Secure Deduplication of General Computations

Yang Tang, Junfeng Yang
Columbia University
{ty,junfeng}@cs.columbia.edu

Abstract

The world's fast-growing data has become highly concentrated on enterprise or cloud storage servers. Data deduplication reduces redundancy in this data, saving storage and simplifying management. While existing systems can *deduplicate computations* on this data by memoizing and reusing computation results, they are insecure, not general, or slow.

This paper presents UNIC, a system that securely deduplicates general computations. It exports a cache service that allows applications running on behalf of mutually distrusting users on local or remote hosts to memoize and reuse computation results. Key in UNIC are three new ideas. First, through a novel use of code attestation, UNIC achieves both integrity and secrecy. Second, it provides a simple yet expressive API that enables applications to deduplicate their own rich computations. This design is much more general and flexible than existing systems that can deduplicate only specific types of computations. Third, UNIC explores a cross-layer design that allows the underlying storage system to expose data deduplication information to the applications for better performance.

Evaluation of UNIC on four popular open-source applications shows that UNIC is easy to use, fast, and with little storage overhead.

1 Introduction

The world's data has been fast exploding for many years. It is estimated that in 2011 alone, 1.8 zettabytes of data were created, and the overall data will grow by 50× by 2020 [21]. This massive amount of data comes in greatly varying forms, ranging from personal photos and videos, to office documents and web pages, to source files, binary programs, and virtual machine images, and to data collected from user clicks or physical sensors.

Meanwhile, the storage of this data has become highly concentrated. It is common practice for enterprises to store data on centralized, powerful storage servers for ease of management [34]. The cloud computing paradigm has migrated data into the cloud so that the computations can be closer to the data. For instance, several organizations have put 56 public data sets totaling 761.2TB onto Amazon Web Services [2]. Even consumers are beginning to aggregate their personal data into the cloud for convenience. For instance, Google, Dropbox, Amazon, and Microsoft all provide the option for users to automatically upload pictures and videos shot using their mobile devices. Facebook stores over 260 billion personal photos [6].

This highly concentrated, massive data poses challenges for storage provisioning and management. Fortunately, prior work has shown that a significant portion of the data is redundant [22] and that *data deduplication* can hugely reduce the storage needed to hold the data and simplify management [13]. For instance, *file deduplication* detects when multiple files have the same data and stores the unique data only once [8]. This scheme is particularly useful when the same file is copied, such as when a user makes a copy of her friend's shared video on Dropbox. *Block deduplication* breaks files down to variable [20, 24] or fixed [36] size blocks and stores each unique block of data once. This scheme is particularly useful for files that are similar but not exactly identical, such as different versions of a document and virtual machine images built from the same OS family. These deduplication schemes have been long prevalent in enterprise storage servers [13]. With the trend of moving consumer data into the cloud, these schemes have also become popular among cloud storage providers such as Dropbox [31].

Not only can data be redundant, the computations on top of the data can also be redundant. For instance, a user may scan her Dropbox files for viruses, while another user runs the same virus scanner on a similar set of files. Different users may be doing the same computations on the public data sets in AWS, such as building an inverted

index for the web pages in CommonCrawl [11]. Given the same input data, the same deterministic computation always produces the same result. Thus, if the computation is slow, it is typically more efficient to memoize [23] and reuse the result than redoing the computation. We term this technique *computation deduplication*.

Several prior systems deduplicate computations (*e.g.*, [9, 15]). However, three main challenges prevent these systems from effectively deduplicating computations in today's cloud or enterprise environments:

First, how can we deduplicate computations done by *mutually distrusting* users? Storage providers such as Dropbox aggregate data from many users who do not necessarily trust each other. Even in an enterprise setting, users frequently have different data access permissions. One naïve approach is to memoize computation results in a cache every user can read or write, but this approach provides neither integrity or security. A malicious user can easily poison the cache, by for instance marking files that contain viruses safe. She can also read results in the cache even though she has no permission to access the actual data in the results. Although this challenge may be solved with information flow tracking or access control systems, these systems are known to be difficult to configure and use.

Second, how can we deduplicate *general* computations? Prior systems deduplicate computations purely at the system level, assuming no cooperation from application developers. As a result, they handle only specific computations. For instance, `ccache` [9] deduplicates only the compilations of C/C++ programs, and Nectar [15] deduplicates the computations of programs written only in DryadLINQ [35], a specially designed language for large scale data-parallel workloads. However, the computations that users want to do on their data can be extremely rich, and it is unrealistic to require storage providers to understand all of them. For instance, while it may be feasible for Amazon to run some basic virus scanning software on the files it hosts, it is impossible for Amazon to understand every advanced virus scanner, every compression tool, and every image/video manipulation utility users want to run on their data.

Third, how can we effectively deduplicate computations on top of *deduplicated data*? Prior systems rely on custom methods to detect that data is redundant. For instance, `ccache` computes a hash of a preprocessed C/C++ source file and uses this hash to search its compilation cache. These methods incur unnecessary overhead when the data is deduplicated because the underlying storage system already knows what data is redundant.

This paper presents UNIC,[1] a system that securely deduplicates general computations. It exports a cache service that allows applications running on behalf of mutually distrusting users on local or remote hosts to memoize and reuse computation results. Key in UNIC are three new ideas:

First, through a novel use of code attestation, a classic primitive to attest what code is running to a (remote) party [29, 30], UNIC achieves both integrity and secrecy. To insert or query the result cache that UNIC maintains, UNIC generates a secure, non-forgeable key that attests to both the application code and the input data. This key strongly isolates applications from each other in the result cache. For instance, if a malicious user modifies the code of a virus scanner in attempt to poison the cached results of this virus scanner, the attempt would fail because the modified code leads to a different key. In addition, since this key is not forgeable, a malicious user cannot query UNIC's cache without already knowing the application code and the input. Since the user knows the code and input already, she can already compute the result by herself.

Second, UNIC provides a simple yet expressive API that enables applications to deduplicate their own rich computations. From a high level, this API supports an application to (1) insert *input* $\rightarrow$ *result* to the result cache UNIC maintains; and (2) query the cache with *input* and get back the cached *result* if any. This application-level computation deduplication design is much more general and flexible than prior system-level designs.

Third, UNIC explores a cross-layer design that allows the underlying storage system to expose data deduplication information to the applications for speed. Applications thus do not need to re-detect whether the input data is redundant. For instance, suppose two files A and B are identical so the filesystem deduplicates them, and UNIC exposes this data deduplication information to the applications. After a virus scanner scans file A, it can immediately skip file B without reading any data from B, significantly increasing its scanning speed.

Our implementation of UNIC stores cached results in Redis, a fast, scalable, replicated key-value store [27]. UNIC implements code attestation in a dynamically loadable Linux kernel module and considers the kernel to be trusted. It implements the computation deduplication API as a library, which applications link with. UNIC leverages ZFS [36], a file system that supports both file and block deduplication, to detect when data is deduplicated on behalf of the applications running with UNIC.

Evaluation of UNIC on four popular open-source applications shows that (1) it is easy to use (to support each application, we needed to change fewer than 1% lines of source code); (2) it is fast (it sped up applications by up to 21.4$\times$); and (3) it incurs little storage overhead (it needed only 3.45% additional storage to cache the results).

The remainder of this paper is organized as follows.

[1] We name our system UNIC (pronounced "unique") because it is conceptually similar to the Unix `uniq` utility applied to computations.

The next section discusses the security model and UNIC's design. §3 describes UNIC's API and usage. §4 presents how UNIC leverages deduplicated data. §5 describes the implementation. §6 shows evaluation results. §7 discusses UNIC's security implications, §8 describes related work, and §9 concludes.

2 Security Model and Design

We begin with UNIC's assumptions, threat model, and the design of UNIC's protocol.

2.1 Assumptions and Non-assumptions

First, UNIC relies on a code attestation mechanism for integrity and secrecy of the cached results. It leverages this mechanism to bind a result to the code and input data that together produce the result. This mechanism can be implemented in multiple ways with different security strengths. For instance, UNIC could use TPM and isolation technologies such as Intel TXT [18] to realize code attestation, but doing so would incur both deployment and runtime overhead, negating our goal of being easy to use and fast. Therefore, for practical reasons, UNIC assumes that the OS is trusted and provides a function to attest the application code, and that the user does not have superuser privileges to interfere with that mechanism. This assumption matches well with many of today's mobile devices that run Chrome OS [14], iOS, and Android.

Second, UNIC assumes correct application code. For instance, when using UNIC, an application developer should use UNIC's API correctly. She should only memoize computations with deterministic results. UNIC also assumes that the application is free of vulnerabilities such as buffer overflows. We note that this assumption is common to almost all prior code attestation work.

Third, UNIC assumes that its underlying storage system provides reasonable security guarantees. To reuse results across sessions, UNIC persists them in an underlying storage system such as a file system. UNIC assumes that this storage system is properly configured such that an attacker cannot access the data stored without going through UNIC. This guarantee and UNIC's security mechanisms described in §2.3 together ensure the integrity and secrecy of its cache of computation results.

2.2 Threats

UNIC enables deduplicating computation among mutually distrusting users. Two attacks are particularly serious for UNIC: *cache poisoning* attacks UNIC's integrity, and *query forging* attacks UNIC's secrecy.

Cache poisoning. A malicious user may write a new application or modify an existing application in an attempt to poison the result cache. Her application may attempt to insert or overwrite entries belonging to a legitimate application. UNIC prevents this attack by isolating applications in the result cache: it guarantees that the cached data for one application can never be accessed by another application. Specifically, UNIC securely binds the computation code and the input data to the computation result leveraging a code attestation mechanism.

Query forging. A malicious user may write a new application or modify an existing application in attempt to query entries in the result cache that she cannot access, and gain information. UNIC prevents this attack again by isolating applications. When an application queries the cache, UNIC generates a search key that attests to both the code and the input data that generate the query. This key is unique to each application. One application thus cannot query entries of another application.

Several other attacks are possible, some of which can be prevented using simple mechanisms such as rate-limiting queries sent to UNIC. We briefly describe how they can be prevented in §7, and leave the implementation for future work.

2.3 Design

UNIC novelly leverages code attestation to cryptographically bind the *result* with the *code* and the *input* that produced the *result*, preventing cache poisoning and query forging attacks.

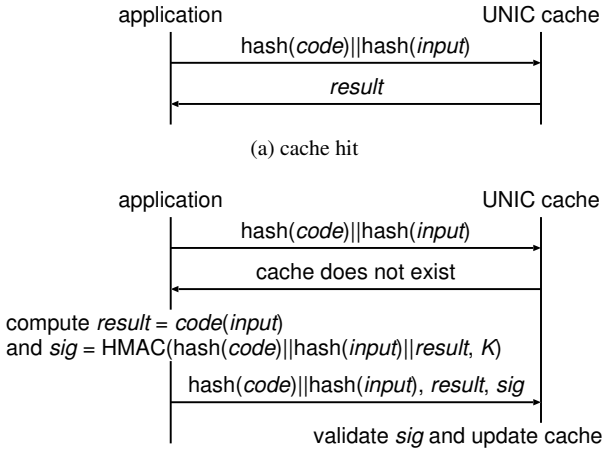
UNIC assumes a trusted OS that securely computes SHA-1 hash and HMAC. A secret key K is shared among trusted OSes. (Existing work [30] details how to distribute this key. We use symmetric key for efficiency; however asymmetric key works, too.) An attacker cannot forge HMAC($data, K$) without knowing K.

UNIC leverages code attestation to bind *result* to *code* and *input* that produced *result*. Specifically, it uses code attestation to compute two things:

(1) $result = code(input)$
// Run *code* on *input* to compute *result*.

(2) $sig = \text{HMAC}(\text{hash}(code)||\text{hash}(input)||result, K)$
// Bind *code*, *input*, and *result*. We use $||$ as the concatenation operator.

The assumptions on trusted OS, unprivileged user, and correct application code together guarantee that *result* is the correct result of running *code* on *input*. This code attestation mechanism further guarantees that (a) *sig* cryptographically attests that *result* is indeed produced by running *code* on *input*, which anyone with access to *code*, *input*, *result*, and K can verify; and (b) *sig* cannot be forged.



(a) cache hit

(b) cache miss

Figure 1: UNIC *protocol.*

UNIC protocol. The UNIC cache is a mapping of

$$\text{hash}(code)||\text{hash}(input) \rightarrow result$$

Since the hash function is collision resistant, the cache space for different computations are isolated.

When an application wants to compute $code(input)$, it sends $\text{hash}(code)||\text{hash}(input)$ to the UNIC cache. If cache exists (Figure 1a), UNIC sends back *result*. If cache does not exist (Figure 1b), the application computes both *result* and *sig*, and sends $\text{hash}(code)||\text{hash}(input)$, *result*, and *sig* to the UNIC cache. The UNIC cache validates that *sig* is indeed $\text{HMAC}(\text{hash}(code)||\text{hash}(input)||result, K)$, and updates the cache.

2.4 Security Analysis

The design of UNIC prevents cache poisoning as follows. Suppose an attacker replaces *result* with *bad_result* when inserting into UNIC. Because of code attestation, she cannot forge *sig*, so UNIC cannot validate *sig*. Suppose she modifies *code* into *bad_code* and computes *bad_result* to poison the cache. Because UNIC validates *sig*, she can only insert

$$\text{hash}(bad\_code)||\text{hash}(input) \rightarrow bad\_result$$

which cannot affect the cache entry of $\text{hash}(code)||\text{hash}(input)$. To avoid a malicious client from polluting the cache space, UNIC can employ a quota mechanism to limit the cache space for each client application.

This design also prevents an attacker from forging a query to steal *result*. To query cache, she must send $\text{hash}(code)||\text{hash}(input)$, so she must already have *code* and *input* because otherwise she would not be able to compute the hashes. Once an attacker has *code* and *input*, she can already compute *result* simply by running *code* on *input* herself. Thus, she cannot gain additional information with this query other than whether there is a result in the cache. §7 further discusses its implications.

```
1: void simple_virus_scanner(file, options) {
2:     buffer = read(file);
3:     result = scan_signature(buffer, options);
4:     print(result);
5: }
```

Figure 2: *A simple virus scanning application.*

3 UNIC API and Usage

UNIC provides a simple yet expressive API for applications to deduplicate their own rich computations. We first motivate our API design through an example, and then formally describe its interface.

3.1 Example

We motivate the design of UNIC API through a step-by-step example showing how a simple virus scanning application could use memoization to deduplicate computation. Conceptually, the application works like Figure 2. It reads the file content into a buffer, executes virus scanning algorithm on the buffer, and outputs the result.

In this piece of code, line 2 reads the file content from disk, potentially a time-consuming I/O operation. Line 3 performs some CPU-bound virus signature matching algorithm, potentially another time-consuming operation. Line 4 prints the result, which is relatively fast because the length of the scanning result (*e.g.*, "no virus found") is much smaller than the original file content. Therefore, we want to improve the performance on lines 2 and 3.

Memoizing Computations. We first examine how to use memoization to avoid duplicate computation on line 3. Since `scan_signature()` is a deterministic function over the input buffer and the signature-scanning options, if we could memoize the result the first time we perform the computation, we would be able to safely reuse the result later on the same input. To do so, we modify the application into Figure 3, using three functions that UNIC provides: `exists()`, `get()`, and `put()`. It first checks if the computation for the given buffer and options exists in the result cache (line 3). If so, it simply gets the memoized result (line 4). Otherwise, it performs the computation as before (line 6) and then puts the result into the cache (line 7).

As discussed in §2.3, the cache is not merely a mapping from the input to the result, but binds the computation code together with them. UNIC internally computes

```
1 : void simple_virus_scanner(file, options) {
2 :   buffer = read(file);
3 :   if (exists(scan_signature, buffer, options)) {
4 :     result = get(scan_signature, buffer, options);
5 :   } else {
6 :     result = scan_signature(buffer, options);
7 :     put(scan_signature, buffer, options, result);
8 :   }
9 :   print(result);
10: }
```

Figure 3: *First step: memoize the computation result.*

```
1 : void simple_virus_scanner(file, options) {
2 :   hash = get_file_hash(file);
3 :   if (exists(scan_signature, hash, options)) {
4 :     result = get(scan_signature, hash, options);
5 :   } else {
6 :     buffer = read(file);
7 :     result = scan_signature(buffer, options);
8 :     put(scan_signature, hash, options, result);
9 :   }
10:   print(result);
11: }
```

Figure 4: *Final version: use filesystem metadata to further reduce I/O operations.*

a non-forgeable authentication code that guarantees that the result (`result`) is indeed generated by the computation code (`scan_signature()`) over the input (`buffer` and `options`). The result cache is updated only if it can verify this authentication code.

Reducing I/O Operations. Memoizing the computation is good, but it would be better if we could also eliminate the need of reading the file content on line 2. This is not trivial because if we did not read the file in the first place, we would never know if the signature scanning is performed on the same content. Fortunately, it is possible if the file is stored on a deduplication-enabled storage.

A deduplication-enabled filesystem, such as ZFS [36], stores all files with the same content as a single copy. It does so by identifying the file content using a cryptographically collision-resistant hash (*e.g.*, SHA-256), and mapping all files with the same content to the same hash. These hashes are stored on the filesystem metadata, separate from the actual file content. Therefore, it creates a perfect opportunity for our application to tell if the file contents are the same without actually reading them.

Figure 4 shows the final version of the application. Instead of reading the file content up front, it now gets the unique hash of the file directly from the filesystem metadata using UNIC's `get_file_hash()` function (line 2), and uses the hash to identify the memoization (lines 3, 4, and 8). Since getting the hash is much faster than reading the whole file, we have further avoided the slow I/O operation when reusing a previously cached computation.

In practice, when using UNIC, the application developer does not need to worry whether the storage has deduplication enabled or not — she should always follow the final version in Figure 4 and use hash to identify the memoization. This is because UNIC *transparently* leverages storage deduplication information. Where such information is absent, UNIC computes and caches the hash by itself. This process is detailed in §4.

3.2 The API

The previous example illustrates the usage of the UNIC API which we now formally describe. It wraps OS- and filesystem-specific details by exporting the following functions:

- `init()` initializes UNIC.
- `get_file_hash(file)` returns the hash of a file, where `file` can be the name of a file, a file descriptor, or an inode number. If the underlying filesystem has deduplication enabled (*e.g.*, ZFS), it gets the hash of the file from the filesystem metadata without reading the file content. Otherwise, it computes the hash from the file content using libcrypto.
- `get_block_hash(file, block)` is similar as above, but returns the hash of a block of a file, where `block` specifies the block number. This is particularly useful if the application's computation is based on blocks, such as a `bzip2` compression. The application should decide whether to use `get_file_hash()` or `get_block_hash()` based on its own logic, which is discussed in §4.
- `exists(computation, hash, id)` checks if a given computation and input exists in the result cache. The parameter `hash` is the hash of input data. The parameter `id` is an optional string identifier defined by the application, used for differentiating multiple computations performed on the same input. For example, the virus scanning application may let `id` be the signature-scanning options.
- `get(computation, hash, id)` gets the result of a given computation and input from the result cache.
- `put(computation, hash, id, result, ttl)` puts an entry of computation, input, and result into the result cache. An optional `ttl` specifies its time-to-live in seconds, and the result cache automatically deletes the entry upon expiration.

4 Leveraging Storage Deduplication

UNIC explores a cross-layer design allowing underlying storage system to expose data deduplication information to the applications.

Typically, a deduplication-enabled filesystem maintains the hash of each file as its metadata. Since UNIC also uses hash to identify the memoization input, it is both convenient and efficient to leverage such filesystem metadata. Therefore, when an application needs to get a hash, UNIC automatically detects the underlying storage system type, and returns the hash directly from the metadata if the filesystem has enabled deduplication. If not, UNIC reads the file content and computes the hash itself. In this way, UNIC provides a consolidated interface for both scenarios, making the storage system details transparent to the applications.

Furthermore, the application does not need to know whether the underlying storage system is file-level or block-level deduplicated. It should decide whether to use `get_file_hash()` or `get_block_hash()` solely based on the application's own logic. Generally, if the application's computation works with the file on a block-by-block basis, such as the `bzip2` compression algorithm, it should use `get_block_hash()`. Otherwise, if the application's computation uses the file as a whole or randomly accesses the file, such as an anti-virus program, it should use `get_file_hash()`.

5 Implementation

We now describe UNIC's components and implementation details.

5.1 UNIC Components

Figure 5 shows the architecture of UNIC. It is deployed on a network of multiple hosts. Each user can log into multiple hosts, and each host can have many users logged in. Because of UNIC's security design (§2), different users do not need to mutually trust each other.

The UNIC module on each host handles application's memoization requests. Since memoization works best when the reuses of computations are frequent, reading data from the result cache should be more common than writing data to it. In light of this, we design UNIC to make read operations as fast as possible. A trusted master cache server handles all write operations. It can be either standalone or co-located with the enterprise's storage (*e.g.*, NFS) server. Each host has an optional read-only slave cache, which periodically syncs from the master cache server. If the slave cache is present, all read operations happen locally. For security, all network communications are encrypted with SSL/TLS. To reduce the handshake latency, the UNIC module on each host establishes a connection with the master cache server when the host boots up, and keeps the connection alive.

Because data updates on the slave caches happen asynchronously, it is possible that a host does not have the latest cached results. However, we point out that memoized computations are deterministic (§2.1), therefore the consistency on the slave caches should not affect the integrity of computations. The only contingency would be that an application may not be able to leverage recently cached results but have to compute on its own.

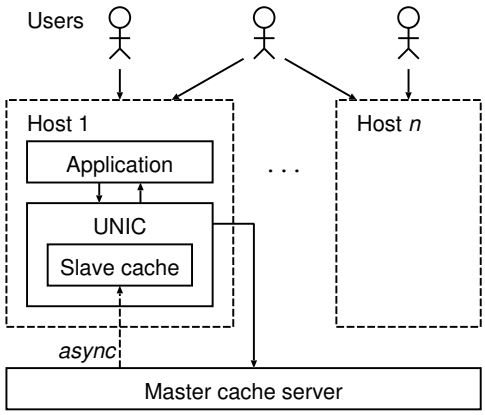


Figure 5: UNIC *architecture.* Additional hosts each have the same architecture as Host 1, and are omitted here due to limited space.

UNIC inserts a kernel module into the Linux kernel as a virtual device for computing hash(*code*) and *sig*. It represents *code* by the image of the executable process, with all libraries statically linked. The secret key K is inaccessible to the user space. The user-space application talks to the kernel module via `ioctl`. For improved performance, the kernel module internally caches hash(*code*) for each caller.

UNIC uses a modified Redis key-value store [27] as the result cache. It modifies Redis to support UNIC's protocol (§2.3), and removes nonessential functions (such as `KEYS` which can list all cache entries) from Redis for security. Therefore, users cannot access the result cache except through UNIC.

5.2 Opportunistic Memoization

When using UNIC, the application developer needs to judge the best opportunity to use memoization because of two reasons. First, memoizing an already-fast computation may not justify the overhead of accessing the result cache. Second, abusing memoization for low-redundancy computations could result in exceeded overhead for entries that are never reused later. However, making the optimal decision at compile time is usually hard because input data cannot be predicted. Therefore, UNIC provides an optimization to opportunistically enable memoization only when the computation is slow and its reuse happens to be frequent at runtime.

To do so, UNIC internally has a model of T_{put}(`result_size`) and T_{get}(`result_size`), meaning

how long it would take to put and get a certain size of result, respectively. This model is independent of the actual content of the result, and it can be learned from a microbenchmark upon the installation of UNIC (see §6.2.1 for our evaluation). UNIC also maintains an accumulator t_{save} for each computation, initialized to 0, for the total time that could have been saved for the future.

UNIC further provides two functions for an application to mark the boundary of a computation. An application calls `begin()` to indicate that a computation starts, and UNIC records the current timestamp as t_{begin}. An application calls `end()` to indicate that the computation has finished, and UNIC records the current timestamp as t_{end}. When `put()` is called, UNIC does not put the data into the result cache immediately, but updates t_{save} to be

$$t_{save} = t_{save} + t_{end} - t_{begin} - T_{get}(\texttt{result\_size})$$

Therefore, the slower and the more frequent a computation is, the larger t_{save} becomes. UNIC only performs the `put()` operation when t_{save} is greater than $T_{put}(\texttt{result\_size})$, *i.e.*, the time that could have been saved from a computation is greater than the time that would be spent for memoizing the computation. In the case that $t_{save} < T_{put}(\texttt{result\_size})$, UNIC ignores the `put()` request, and simply updates t_{save}.

6 Evaluation

We evaluated UNIC on a workstation with an Intel Core i7-2600 CPU and 32GB RAM, running Fedora 20 with Linux 3.16.2. The cache server was running Redis 2.6.17. Our goal is to show that UNIC significantly improves performance with memoization while requiring minimal developers' effort and storage space.

The rest of this section focuses on three questions:
§6.1 Is UNIC easy to use?
§6.2 Does UNIC reduce computation time?
§6.3 What is UNIC's storage overhead?

6.1 Application Adaptation Effort

To evaluate whether UNIC is easy to use, we picked four popular open-source applications that we use daily: (1) `clamav`-0.98.1, an anti-virus software that scans a directory for viruses [10]; (2) `pbzip2`-1.1.8, a multi-threaded compression utility that compresses a single file [25]; (3) `grep`-2.18, a tool that searches for a regular expression within one or many files; and (4) the compiler `gcc`-4.8.3. We adapted them to use UNIC's API[2]. We used file-level memoization for `grep`, `clamav`, and `gcc`, and block-level memoization for `grep` and `pbzip2`.

[2]Our adaptation of `gcc` is based on `ccache` [9].

Application	Total LoC	Changes	Percentage
`clamav` (file)	1,732,762	12	<0.01%
`pbzip2` (block)	4,376	18	0.41%
`grep` (file)	9,658	35	0.36%
`grep` (block)	9,658	69	0.71%
`gcc` (file)	29,023	30	0.10%

Table 1: *Lines of code changed for each application.* Parenthesis indicates whether the adaptation uses file-level or block-level memoization. The numbers for `gcc` are based on `ccache`.

Table 1 shows the lines of changed code for each application to use UNIC's APIs. Changing dozens of lines (<1% of total lines) suffices for all these applications.

To further illustrate, we next present how we adapted `grep`, the application with the most code changes.

6.1.1 Case Study: `grep`

GNU `grep` is a line-based pattern searching utility. To invoke `grep`, the user specifies a search pattern and the path to a file or directory. Then `grep` iterates through all files in the directory and search for the pattern.

Common to all applications, the first step is to add a call to `init()` at the beginning of `main()` in order to initialize UNIC. For `grep` specifically, there are two design choices: we can memoize either at file-level or at block-level. Memoizing at file-level is faster when the whole file is unchanged, whereas memoizing at block-level can exploit sub-file similarities for different files. Next we discuss each of them.

File-level Memoization. Adapting `grep` for file-level memoization is relatively straightforward. When `grep` works on a new file, we call `get_file_hash()` to get the hash of the file from ZFS and call `exists()` to check if there is a corresponding entry in the result cache. If so, we call `get()` to retrieve the memoized result, output it, and move on to the next file. If not, we follow the original algorithm and call `put()` to memoize whatever is output. We also call `put()` to memoize the number of matched lines in the current file, which `grep` uses for internal bookkeeping purposes.

Block-level Memoization. Adapting `grep` to memoize at block-level requires tighter integration with its workflow. For each file, `grep` reads its content in 32KB chunks, and performs pattern searching one chunk at a time. However, since the searching is line-based (delimited by '\n'), it is possible that lines are not well-aligned with chunk boundaries. For example, one line may span across the end of the previous chunk and continue at the following chunk. In this case, `grep` adjusts its chunk boundary to include the residue of the line in the previous chunk and exclude the partial line at the end of current chunk, as shown in the shaded region in Figure 6.

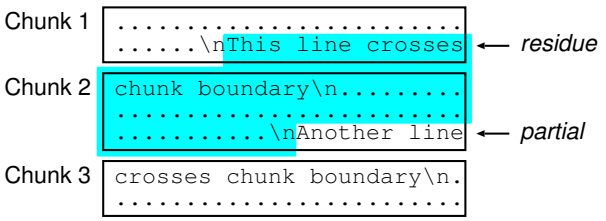


Figure 6: *Misalignment between line and chunk boundaries in* `grep`. Shaded region is the adjusted chunk for computation.

Unfortunately, this poses a challenge to using UNIC directly, because ZFS keeps hash metadata only for entire aligned 32KB disk blocks. On the other hand, we cannot simply use the hash of the unadjusted chunk to address the cache, because this would err if two chunks were the same but their residues in the previous chunk differed. Our solution is to combine the hash of all chunks from the beginning of the residue until the current chunk. Note that this may lose the rare opportunity of reusing memoized results for chunks who only differ at the last partial line, but it preserves correctness nevertheless.

Our experience with adapting the other three applications were straightforward. Overall, we found UNIC easy to use and the adaptation effort was generally little.

6.2 Performance

To understand the performance of UNIC, we first use microbenchmarks to evaluate the throughput of UNIC's basic operations. We then run UNIC on four real-world applications to see how UNIC reduces application running time. Next, we study how UNIC is able to reuse previous computation results for some evolving data. Finally, we study how UNIC performs with a group of multiple users whose data are similar yet different.

6.2.1 Microbenchmark

We first use microbenchmarks to evaluate the throughput of the `get()` and `put()` operations. We wrote a program that calls `put()` 10,000 times followed by calling `get()` 10,000 times. The hashes of the 10,000 entries are all different, and we varied the result size from 1KB to 1MB.

Figure 7 shows the results, where each data point is an average of 10 individual experiments with an error bar showing the maximum and minimum value in the 10 experiments. The x-axis is the size of the memoized result. The y-axis is the total time in performing the 10,000 operations. The solid line is for `put()` and the dashed line is for `put()`. From the results we find that the time for an operation is on the order of ten microseconds when the memoized result is small in size (<10KB), which is mostly the case (see §6.3). Even if the memoized result is as large as 1MB, the time to get a memoized entry is only

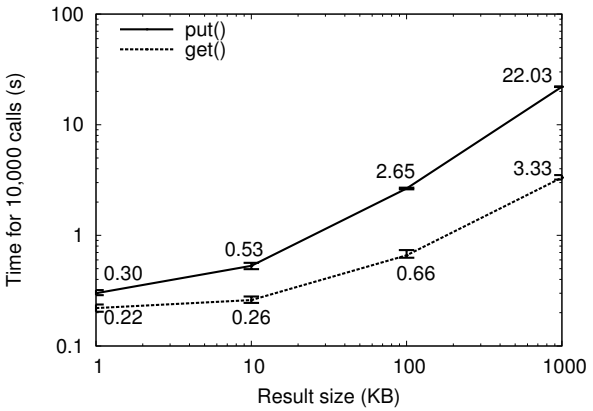


Figure 7: *Throughput of* `put()` *and* `get()` *operations.* The x-axis is the size of memoized result. The y-axis is the total time in performing 10,000 `put()` (solid line) and `get()` (dashed line) operations.

0.33ms, which is normally much faster than doing real computation on that size of data. Therefore, UNIC's basic operations are sufficiently fast for doing useful caching of computations.

6.2.2 Application Performance

We next show how real-world applications benefit from UNIC, and how storage deduplication further helps. We conducted the following experiments. (1) We used `clamav` to scan for viruses on two data sets. The first is the linux-3.12 kernel source code tree. The second is the Dropbox folder for one of the co-authors, which contains 10.8GB of documents, music, pictures, videos, and applications. (2) We used `pbzip2` to compress `linux-3.12.tar` into `linux-3.12.tar.bz2`. (3) We ran `grep` on two data sets. The first is the linux-3.12 kernel source code tree, which consists of 47,336 small files totaling 508MB. The second is the tags file of the linux-3.12 kernel source code generated by `ctags -R`, which is a single text file of 250MB. For each data set, we ran a simple query ('`void`') and a complex query ('`^\s*struct\s+\w+\s+\**\s*\w+\s*=\s*\w+\((\w+(,)*)+\);`' for the source code tree, which matches declaring and initializing a structure pointer to the return value of a function, such as "`struct task_struct *task = get_proc_task(inode);`", and '`/[A-Za-z]+\.c.*d.*file`' for the tags file, which matches a specific type of tag). (4) We used `gcc` to compile linux-3.12 kernel with the `allnoconfig` configuration. Because `gcc` has a nontrivial way to represent input dependencies for cache reusability rather than a file hash, our adaptation does not leverage storage deduplication information. All data files are on a freshly-formatted ZFS disk with cold buffer cache.

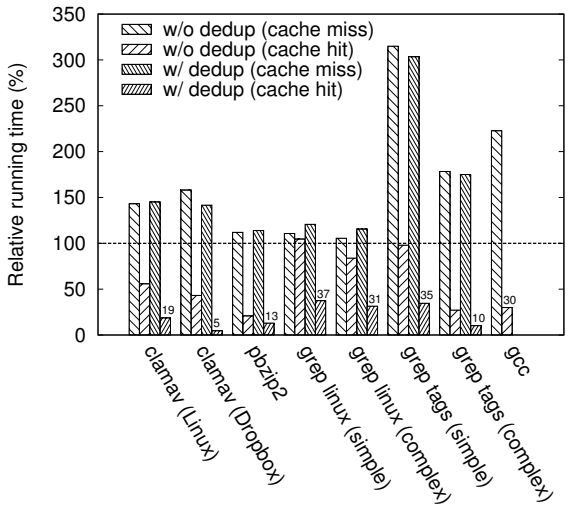


Figure 8: *Relative running time of applications.* The y-axis is the running time relative to the original application. For each cluster, the first bar is cache-miss execution without FS deduplication, the second bar is cache-hit execution without FS deduplication, the third bar is cache-miss execution with FS deduplication, and the fourth bar is cache-hit execution with FS deduplication. The dashed line at 100% shows the running time for the original application.

For each application, we compared the running time (1) without UNIC (the baseline), (2) with UNIC but without filesystem deduplication (the first and second bars on Figure 8), and (3) with both UNIC and filesystem deduplication support (the third and fourth bars). For experiments with UNIC, we further compared the running time (1) for execution on an initially empty result cache, causing cache misses and thus putting entries to the cache (the first and third bars), and (2) for execution when the result cache had already been pre-populated, causing cache hits (the second and fourth bars).

Figure 8 shows the running time for each experiment. Each number is an average of 10 individual runs. Although running applications on an empty result cache incurs an average overhead of 68.2%, running them on a warm result cache gives an average speedup of 2.39×. If filesystem deduplication is available, the average overhead of cache-miss execution drops to 59.3% and the average speedup with memoization increases to 7.58×. Furthermore, complex computations (*e.g.*, scanning for viruses or compressing a file) benefit the most from memoization (up to 21.4× speedup), while simple computations (*e.g.*, searching for a short string) suffer more from the cache-miss overhead. Therefore, opportunistically enabling memoization would be the best practice. With our strategy described in §5.2, memoization is enabled at the second occurrence of `put()` for one application ("`grep` tags" with simple query), and at the first occurrence for all other applications.

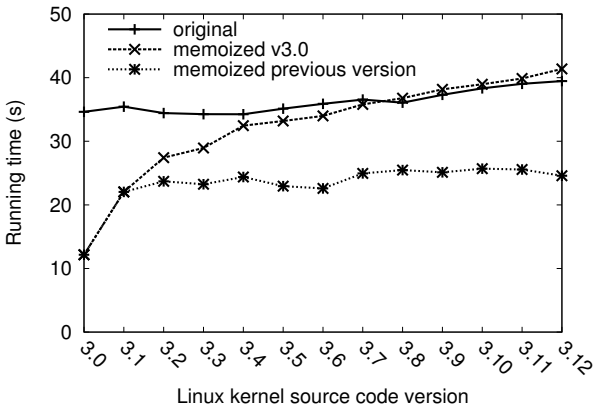


Figure 9: *Effectiveness of memoization with evolving data.* Solid line is the original `grep` without memoization. Dashed line has the result cache populated with v3.0. Dotted line has the result cache populated with the immediate previous version.

6.2.3 Effectiveness with Evolving Data

The previous evaluation focused on the memoization benefit on exactly the same computation. Next we show the effectiveness of memoization if the input data is evolving, *i.e.*, if UNIC has memoized computation on an old version of data, how it can speed up computation on a new version of the data.

We used `grep` to search for '`void`' on thirteen major versions of the Linux kernel source code, from v3.0 to v3.12. All files are on a freshly-formatted deduplication-enabled ZFS disk with cold buffer cache. We performed three sets of experiments. The first one used the original `grep` without UNIC. In the second experiment, we first populated the result cache when running `grep` on v3.0, and then measured the time for running `grep` on each version based on the same memoization of v3.0. In the third experiment, we ran `grep` on each version in a "rolling" manner, *i.e.*, each execution was based on the memoization of the immediate previous version, which resembles a more practical scenario.

Figure 9 shows the running time for all executions, where each number is an average of 10 runs. With a single memoization of v3.0, the speedup is significant for running on v3.1 (1.61×), but diminishes along the increment of version number, and eventually becomes ineffective after v3.8, because the source code differs significantly from the memoized version and the cache hit rate drops below 0.3. On the other hand, when memoized the immediate previous version, the speedup is almost constant, with an average of 1.50×. The reason is that the amount of source code difference is almost constant between each two consecutive versions, and many mem-

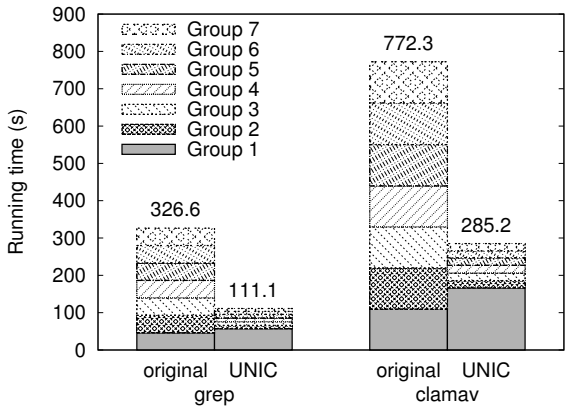


Figure 10: *Effectiveness of memoization across users.* For each cluster, the first bar is the original application, and the second bar is the application modified to use UNIC. Each bar shows the breakdown of running time on each group, the number on top showing the total time.

oized results can be reused (hit rates are between 0.73 and 0.81). Therefore, UNIC is more effective when the divergence of the actual input data from the memoized data is small, which is likely true in a practical scenario.

6.2.4 Effectiveness with Multiple Users

We next evaluate the memoization effectiveness for multiple users with similar yet different data. We took the project directories of seven groups of students in a graduate-level operating system course offered by our university. The average size of each directory is 1.6GB. We performed two executions on each group's directory: (1) use `grep` to search for '`void`', and (2) use `clamav` to scan for viruses. This resembles the enterprise setting where multiple people working on the same project have similar data and perform common computing tasks such as virus scanning. The result cache was originally empty, and was gradually filled by UNIC during the process.

Figure 10 shows the breakdown of each application's running time on each group. The trend is that the original application takes almost the same amount of time for all groups. With UNIC, although the first group takes longer time to execute (24.1% for `grep` and 51.9% for `clamav`), all subsequent groups consistently take a much shorter time (5.17× speedup for `grep` and 5.57× speedup for `clamav`). This is because for the first group, all computations are new and UNIC needs to insert them to the result cache. Once this is done, all subsequent groups can benefit from it. The overall speedups for the executions on all seven groups are 2.94× for `grep` and 2.71× for `clamav`. We foresee that with more number of groups the overall speedup should be even higher. Therefore, UNIC is practical for a group of users working together or doing similar tasks.

6.3 Storage Space

We now evaluate the storage overhead of UNIC. For each application we used for the performance evaluation in §6.2.2, we examined the number of entries in the result cache. To study the total space used for memoization, we also let Redis dump a snapshot of all data and measured the size of the dump file.

Table 2 shows the results. Column (a) is the number of input files. Column (b) is the total size of input files. Column (c) is the number of entries in the result cache. Column (d) is the size of the Redis dump file. The relative storage overhead is thereby Column (d) divided by Column (b), which is shown in Column (e). The results depict that the average overhead of the memoization storage for all applications is 3.45%, negligible compared with the storage of all file data. Therefore, UNIC incurs little storage overhead.

7 Discussion and Limitations

We discuss UNIC's security implications and limitations.

Denial-of-service attacks. A malicious user may issue a large number of put requests on manufactured inputs, and pollute the result cache with useless results. Several approaches can be used to defend against it. For example, UNIC may rate-limit puts to the result cache, employ a quota mechanism to limit the cache space for each client application, or enforce time-to-live limits on cached results. We argue that even if the result cache is full, the worst outcome would be that future computations cannot be memoized and have to be recomputed, yet the secrecy and integrity of computations are not violated.

Side-channel information leakage. A malicious user may enumerate through a large set of inputs on an application, and observe if some executions are significantly faster than others. Based on the observed timings, she may infer what computations have been done by other users and what have not. While defending against this side-channel attack is out of the scope of this paper, we note that the application developers may defend against it by rate-limiting queries to the result cache or randomly forcing cache misses even if the result exists in the cache.

Brute-force attacks. A malicious user may enumerate through all possible hash values of the application code and input, in hopes of getting cached results. We argue that the possibility for an unprivileged user to get a valid hash is minimal. Even if she manages to get an entry, she only knows the result, but she cannot generate the original code and input from the hash. In the example of virus scanning, she might brute-force a hash and discover the result of scanning some file, but she cannot determine the original content of that file. Again, UNIC

Application	(a) File count	(b) File size	(c) Entry count	(d) Dump size	(e) Overhead
`clamav` (Linux)	47,336	508.1MB	44,277	2.8MB	0.55%
`clamav` (Dropbox)	2,792	10.8GB	82,061	4.4MB	0.04%
`pbzip2`	1	544.0MB	4,151	106.4MB	19.55%
`grep` linux (simple)	47,336	508.1MB	70631	11.2MB	2.21%
`grep` linux (complex)	47,336	508.1MB	51532	4.2MB	0.83%
`grep` tags (simple)	1	250.0MB	2	5.3MB	2.13%
`grep` tags (complex)	1	250.0MB	2	4.5MB	1.80%
`gcc`[3]	47,336	508.1MB	522	2.3MB	0.46%

Table 2: *Storage overhead.* Columns are: (a) the number of input files, (b) total size of input files, (c) number of entries in the result cache, (d) size of the Redis dump file, and (e) relative storage overhead.

may defend against this attack by rate-limiting queries to the result cache. Furthermore, if the result is sensitive by itself (*e.g.*, `cat`), the application developer may encrypt it before putting it to the result cache, or the system administrator may disable UNIC for such applications.

Application bugs. Ensuring bug-free code is a hard problem orthogonal to UNIC and code attestation. If the application contains a bug such as buffer overflow, a malicious user may exploit the bug to poison the result cache. Existing systems such as baggy bounds checking [1] and AddressSanitizer [28] can prevent many memory access bugs. Other countermeasures include letting the application rerun the computation and verify the cached result periodically, and purging the result cache when a bug is found. In addition, using hardware-enforced isolation mechanisms such as Intel TXT [18] with TPM, or Intel SGX [5, 17] may avoid this issue.

8 Related Work

Storage deduplication. Storage deduplication reduces data redundancy at either file-level [22] or block-level [12, 32]. ZFS [36] is a widely used cross-platform filesystem that does block deduplication at the time data is written. These works are orthogonal to UNIC, and UNIC's cross-layer design allows it to transparently leverage storage deduplication information.

Ad-hoc caching. Many applications use ad-hoc caching to improve performance, but they either trust all users, or simply disallow cross-user caching. For example, `ccache` [9] caches compiler outputs on the local filesystem, but the cache can be easily exploited or poisoned by any user. On the other hand, `clamav` [10] only caches virus scanning results within a single session, rendering cross-session and cross-user caching impossible. UNIC improves the status quo with strong security guarantees.

Memoization. Memoization [19, 23, 26] is a technique that reuses prior computation results of functions without side effects. Vesta [16] uses memoization for software configuration management. Nectar [15] memoizes intermediate results from DryadLINQ [35] programs. Incoop [7] uses memoization to build a MapReduce framework for incremental computations. However, these systems handle only specific computations, and it is non-trivial to generalize their use cases. UNIC can be used to deduplicate general computations.

Code attestation. Many code attestation techniques exist to provide integrity of computations. For example, result-checking [33] verifies the result produced by a program by computing it in two ways. Secure boot mechanisms [3, 4] verify the integrity of the software stack after booting. BIND [30] ties the proof of what computation has been run to the result that the computation has produced. Pioneer [29] provides code integrity guarantees for running software on an untrusted system. UNIC makes novel use of the code attestation mechanism to protect the secrecy and integrity of memoization.

9 Conclusion

We presented UNIC, a general system for applications to securely deduplicate their rich computations. It uses code attestation mechanism to achieve both secrecy and integrity. It explores a cross-layer design that allows applications to leverage storage deduplication information for speed. Evaluation results show that UNIC is easy to use, speeds up applications by up to 21.4×, and incurs little storage overhead.

Acknowledgments

We thank Yinzhi Cao, Gang Hu, David Williams-King, Xi Wang (our shepherd), and the anonymous reviewers for their valuable comments. This work was supported in part by AFRL FA8650-11-C-7190 and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1054906; an NSF CAREER award; an AFOSR YIP award; and a Sloan Research Fellowship.

[3] Not all files are used for compilation due to our experiment configuration.

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 51–66, 2009.

[2] Amazon Web Services. Public data sets. `http://aws.amazon.com/datasets`.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, 1997.

[4] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith. Automated recovery in a secure bootstrap process, 1998.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the Eleventh Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 267–283, Oct. 2014.

[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010.

[7] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[8] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in windows 2000. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, 2000.

[9] ccache. `http://ccache.samba.org/`.

[10] Clam AntiVirus. `http://www.clamav.net/`.

[11] CommonCrawl. `http://commoncrawl.org/`.

[12] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A scalable secondary storage. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, 2009.

[13] L. DuBois, M. Amaldas, and E. Sheppard. Key considerations as deduplication evolves into primary storage. White Paper 223310, Mar. 2011.

[14] Google Chrome OS. `http://www.google.com/chromeos/index.html`.

[15] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[16] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, 2000.

[17] Intel. Software guard extensions programming reference. `https://software.intel.com/sites/default/files/329298-001.pdf,`.

[18] Intel. Intel trusted execution technology: White paper. `http://www.intel.com/content/www/us/en/trusted-execution-technology/trusted-execution-technology-security-paper.html,`.

[19] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):546–585, May 1998.

[20] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, 1994.

[21] L. Mearian. World's data will grow by 50x in next decade, IDC study predicts. `http://www.computerworld.com/s/article/9217988/World_s_data_will_grow_by_50X_in_next_decade_IDC_study_predicts`.

[22] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14:1–14:20, Jan. 2012.

[23] D. MICHIE. "memo" functions and machine learning. *Nature*, 218:19–22, Apr. 1968.

[24] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, 2001.

[25] PBZIP2. Parallel BZIP2 (PBZIP2). http://compression.ca/pbzip2/, 2011.

[26] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, 1989.

[27] Redis. http://redis.io/.

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (USENIX '12)*, 2012.

[29] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, 2005.

[30] E. Shi, A. Perrig, and L. van Doorn. BIND: a fine-grained attestation service for secure distributed systems. In *Security and Privacy, 2005 IEEE Symposium on*, pages 154–168, May 2005.

[31] C. Soghoian. How Dropbox sacrifices user privacy for cost savings. http://paranoia.dubfire.net/2011/04/how-dropbox-sacrifices-user-privacy-for.html.

[32] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, 2010.

[33] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM (JACM)*, 44(6):826–849, Nov. 1997.

[34] Webopedia. Enterprise storage. http://www.webopedia.com/TERM/E/enterprise_storage.html.

[35] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level languag. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, Dec 2008.

[36] ZFS: the last word in file systems. http://www.sun.com/2004-0914/feature/.