Anonymization Techniques for URLs and Filenames

Technical Report UCSC-CRL-03-05

Geoff Kuenning
geoff@cs.hmc.edu
Harvey Mudd College

Ethan L. Miller
elm@cs.ucsc.edu
University of California, Santa Cruz

Storage Systems Research Center
Jack Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://www.ssrc.ucsc.edu/

September 2003

Abstract

Investigating the behavior of computer and network systems often involves collecting and analyzing traces of user activity. Because traces are expensive to store and collect, and because it is desirable to replicate experimental results, it is common to publish trace data for use by other researchers. To maintain the privacy of users, traces are usually anonymized before publication. We discuss prior methods for anonymizing such traces, including weaknesses and drawbacks in those methods. We then present a new method that can anonymize strings such as URLs while preserving a maximum amount of information useful to researchers.

1 Introduction

In an experimental field such as computer systems science, it is common to study real-world behavior as a means of gaining insight. One time-honored methodology is the collection of trace data, either as a snapshot or over a period of time, for later replay or analysis. Such studies have analyzed file system behavior [9, 10, 14, 17, 19, 23, 22, 28, 35, 36, 43, 45, 46, 47, 50, 51], network performance [6, 15, 25, 32, 38, 41, 39, 49, 57], and WWW behavior [1, 2, 3, 7, 8, 11, 12, 13, 16, 29, 33, 55]. These traces have had a significant effect on system design, resulting directly in improved file-system designs [44], network performance models [12, 25], and Web caching methods [8, 24, 55]

Many researchers have chosen to make their traces available to others because the collection of trace data is difficult, time-consuming, and resource-expensive. Such trace sharing allows follow-on studies as well as enabling independent verification of a study's conclusions. However, sharing trace data also introduces significant privacy problems, as described by Blaze [9]:

> As in any research where data are collected about the behavior of human subjects, the privacy of the individuals observed is a concern. Although the contents of files are not logged by the toolkit, it is still possible to learn something about individual users from examining what files they read and write. At a minimum, the users of a monitored system should be informed of the nature of the trace and the uses to which it will be put. In some cases, it may be necessary to disable the name translation from `nfstrace` when the data are being provided to others. Commercial sites where filenames might reveal something about proprietary projects can be particularly sensitive to such concerns.

Because of the privacy issues, many researchers have chosen to keep their trace data private rather than risk violating the rights of the traced users or exposing valuable competitive information. Others, especially before the expansion of the Internet, have chosen to publish the data anyway, assuming that serious researchers will respect their users or will simply not be interested in analyzing individual behavior on an identifiable personal level. Still others have attempted to find a compromise by anonymizing their trace data to hide sensitive information while still making the scientific data available to the world.

In this paper, we present two methods for anonymizing Web and filesystem traces that avoid privacy risks while still preserving nearly all of the information needed by systems researchers. Our methods are efficient and secure, yet also provide researchers with maximum information from the traced system.

2 Related Work

Until recently, little has been published specifically on the subject of trace anonymization. Rather, most discussions are limited to a few passing sentences, or are incidental asides to the primary thrust of the paper. Nonetheless, it is useful to study what previous researchers have done to balance trace usefulness with the need for privacy.

File system tracing, like CPU tracing, has a very long history. The results of the earliest file system tracing projects [36] were available only on tape, if at all. This approach limited the distribution of the traces, ensuring privacy by obscurity. With the advent of the Internet, however, it has become common to make traces available when a study is published. As mentioned above, many researchers who distributed traces have chosen to do so in their original form with full pathnames included [23, 50].

However, some traces inherently provide anonymity. For example, the Auspex traces [14] included only hexadecimal file identifiers rather than full pathnames. Although back-mapping was theoretically possible

[9], the published Auspex traces apparently did not include enough information to enable such deductions. Similarly, the Sprite traces [5] included per-file unique identifiers (i-node numbers) rather than pathnames.

With the advent of the Web, researchers have begun to address the privacy issues inherent in publishing traces. Much of this work has been done at the IP-address level. For example, Paxson [40] provided simple scripts that replace IP addresses by sequential numbers. A drawback of this method is that network information is lost, so that it becomes difficult or impossible to identify closely related sites such as `www.ebay.com` and `cgi3.ebay.com`. Peuhkuri [42] used MD5 hashing with padding by a secret key (a trace compression technique was also folded into the anonymizer). Peuhkuri also preserved the network portion of the IP address so that individual networks could be identified (this approach could potentially be a problem for lightly populated Class C networks).

Minshall [31] approached the problem of network preservation in a different manner. The `tcpdpriv` anonymizer uses a transformation that guarantees that any two addresses that match in the first *k* bits will be anonymized to values that also match in the first *k* bits, while the remaining bits are randomized.[1] Minshall's method is described only by the source code, but Xu [56, Section 3.1] explains the algorithm quite succinctly. Xu also proposes a significant improvement to the algorithm, which eliminates the need for large tables, uses a cryptographically secure hash, and allows parallel and distributed anonymization.

Traces that incorporate more extensive information, such as URLs or file pathnames, present a different set of challenges. Gibson [22] "scrambled" pathnames using a variant of the MD5 one-way hashing algorithm. Each individual component of the pathname was hashed to produce an 11-character result [21], leaving the total number of components unchanged.

Douceur and Bolosky [18] chose a middle ground in releasing their detailed investigation of file sizes at Microsoft [17]. Each directory component was individually hashed using a keyed one-way hash similar to that described in Section 4.2.2. The same one-way hash was applied to the file name. The 3-character file extension was encoded separately using a technique similar to sequential numbering. To prevent excessive opacity, two small translation tables were then provided along with the traces, one giving the hashed encoding for 8 common directory names and the other giving the encoding for the 1000 most common extensions. This approach allowed other researchers to study some characteristics while maintaining a certain level of privacy.

Wolman *et al.* [54] took perhaps the most aggressive approach in their Web study. URLs were passed through MD5 *in toto*. MD5 was not used for IP addresses, however, since those could easily be cracked with an exhaustive search. Instead, four non-algorithmic lookup tables encoded each octet separately. To provide further protection, several bits of the IP address were deliberately destroyed before encoding so that even if a reverse coding were discovered, it would be impossible to associate a particular behavior with a specific IP address. Wolman's traces are not publicly available; these precautions were taken internally as part of the original trace collection so that maximum privacy could be ensured [53].

Anonymization is primarily an issue for traces that contain readily decipherable information, such as IP addresses, file names, URLs, or commands. In theory, other traces, such as memory access streams or the instructions executed by a CPU, could be used to reconstruct the data streams (including human-readable names and paths) seen by executing programs. However, anonymization is much less of a concern for such traces for several reasons. First, such reconstruction is quite difficult; merely simulating the trace may not be sufficient without the source code for the program. Second, obscuring the instructions run and values transferred may make the trace unusable because the decisions made by the CPU will be different if the data is scrambled. Third, these types of traces typically are short in human time scales, covering seconds

[1] Potential users of `tcpdpriv` should note that there is a bug in the random-number generation if the symbol `SVR4` is defined, such that there are only about 27.5 bits of randomness rather than the intended 32.

or minutes of CPU time. The number of human actions in this time is small, so even a very large trace will only reveal a few human actions.

3 Difficulties with Existing Approaches

The privacy problems with unanonymized publication of traces are clear, and will not be discussed further here. However, anonymization has its own difficulties and drawbacks, as will be seen.

3.1 Information Lossage

As seen in Section 2, most researchers who have concerned themselves with privacy have tolerated significant loss of information as part of the process. The most common approach is to simply replace an entire URL with some sort of encoded equivalent. Although this provides almost complete privacy for the traced users, it also creates difficulties for subsequent researchers. Files or URLs can be uniquely distinguished, but further analysis becomes impossible. Our previous work with filesystem traces has shown that details of pathnames can be useful to the investigator; for example, we have at times found it useful to separately characterize mail files [28] or temporary files [27]. Baker [4] has commented:

> Pathnames were not available for the Sprite traces, and I don't believe they were available for the BSD ones either. This proved to be a mistake for me, since I noticed behavior afterwards that I wanted to correlate if possible with file type. We had inumbers, so I was able to figure out some of it, but not much of it.

Gibson [20, 22] points out another problem with tracing based on unique file identifiers such as inodes: many programs such as `emacs` modify files by reading the original file, modifying it, writing back a new file with a new unique identifier, and then deleting the original file. From the user's point of view, only the file's modification date has changed, though the file system has actually processed both a file creation and deletion. With traditional unique-identifier-based tracing, the link between the old and new file would be not be available, but filename tracing would allow the relationship to be recorded.

Recording full pathnames may also be useful for long-term file-migration studies, as discussed by Miller [30] and Strange [48]. One of the studies [48] reported that strange behavior in the trace was the "result of various builds, including a recompilation of `emacs`." This study further noted that "the large spikes of activity on user disks are often the result of a user copying many files to or from a different disk or to tape." This observation would not be possible on a trace where files were identified solely by a unique numeric value.

In Web tracing, knowing the URL is often critical to understanding the trace. URLs can encode such information as whether content is static or dynamic, whether the content has been customized to the particular user, the language used to create or generate the page, or where the page resides in a subtree. Semantic analysis can reveal even more information, much of which is useful in system design.

3.2 Failures of Anonymization

We have also found that some existing approaches to trace anonymization provide less privacy than might first be assumed. In particular, unpadded algorithmic anonymization of individual components can be susceptible to both exhaustive search (since many components tend to be short) and informed guessing. In a Web trace, computing the hash of common strings such as "`cgi-bin`," "`htm`," and "`index`" will allow an adversary to identify critical directories and important classes of files. In a file trace, values such as "bin," "root," and "Makefile" can be similarly used to identify home and source directories. Armed with this sort of information, semi-exhaustive searches can look for common patterns or subpatterns such as ".c" or "pw=." Since a decoded component can be applied throughout the trace, such techniques can be leveraged to uncover considerably more details than might originally be expected. A wise explorer would do well to MD5-hash the names of all well-known system executables, popular Web sites, and Usenet news groups (both directly and under common transformations such as MIME encoding) as a starting point for discovering information thought to have been carefully hidden.

Other information such as page and file sizes (which are usually recorded in traces) and access patterns can be associated with partially decoded pathnames to aid in the deanonymization process. For example, a regularly accessed file in a trace might indicate a cron job, while one accessed only in the morning might be ".profile." In a Web trace, a moderately large file might be worth an exhaustive search of 5-character names followed by the suffixes ".gif" and ".jpeg", while a very large file might similarly deserve a search for ".mp3" preceded by the names of popular songs. Some pages can be uniquely or nearly uniquely identified by a combination of pathname and size; for example, Google's home page is usually accessed with a null pathname (http://www.google.com/) and has a size that is stable over long periods. An attacker might also use an Internet archive such as The Wayback Machine [52] to correlate page sizes with specific times and develop an attack.

3.3 Undefendable Attacks

Some attacks will remain possible unless nearly all of the information in a trace is destroyed. For example, the idea of using page sizes as hints in an attack leads naturally to an extension, using the sizes as fingerprint. Most nontrivial static Web sites contain a combination of page sizes that are unique to that location. If a moderately large percentage of such a site appears in the trace, it will be possible to identify the site from the combination of sizes alone. Once the site is known, it becomes possible to correlate trace records with individual pages, which in turn may allow some common strings (such as "login.html" at the root of a site) to be identified even if the pathname portion of the URL is anonymized *in toto* using a one-way technique such as sequential numbering.

Peuhkuri [42] suggests an interesting chosen-plaintext attack that will work against almost any anonymization method. The idea is that during trace collection, the attacker inserts accesses that will later be identifiable through timing, size, or any other information that is preserved in the trace. Later, the trace can be examined and the chosen plaintext can be correlated with its anonymized counterpart; this information can then be used to deanonymize other trace records. In network traces, the attack can be enhanced by forging source IP addresses so that the chosen plaintext appears to come from a victim site; this attack will succeed even if a different anonymization encoding is used for each source IP.

We believe that there is no practical defense against these attacks, except for choosing not release traces or to include only the most basic information (such as timestamp and packet size).

4 A Flexible Approach to Anonymizing Strings

For most purposes, we believe that Xu's approach [56] represents the best compromise between information preservation and privacy for IP addresses. However, because of the drawbacks associated with existing string anonymization methods, we have developed a new approach. Our method seeks to find a new balance between the privacy of users and the needs of researchers. To do this, we have placed fine-grained control of privacy into the researcher's or even the users' hands. In addition, our approach can use either of two encoding methods (sequential numbering and secret-key MD5 hashing), both of which are simple and efficient yet resistant to most attacks including exhaustive search.

Our approach was designed with the following goals:

- Anonymization should preserve as much information as possible. In particular, it should be possible to analyze an anonymized trace to discover important non-private information, such as which URLs were static or which files contained source code.

- When possible, anonymization should be under the control of the researcher, the system administrator, and the individual user. When users participate in the trace collection, a user should be able to hide certain strings in a simple, convenient, and foolproof manner.

- Anonymization should be irreversible. It should not be possible to recover original names by processing anonymized ones unless the attacker has access to information not contained in the traces. In particular, exhaustive search should not be able to produce even partial information about anonymized pathnames.

- Anonymization should not unnecessarily increase the size of trace files. Traces are large enough on their own without additional expansion due to artifacts such as changing a 3-character path component into a much longer encrypted version.

- When appropriate, it should be possible to apply string-based anonymization to DNS names, rather than converting them to IP addresses and then using Xu's method.

The reader will note that efficiency is not listed as a goal of our design. We feel that because anonymization is usually performed only once on a trace, it is acceptable if the process is somewhat inefficient. The only caveat is that if trace collection is continuous, anonymization must be able to keep up with the average rate of arrival of new records without reducing the rate of service to users.

4.1 General Approach

Our anonymizer operates on a single string (URL or pathname) at a time. When a string is to be anonymized, a bit mask, equal in length to the string, is created. The mask records which parts of the string should be anonymized. Initially, it is set to all ones, indicating complete anonymization.

The string is then matched, in order, against a series of regular expressions. If a particular expression matches, the bits corresponding to the matched portions of the pathname are either set or cleared, depending on the expression. For example, in a file trace, "`/usr/.*`" might cause bits to be cleared, indicating that filenames in the `/usr` tree should not be anonymized, but a later expression `/usr/games/.*` could nevertheless cause the names of games to be hidden.[2] In a Web trace, "`[^/]*$`" could be used to force the final component of a URL to be passed in the clear; "`[?&]pw=[^&]*`" could subsequently be used to ensure that embedded passwords were still anonymized. The pattern "`[A-Za-z0-9]*`" is often used as a final step to ensure that query elements, pathname components, extensions, etc. may be identified as such.[3]

[2] Of course, if this were the only hidden directory in `/usr`, it would not be difficult to figure out that *some* game was being played.

[3] Because this choice is so common, a command-line switch offers it as an optimized path that avoids the cost of regular-expression matching.

After all patterns have been processed, the bit mask will indicate which characters of the string need to be anonymized. Each contiguous substring is then separately encoded and inserted into its proper place among the unanonymized characters.

4.2 Encoding Methods

Once substrings have been been identified for anonymization, they are encoded to prevent anyone from recovering the original values from the trace. We have investigated two different methods for doing this: sequential numbering of unique pathname components, and keyed MD5 hashes of pathname components. The first method is somewhat faster but requires a large lookup table; the second is more memory-efficient and can dynamically anonymize traces collected across multiple machines or during different time periods.

To increase the portability and generality of our anonymizer, we have chosen an ASCII encoding format even though binary would be more space-efficient. Text formats are amenable to processing by a wide variety of general-purpose tools, and can be easily compressed or converted to a binary format if necessary. However, none of our techniques are inherently restricted to text encoding. Since the extension to binary is straightforward, we will not consider it further in this paper.

4.2.1 Sequential Component Numbering

The sequential-numbering method rests on a simple observation: anonymized components appear in a certain order that is dictated by the format and content of the trace file. Assuming that a particular component is always encrypted to the same output string, an attacker can post-process any trace, no matter how complex the encryption method, to replace the first-encountered component with the number “1”, the second with “2”, and so forth. Thus, it is sufficient to use the same replacement methodology in our own encoding method, since doing so only reveals information that could be reconstructed in any case.

To avoid collisions with strings that might appear naturally, the numbers are encoded in base 64 and further delimited with a user-specified character (“|” by default), chosen to be unlikely or impossible to find in actual trace data.

This encoding system is implemented by looking up a component in a hash table. Each component is handled separately, so that (for example) “`cgi`” will be encoded the same way regardless of where it appears in a string. The system doing the tracing need only keep a hash table translating components into integers. We discuss the memory requirements for such a table in Section 5.

4.2.2 Keyed MD5 Hashes

Using a lookup table to generate encoded components works well, but has two drawbacks. First, it requires a large amount of memory, which may be a scarce resource on a server. Second, it is difficult to “synchronize” traces on different machines, or even on the same machine after a restart, because the lookup table determines the encoding of a particular component. Thus, a request into a cluster of proxy caches might be encoded differently depending on which cache it was sent to, making it difficult to study the overall stream of requests to the proxy cache cluster.

Keyed MD5 hashing [42] provides an effective solution to these drawbacks. Rather than keeping a lookup table converting components into integers, we simply append a short string constant to each component, hash it using a secure hash algorithm such as MD5, and output the result. As long as the constant is kept secret, there is no way to recover the original component. Hashed message authentication codes

(HMACs) [26] use a similar mechanism to provide integrity for messages transmitted over insecure networks.

This approach to encoding components allows traces to be gathered in separate locations (or at separate times) with a minimum of overhead—all that need be distributed is a single, relatively short string. A 120-bit key can be encoded into 20 characters, which is sufficient to protect privacy against exhaustive search. As long as this key is not distributed, it will be impossible to recover the original component names. Moreover, the key can be destroyed as a matter of routine *before* releasing the traces to the public. Because it is small, it need never be recorded in a computer file at all, and could instead be an argument to the trace capture program. A significant advantage of keyed hashing is that the anonymizer need not keep a lookup table of all component values, dramatically reducing memory usage.

One potential drawback to recording full secure hashes is expansion of the trace length. Even if the hashes are stored in binary, a 5-character component will be expanded from 40 to 128 bits by the MD5 algorithm, a problem that will be exacerbated if traces are kept in a more easily processed ASCII format. To address the size problem, we reduce the size of the secure hash output to the trace. This does not affect its security, but does increase the chance that two distinct components will result in the same value being output. The chance that k randomly-chosen sequences of n bits will all be unique is $e^{-k(k-1)/2^n} \approx e^{-k^2/2^n}$ [34]. Since $1-e^{-x} \approx x$ for small x, the chance of a hashed component name collision is $1-e^{-k^2/2^n} \approx k^2/2^n$. If we assume 2^{24} (16 million) unique components and fold the output of MD5 down to 64 bits, the probability of confusing exactly two components in the entire trace is 0.0015% (2^{-16}); the probability of multiple collisions is only slightly higher. Moreover, to significantly affect the analysis of the trace, both of the colliding components would have to be important to the analysis and appear frequently.

Reducing the hash length to 64 bits, however, still results in an expansion for short components; for example, an ASCII encoding similar to `base64` that records 6 bits per character will result in 11-character hashes. To combat this problem, we tied the hash length to the length of the original component (see Table 1). This approach minimized the increase in trace length while preserving anonymity, as shown in Section 5.

4.3 Anonymization Control

A difficult question in designing a anonymizer is the issue of exactly how much information to hide. Previous researchers have taken an all-or-nothing approach. The former method, while admirably protecting privacy, often obscures information that would be valuable to the researcher but harmless to the traced user. The latter, as discussed previously, is unacceptable in a modern computing environment.

Our anonymizer compromises between these two approaches by allowing fine-grained control of what is hidden and revealed. The intention is to maximize the data available to the researcher while minimizing the risk to the user. To this end, a control file defines the regular expressions used for anonymization. As described in Section 4.1, the default is to anonymize the entire pathname, erring on the side of privacy if no control file is given.

The control file is comprised of a number of one-line commands, each containing a keyword and a regular expression. Comments are indicated by the usual pound sign. The keyword is either `pass` or `clean`, indicating that the pattern matches information that is to be passed through unchanged or anonymized, respectively. Parenthesized subexpressions allow the anonymization change to be limited to a subset of the matched string.

Figure 1 shows a simple control file appropriate for maintaining maximal privacy in URLs. Only the most innocuous information is left in the clear; most web-page names and all query arguments are protected. However, component-count information such as the depth of a particular page is preserved. This example

```
# Pass only some separators, common
# suffixes, and index pages.
pass \(\(index|main\)\.html?\)$
pass \(\(index|main\)\.html?\)[^a-zA-Z0-9.]
pass \(\.asp|\.html?|\.php3|\.gif\)$
pass \(\.asp|\.html?|\.php3\)[^a-zA-Z0-9.]
pass [/.?#=+&:;'",]
# Clean DNS names completely.
clean ^[a-zA-Z]*://([^/]*)/
```

Figure 1: URL control file that errs on the side of privacy. Note that the IP address is anonymized separately. For brevity, the suffix list is incomplete.

```
# Invert the default of cleaning everything
pass ^.*$
# Hide everything
# under /home/medium
clean /home/\(medium\)$
clean /home/\(medium/.*\)$
```

(a) Control file that provides intermediate privacy.

```
# Invert the default of cleaning everything
pass ^.*$
# Hide the names of saved mail files.
# The home directory name will be
# preserved.
clean /home/relaxed/\(mail|mail/.*\)$
# Hide game playing
clean /usr/games.*|/usr/share/games.*| \
      /var/games.*|/var/lib/games.*
```

(b) Control file that provides very little privacy.

assumes that IP addresses are not anonymized separately. DNS names are anonymized as a unit (including the username and password, if given).

Figure 2(a) shows a medium-level control file appropriate for a user named `medium`, who wants to compromise on privacy by allowing system filenames to appear unchanged while hiding the names of all his own personal files.[4]

Finally, Figure 2(b) shows a control file that allows almost everything to be seen, hiding the names of only a few files that are known to be sensitive.

5 Measurements and Performance

Any method of obscuring sensitive information in traces must meet several performance criteria: sufficiently low resource usage to avoid interfering with normal system operation, production of traces that accurately reflect system behavior without revealing private information, and generation of traces that are not much longer than the corresponding non-anonymized traces.

5.1 Component Frequencies

The number and relative frequencies of components are important factors in determining how well our approaches to anonymization will work. We analyzed the number of unique components in Web traces from the National Laboratory for Applied Network Research (http://www.nlanr.org/) to see how many unique components of various sizes appeared in the trace. We performed this analysis on on the references for an entire collection of traces for a single day (July 5th, 2001). DNS names were included verbatim and treated as strings, rather than being converted to IP addresses.

[4] Since system programs are identified, a thorough investigator would still be able to identify many other files, such as sources, browser bookmarks, window manager startup files, etc.

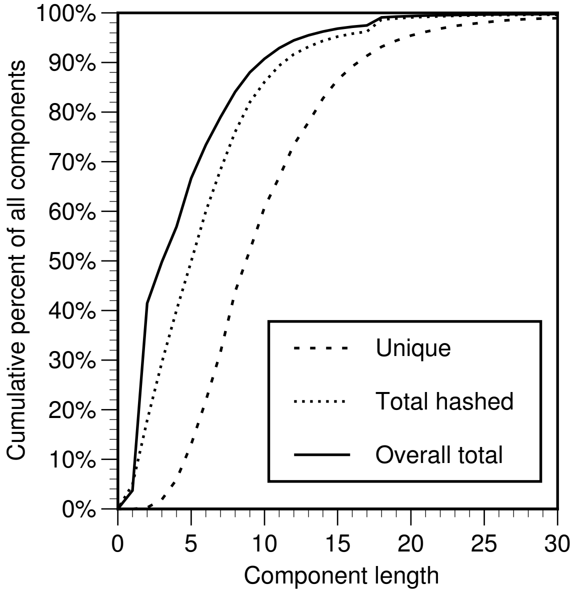

Figure 2: Cumulative component counts (as a percent of total) in our trace of 4.3 million URL references. The "unique" line shows the number of unique components of each length and the "total hashed" line shows the total count of components for each length. These counts do not include the unhashed components listed in Figure 3. The "overall total" line shows the cumulative total count for *all* components, including those shown in Figure 3.

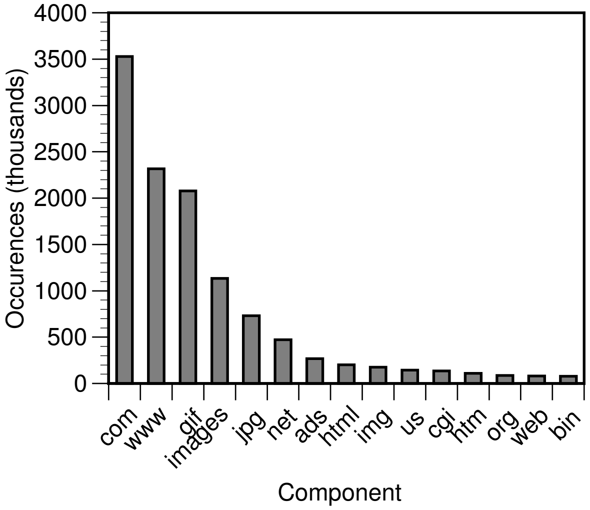

Figure 3: Components not hashed for our anonymized WWW trace. In addition to the components shown, `edu`, `gov`, `shtml`, `jpeg`, and `mil` were unhashed, but they were omitted from the graph because they each made up less than 0.1% of the components in the trace.

As shown in Figure 2, a composite of eight proxy cache traces with over 4.3 million references had 34.7 million components, but only 1.28 million unique components. Moreover, nearly 40% of the components in the trace were included among the 16 listed in Figure 3; these components would likely not be anonymized unless there was a concern over the types of sites (`.gov` vs. `.edu` vs. `.com`, for example) being accessed. There were fewer than 75,000 unique components of four characters or shorter, and no component length had more than 157,000 unique components.

5.2 Anonymization Rate and Memory Requirements

The overall anonymization rate is dominated by three primary factors:

1. The number of unique components M anonymized in each record,
2. The number of regular expressions N in the control files, and
3. The matching efficiency of the regular expressions.

The time required by item 1 is approximately linear in M. In the sequence-number method, this value is dominated by lookup time in our implementation (which uses hash tables). The lookup time is approximately constant for reasonably short component lengths. In the keyed-hashing method, the time per component is dominated by the overhead of the MD5 algorithm and the cost of hashing the appended key, which again are approximately constant.

Since each regular expression is searched for separately, the total anonymization time is linearly proportional to N. For factor 3, we have found that most of the regular expressions used in practice are linear in the length of the matched string, so that this factor can usually be ignored. However, if maximal speed is important, it is critical to minimize the number of regular expressions and to avoid writing expressions that will cause excessive backtracking.

Thus, the overall complexity of anonymization is $O(MN)$.

To quantify the approximate performance of our anonymization methods, we ran our C++-based anonymizer on the traces discussed in Section 5.1. Only the anonymization method was varied; all other factors remained constant. All timing experiments were performed on a 400-MHz Pentium II processor with 256 MB of memory. All anonymization output used a base-64 ASCII output encoding for portability; we expect that binary output would be faster by an approximately constant factor. All of our test runs made use of the optimized code for separating components at non-alphanumerics.

5.2.1 Sequence Numbers

Anonymizing the composite trace of 4.3 million records to /dev/null using sequence numbers took 537 seconds, achieving a rate of 8121 records/second. The average record contained 10.24 separately anonymized components, so the anonymizer cleaned 83242 components/second.

As expected, adding control patterns had a significant effect on performance. When run with a control file that passed the 21 components listed in Figure 3, using a single regular expression to match them, the sequence-based anonymizer took 633 seconds to clean the trace, a rate of 6898 records/second and 70701 components/second (counting both cleaned and passed components for the purpose of calculating the latter figure). When the 21 components were listed in 21 separate regular expressions, the time jumped over tenfold: 6565 seconds, or 665 records and 6815 components per second. Clearly, it is critical to minimize the number of regular expressions in our implementation.[5]

As discussed above, the lookup table contains one entry for each unique component. In our implementation (which has not been optimized for space), each entry occupies approximately 32 bytes plus the length of the component. Since most components are short (see Figure 3), few entries occupy more than 40 bytes, so the entire table can be stored in 50-100 MB, depending on the load factor chosen. Note that the table size is primarily affected by the number of unique components in this large Web trace. A trace with fewer unique strings, such as a typical filesystem trace, would require correspondingly less memory. When run on the composite trace, the hash-based anonymizer grew to a maximum size of 95.2 MB.

5.2.2 Keyed Hashing

Keyed hashing was substantially slower; anonymizing the same trace with no patterns file took 778 seconds, for a rate of 5612 records/second or 57521 components/second. However, as expected, the data size was vastly smaller than for sequence numbers; the maximum process size during the same test was only 1.9 MB.

It is worth noting that even a fast Web server [37] can handle only about 1040 references per second on the 400-MHz processor we used for our tests.[6] Thus, anonymizing with keyed hashing would consume less than 19% of the CPU on a dedicated Web server; with appropriate priority settings, traces can be buffered during load spikes so that cost of anonymizing will not affect the peak performance of the server. Alternatively, a single dedicated machine could anonymize the traces generated by 5 servers. Some Web servers, such as Apache, can pipe their trace output directly to a program; others can be set up to write their logs to a named pipe, making it possible to anonymize on the fly without ever recording unanonymized information.

5.3 Reducing Trace Size

Some previous approaches to anonymization [18, 22] could cause increases in the trace size. Given the large storage requirements of tracing, further size increases seem undesirable.

Our study confirms the widely-held belief that there are relatively few unique components even in a large Web trace. We can use this knowledge to anonymize the trace without dramatically increasing the trace size. In this section, we will discuss our two approaches to anonymizing traces without a size increase: secret hashing of components, and the use of a lookup table to translate components into numbers.

5.3.1 Sequence Numbers

Sequentially numbering components is most efficient if the most frequently referenced components receive the lowest numbers, similar to a Huffman code. Creating such an encoding would require multiple passes through the input, which is not practical for online applications. In practice, however, our method still reduced the overall trace size. When run on the composite NLANR trace with full anonymization enabled, the output file was only 72% of the input size. Some of that reduction was due to the fact that IP addresses were anonymized as a unit, cutting 12-15 characters to 4-6. Offsetting that advantage was the fact that all anonymized strings were bounded by a unique character on both sides, so that no anonymized string was

[5] Alternatively, all of the expressions could be compiled into a single state machine. However, the library that we used did not support this option.

[6] This figure was calculated by adjusting the value in Table 1 of [37] for clock rate.

Component length	Hash length	Ratio
1	4	4.0
2	4	2.0
3	6	2.0
4	6	1.5
5	8	1.6
6	8	1.3
7	10	1.4
8	10	1.25
9+	10	< 1.1

Table 1: Hash lengths (in characters) for various sizes of path name components. Hashes pack 6 bits per character using a scheme similar to base64 encoding.

shorter than 3 characters and all but 64 were 4 characters or longer, while many input components are very short.

5.3.2 Keyed Hashing

Hashing each component as described in Section 4.2.2 will prevent trace consumers from figuring out which components correspond to which plaintext words. However, the naive approach to storing the components requires that a large hash be used for each component, increasing the size of the trace.

For example, the 4.3-million-reference Web trace we studied had 34.7 million components, of which 23 million were not among the unanonymized components listed in Figure 3. Hashing each anonymized component into a 60-bit value derived from an MD5 hash would require 10 bytes (characters) per component, for a total of 231 MB. The original components corresponding to this data would only require 150 MB; the hashes would thus expand this portion of the trace by 54%. Reducing the hash size to 54 bits (9 characters) would still expand the components by 39%, and this approach would have a 2^{-14} chance of collision for a one-day Web trace.

To further reduce the size of the trace, we varied the length of the hash value based on the length of the component. This does not significantly weaken the anonymization because there are still far too many combinations of components of fixed length to draw any conclusions based solely on component length. Reducing the length of the hash for shorter components works because there are fewer components of shorter lengths—hardly a surprise because there are limited numbers of shorter components, and many of them make little sense in English. The variable-length hashes we used are summarized in Table 1.

Applying these shorter hash lengths further reduced the size of the components to 176 MB, for an expansion factor of 18% over the unanonymized trace. By switching to 54-bit hashes for 7-character and longer components, we further reduced the anonymized component size to 167 MB, or 112% of the original component length. The expansion ratios of all of the hashing methods we tried are shown in Figure 4.

6 Conclusions

Tracing is an important tool in the arsenal of systems and network researchers. In the past, trace sharing has been limited by the danger of revealing private information. Anonymization, when applied, has tended to be

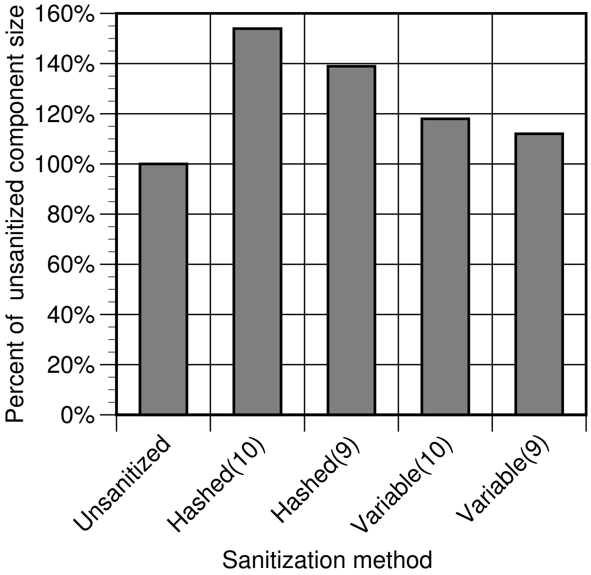

Figure 4: Expansion factors for various hashing approaches over the unanonymized components. These factors are worst-case values, and exclude portions of the trace not anonymized such as component separators, timing information, and operation codes (`GET`, `POST`, etc.).

ad-hoc, has sometimes failed to achieve its goals, and has often introduced undesirable characteristics into the traced data.

We have presented a new and flexible approach that allows anonymization to be limited to sensitive component of the trace. We have also presented two anonymization methods that can guarantee anonymity and privacy without excessive growth in the trace size or unacceptable CPU demands. One of the methods runs faster and produces smaller traces at the expense of requiring a large database on a single system, while the other produces larger traces but can be used in a distributed environment. Both methods are suitable for general use.

Software Availability

The software used in this project is available for download at:
`http://ssrc.cse.ucsc.edu/software.shtml`.

Acknowledgments

Some of this research was supported by the Defense Advanced Research Projects Agency under contract number N00174-91-C-0107. We would like to thank the National Laboratory for Applied Network Research (supported by the National Science Foundation under grants NCR-9616602 and NCR-9521745) and Tim Gibson for allowing us to use the traces they gathered in our study.

References

[1] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching proxies: Limitations and potentials. In *Proceedings of the Fourth International World Wide Web Conference*, Dec. 1995.

[2] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, Dec. 1996.

[3] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *ACM SIGMETRICS Conference Proceedings*, pages 126–137. ACM, May 1996.

[4] M. Baker, 2000. Personal communication.

[5] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Sherriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 198–211. ACM, Oct. 1991.

[6] H. Balakrishnan, S. Seshan, V. Padmanabhan, M. Stemm, and R. H. Katz. TCP behavior of a busy internet server: Analysis and improvements. Technical Report 97-966, University of California, Berkeley, Aug. 1997.

[7] H. Balakrishnan, S. Seshan, M. Stemm, and R. H. Katz. Analyzing stability in wide-area network performance. In *ACM SIGMETRICS Conference Proceedings*, Seattle WA, USA, June 1997. ACM.

[8] P. Barford, A. Bestavros, A. Bradley, and M. E. Crovella. Changes in web client access patterns: Characteristics and caching implications. *World-Wide Web Journal*, 2:15–28, 1999.

[9] M. Blaze. NFS tracing by passive network monitoring. In *USENIX Conference Proceedings*, pages 333–343, San Francisco, CA, Jan. 1992. USENIX.

[10] G. P. Bozman, H. H. Ghannad, and E. D. Weinberger. A trace-driven study of CMS file references. *IBM Journal of Research and Development*, 35(5–6):815–828, Sept.–Nov. 1991.

[11] H.-W. Braun and K. Claffy. Web traffic characterization: an assessment of the impact of caching documents from ncsa's web server. In *Proceedings of the Second International World Wide Web Conference*, Oct. 1994.

[12] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *ACM/IEEE Transactions on Networking*, 5(6):835–846, Dec. 1997.

[13] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW client-based traces. Technical Report 95-010, Boston University, Apr. 1995.

[14] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280. USENIX, Nov. 1994.

[15] P. Danzig, S. Jamin, R. Caceres, D. Mitzel, and D. Estrin. An empirical workload model for driving wide-area TCP/IP network simulations. *Journal of Internetworking: Research and Experience*, 3(1):1–26, Mar. 1992.

[16] C. Dharap and M. Bowman. A facility for tracing wide-area information access. Technical report, Pennsylvania State University, Department of Computer Science, University Park, Pennsylvania, Nov. 1994.

[17] J. Douceur and W. Bolosky. A large-scale study of file-system contents. In *ACM SIGMETRICS Conference Proceedings*, Atlanta, Georgia, USA, May 1999. ACM.

[18] J. R. Douceur and W. J. Bolosky. Sanitized data set from “A large-scale study of file-system contents”. 6 CD-ROMs published by Microsoft Research, Redmond, WA, 1999. Contact `johndo@microsoft.com` or `bolosky@microsoft.com` for availability.

[19] R. A. Floyd and C. S. Ellis. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):238–247, June 1989.

[20] T. J. Gibson. *Long-term Unix File System Activity aand the Efficacy of Automatic File Migration*. PhD thesis, University of Maryland, Baltimore County, May 1998.

[21] T. J. Gibson, 2000. Personal communication.

[22] T. J. Gibson, E. L. Miller, and D. D. E. Long. Long-term file activity and inter-reference patterns. In *Proceedings of the 24th International Conference on Technology Management and Performance Evaluation of Enterprise-Wide Information Systems*, pages 976–987, Anaheim, CA, Dec. 1998. Computer Measurement Group.

[23] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the Summer USENIX Conference Proceedings*, Boston, MA, June 1994. USENIX. Also available as University of Kentucky Technical Report CS247-94.

[24] J. Gwertzman and M. Seltzer. World Wide Web cache consistency. In *USENIX Conference Proceedings*, pages 141–152. USENIX, Jan. 1996.

[25] J. Heidemann, K. Obraczka, and J. Touch. Modeling the performance of HTTP over several transport protocols. *ACM/IEEE Transactions on Networking*, 5(5):616–630, Oct. 1997.

[26] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, Internet Request For Comments, Feb. 1997.

[27] G. H. Kuenning. *Seer: Predictive File Hoarding for Disconnected Mobile Operation*. PhD thesis, University of California, Los Angeles, Los Angeles, CA, May 1997. Also available as UCLA CSD Technical Report UCLA-CSD-970015.

[28] G. H. Kuenning, G. J. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. In *USENIX Conference Proceedings*, pages 291–306. USENIX, June 1994.

[29] T. T. Kwan, R. E. McGrath, and D. A. Reed. User access patterns to NCSA’s world wide web server. from web, 1995.

[30] E. L. Miller and R. H. Katz. An analysis of file migration in a UNIX supercomputing environment. In *USENIX Conference Proceedings*, pages 421–433. USENIX, Jan. 1993.

[31] G. Minshall. Tcpdpriv. http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html, Aug. 1997.

[32] J. C. Mogul. Observing TCP dynamics in real networks. Technical Report 92.2, DEC Western Research Laboratory, Apr. 1992.

[33] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report 95/5, DEC Western Research Laboratory, Oct. 1995.

[34] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[35] L. B. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. Technical Report CMU-CS-94-213, Carnegie-Mellon UniversitySchool of Computer Science, Pittsburgh, PA, Nov. 1994.

[36] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 15–24. ACM, Dec. 1985.

[37] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, Feb. 2000. Available online at http://www.acm.org/pubs/citations/journals/tocs/2000-18-1/p37-pai/.

[38] V. Paxson. Growth trends in wide-area TCP connections. *IEEE Network Magazine*, 8(4):8–17, July 1994.

[39] V. Paxson. End-to-end internet packet dynamics. *ACM/IEEE Transactions on Networking*, 5(5):601–615, Oct. 1997. (Revised version of SIGCOMM paper).

[40] V. Paxson. Scripts for sanitizing TCPDUMP trace files. http://ita.ee.lbl.gov/html/contrib/sanitize.html, 1997.

[41] V. Paxson and S. Floyd. Wide-area traffic: The failure of Poisson modeling. *ACM/IEEE Transactions on Networking*, 3(3):226–244, Aug. 1995. An earlier version of this paper appeared in SIGCOMM 94, pp. 257-268, August 1994.

[42] M. Peuhkuri. A method to compress and anonymize packet traces. In *Proceedings of the First ACM Internet Measurement Workshop*, pages 257–261, San Francisco, CA, Nov. 2001. ACM, ACM.

[43] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *USENIX Conference Proceedings*, San Diego, CA, June 2000. USENIX.

[44] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), Feb. 1992.

[45] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Proceedings of the Winter Usenix*, pages 405–420. USENIX, Jan. 1993.

[46] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, Dec. 1981.

[47] K. W. Shirriff and J. K. Ousterhout. A trace-driven analysis of name and attribute caching in a distributed system. In *USENIX Conference Proceedings*, pages 315–331. USENIX, Jan. 1992.

[48] S. Strange. Analysis of long-term UNIX file access patterns for application to automatic file migration strategies. Technical Report UCB/CSD 92/700, University of California, Berkeley, Aug. 1992.

[49] K. Thompson, G. J. Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics (extended version). *IEEE Network Magazine*, 11(6):10–23, Nov/Dec 1997.

[50] M. Uysal, A. Acharya, and J. Saltz. Requirements of I/O systems for parallel machines: An application-driven study. Technical Report CS-TR-3802, University of Maryland, College Park, MD, May 1997.

[51] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th Symposium on Operating Systems Principles*, Kiawah Island, SC, USA, Dec. 1999. ACM.

[52] The Wayback Machine. http://www.archive.org.

[53] A. Wolman, 2000. Personal communication.

[54] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of web-object sharing and caching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, Oct. 1999. USENIX.

[55] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the Seventeenth Symposium on Operating Systems Principles*, pages 16–31, Kiawah Island, South Carolina, Dec. 1999. ACM.

[56] J. Xu, J. Fan, M. Ammar, and S. B. Moon. On the design and performance of prefix-preserving IP traffic trace anonymization. In *Proceedings of the First ACM Internet Measurement Workshop*, pages 263–266, San Francisco, CA, Nov. 2001. ACM, ACM.

[57] M. Yajnik, J. Kurose, and D. Towsley. Packet loss correlation in the MBone multicast network. In *Proceedings of the IEEE Global Internet*, London, U.K., Nov. 1996. IEEE.