Notes in Computer Science Vol. 839*, Springer-Verlag, 1994.

[Bellare96a] Bellare, M., Canetti, R., and Krawczyk, H., “Keying Hash Functions for Message Authentication”, *Advances in Cryptology: Crypto ‘96 Proceedings*, 1996.

[Bellare97] Bellare, M., and Micciancio, D., “A New Paradigm for collision-free hashing: Icrementality at reduced cost,”, *Advances in Cryptology - Proceedings of Eurocrypt 97, Lecture Notes in Computer Science Vol. 1233*, Springer-Verlag, 1997

[Dahlin94] Dahlin, M. et al., “Cooperative Caching: Using Remote Client Memory to Improve File System Performance,” *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 267-280, November 1994.

[Eberle92] Eberle, H., “A High-Speed DES Implementation for Network Applications,” *Advances in Cryptology -- Proceedings of Crypto ‘92, Lecture Notes in Computer Science*, Springer Verlag, pp. 521-539.

[Finley99] Scott Finley, Cognitive Designs, Inc., Personal Communication, January 1999.

[Gibson97] Gibson, G., Nagle, D., Amiri, K., Chang, F., Feinberg, E., Gobioff, H., Lee, C., Ozceri, B., Riedel, E., Rochberg, D., and Zelenka, J., File Server Scaling with Network-Attached Secure Disks, *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Seattle, Washington, June 1997.

[Gibson97b] Gibson, G., Nagle, D., Amiri, K., Chang, F., Gobioff, H., Riedel, E., Rochberg, D., and Zelenka, J., *Filesystems for Network-Attached Secure Disks*, School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-97-118, 1997.

[Gibson98] Gibson, G., Nagle, D., Amiri, K., Butler, J., Chang, F., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., Zelenka, J. “A Cost-Effective, High-Bandwidth Storage Architecture”, *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[Gobioff99] Gobioff, H., “Security for a high-performance commodity storage subsystem”, Forthcoming, 1999.

[Grochowski96] Grochowski, E., Hoyt, R. F., “Future Trends in Hard Disk Drives”, *IEEE Transactions on Magnetics*, Vol. 32, No 3., May, 1996.

[HiFn99] Hi/fn, Inc. HiFn, 7751 Encryption Processor Data Sheet, 1999.

[NIST98] National Institute of Standards and Technology, Advance Encryption Standard Development Effort Webpage, http://csrc.nist.gov/encryption/aes/aes_home.htm

[OceanLogic99] Ocean Logic Pty Ltd, DES core, *www.users.bigpond.com/oceanlogic/des.htm*, January 1999.

[Preneel95] Preneel, B. and van Oorschot, P. C., “MDx-MAC and building MACs from hash functions”, *Advances in Cryptology - Crypto ‘95 LNCS 963*, Springer-Verlage, 1995, pp. 1-14.

[Preneel98] Preneel, B., Rijmen, V., and Bosselaers, A., “Principles and performance of cryptographic algorithms,” *Dr. Dobb’s Journal*, Vol. 23, No., 12, December 1998, pp. 126-131.

[Riedel98] Riedel, E., van Ingen, Catharaine, and Gray, J., “A Performance Study of Sequental I/O on Windows NT”, *Proceedings of the Second Usenix Windows NT Symposium*. Seattle, WA, August 1998.

[Riedel96] Riedel, E., and Gibson, G., “Understanding Customer Dissatisfaction with Underutilized Distributed File Servers”, *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*. College Park, MD. September 1996.

[Rivest91] Rivest, R. “The MD4 Message Digest Algorithm”, *Proceedings of Crypto 90, Lecture Notes in Computer Science Vol. 537*, Springer Verlag, 1991, pp. 303-311.

[Schneier97] Schneier, and Whiting, D., Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor, *Fast Software Encryption, Fourth International Workshop Proceedings*, Springer-Verlag, 1997, pp. 242-259.

[Schlosser98] Schlosser, S., and Schmidt, B., Personal Communication, 12/98

[Soltis96] Soltis, S. et al., “The Global File System”, *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*. College Park, MD. September 1996.

[Stamos84] Stamos, J. W., “Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory. ACM Transactions on Computer Systems 2(2), page 155-180, 1984.

[vanPelt99] van Pelt, P., Marketing Manager, Pijinenburg Custom Chips B.V., Personal Communication, January 1999.

[VLSI99] VLSI Technology,VLSI 115 Datasheet Version 2.0, January 1999.

Appendix

A Security of Hash and MAC

How does the “Hash and MAC” approach effect the security of the system? MACing the concatenation of hash values is very similar to signing them with a public key except it is much faster and does not provide the non-repudiabilty property associated with public key signatures.

If we assume the basic MAC function is secure, is the MAC of hash values secure? When something is considered “secure”, it is normally secure for an arbitrary input. If there was a class of inputs for which it was insecure then the MAC function as a whole would not be secure. An adversary breaks a MAC if they can recover the key or generate a MAC value for a message which they has not seen before. Concretely, if you break “Hash and MAC” by attacking the MAC function then you have defined a set of inputs, the concatenation of hash values, that you can use as an input to the MAC to break the original MAC. By our initial assumption, the MAC is secure so this can not be true.

An adversary could attack “Hash and MAC” through the message digest. “Hash and MAC” trades off some security in exchange for increased performance. An adversary can mount an off-line, essentially computing with no information about the message being attacked, attack against the message digest function. With a normal MAC, an adversary could not start an attack until they were given a message to attack because the result of the key dependent computation was essential to the attack. An adversary can apply arbitrary computational power to *precompute* two data blocks that generate the same digest (i.e., a collision). Alternately, an adversary who observes a series of requests and their associated message digests can attempt to find a second data block that generates the same digest as a given message block (i.e., a second pre-image). The difference is between the adversary being allowed to select both blocks in the collision as opposed to being given one of the blocks, which can be viewed as a challenge, and trying to find a second block which generates the same MAC. As long as NASD uses a strong message digest with a large output, such as SHA-1 or RIPEMD-160 which produce 160 bit outputs, the off-line attack is a small risk. The best current attacks against these message digests requires a brute force search of the input space. In order to find a second pre-image of a given message digest, an adversary expects to compute digests of on average 2^{160} data blocks. A far simpler task, given large amounts of memory, is to find a pair of data blocks which generate the same hash by exploiting the birthday paradox[1] but this attack still expected to require 2^{80} digest calculations.

1. The birthday paradox is a standard statistical problem and a good explanation can be found in [Menzenes98]. The paradox is that the probability of two people in a room having the same birthday is substantially about the square root of the probability of someone in a room having the same birthday as you. This is similar to the relationship between a collision and a second pre-image.

Assuming an adversary is able to find a collision, they can exploit the collision if one of the colliding blocks is already within the storage system and the adversary can replace the in-system block with the out-of-system colliding block in a message exchange. An adversary can potentially tabulate a large number of digests and watch message traffic to the drive for an opportunity but the odds of such an opportunity presenting itself is:

$$2 \times \frac{NumOfCollisions \times NumOfBlocksSeen}{2^{160}}$$

An adversary will have an easier task if they can insert one half of a collision into the storage system and then replace it with the other half. In this case, they could have already written the second of the two blocks to the storage rather than swapping the blocks while they were being read. Thus, an adversary can primarily exploit a collision in a multi-tier system, such as a database system, where write operations are filtered through another host which decides if a writes should be forwarded to the storage. If a collision is found, the adversary can swap a bad data block for the forwarded data block. Because the filtering host is making a decision based on the contents of the initial write request, it is implicitly enforcing some structure on the writes it forwards on to storage. Since one half of the collision must fit the required structure to the filtering host will forward it on, this structure improves security by constraining the set of useful collisions an adversary can theoretically generate.

Because “Hash and MAC” generates multiple independent digests which are used to create the final MAC, an adversary can parallelize an attempt to find a second pre-image of the digests. If the request is divided into r different data blocks, an adversary can attack r different values when trying to find the second per-image of a digest. In contrast, a normal MAC algorithm has a single MAC value that can be attacked because all partially computed values are key dependent and hidden in the MACs internal state. Even for extremely large requests and heavily used storage devices, r will not be large enough substantially reduce the 2^{160} computations required to find a second pre-image. For example, if a client transferred a terabyte of data and the digests were generated on 8K disk blocks then an adversary could attack 2^{22} unique messages digests. This only reduces the work factor from 2^{160} down to 2^{138}. In order for parallelism to reduce the work of finding a second pre-image down to the work of finding a simple collision, the adversary would need to observe 2^{80} disk blocks and attack them in parallel.