A High-level Programming Environment for Packet Trace Anonymization and Transformation

Ruoming Pang
Department of Computer Science
Princeton University
rpang@cs.princeton.edu

Vern Paxson
International Computer Science Institute
vern@icir.org

ABSTRACT

Packet traces of operational Internet traffic are invaluable to network research, but public sharing of such traces is severely limited by the need to first remove all sensitive information. Current trace anonymization technology leaves only the packet headers intact, completely stripping the contents; to our knowledge, there are no publicly available traces of any significant size that contain packet payloads. We describe a new approach to transform and anonymize packet traces. Our tool provides high-level language support for packet transformation, allowing the user to write short policy scripts to express sophisticated trace transformations. The resulting scripts can anonymize both packet headers and payloads, and can perform application-level transformations such as editing HTTP or SMTP headers, replacing the content of Web items with MD5 hashes, or altering filenames or reply codes that match given patterns. We discuss the critical issue of verifying that anonymizations are both correctly applied and correctly specified, and experiences with anonymizing FTP traces from the Lawrence Berkeley National Laboratory for public release.

Categories and Subject Descriptors: C.2.m [Computer-Commnication Networks]: Miscellaneous—packet trace processing; D.3.4 [Programming Languages]: Processors—network trace rewriting

General Terms: Security, Measurement

Keywords: packet trace, anonymization, transformation, Internet, privacy, measurement, network intrusion detection

1. INTRODUCTION

Researchers often use tools such as `tcpdump` to capture network packet traces. Packet traces recording real-world Internet traffic are especially useful for research on traffic dynamics, protocol analysis, workload characterization, and network intrusion detection. However, sharing of Internet packet traces is very limited because real-world traces contain many kinds of sensitive information, such as host addresses, emails, personal web-pages, and even authentication keys. The traces must be first "anonymized" to eliminate any private information (e.g,, IP addresses, user IDs, passwords) before they can be shared among researchers.

To date, Internet packet trace anonymization has been limited to only retaining TCP/IP headers [21, 17], with IP addresses renumbered and packet payloads completely removed. To our knowledge, there are *no* publicly available traces of any significant size that contain TCP payloads. The lack of such traces greatly limits research on application protocols. It is especially crippling for network intrusion detection research, forcing researchers to devise synthetic attack traces that often lack the verisimilitude of actual traffic in critical ways, resulting in errors such as grossly underestimating the false positive rate of "anomaly detection" techniques. [10, 1]

In this work we develop a new method to allow anonymization of packet payloads as well as headers. Traces are processed in three steps:

1. Payloads are reassembled and parsed to generate application-protocol-level, semantically meaningful data elements.

2. A policy script transforms data elements to remove sensitive information and sends the resulting elements to the composer.

3. The trace composer converts application protocol data elements back to byte sequences and frames the bytes into packets, matching the new packets to the originals as much as possible, in order to preserve the transport protocol dynamics.

Parsing allows the trace transformation policy script to operate on semantically meaningful data elements, such as usernames, passwords, or filenames, making policy scripts more concise and comprehensible than those operating directly on packets or byte sequences. Working at a semantic level also gives the opportunity for less draconian anonymization policies. For example, the added information that the string "`root`" appears in a filename ("`/root/.cshrc`") rather than as a username might, depending on a site's anonymization policy, allow the string to appear in an anonymized trace, whereas a purely textual anonymization would have to excise it, because it could not safely verify that the occurrence did not reflect a username.

The design of trace composer aims to generate "correct" traces, for instance, as payload data is modified, checksums, sequence numbers, and acknowledgments will be accordingly adjusted. The output traces just look as if they were collected from the real Internet, except that they do not carry private information. Accordingly, analysis tools that work on raw traces will likewise work on the anonymized traces.

In order to make the anonymization process amenable to validation, we follow a "filter-in" principle throughout our design of the anonymizer: instead of focusing on "filtering out" sensitive information, the anonymizer focuses on what, explicitly, to *retain* (or insert, in a modified form) in the output trace. With this principle, it becomes much easier to examine a policy script for privacy holes.

An optional "manual inspection" phase can keep more nonsensitive information in the output trace as the general anonymization script may have to make conservative judgments for some data elements; for example, whether to allow the command "UUSER" to appear in a trace of anonymous FTP traffic (the presence of such a typo can be useful for some forms of analysis, such as anomaly detection).

We implemented the anonymizer as an extension to Bro [16], a network intrusion detection system, to take advantage of its application parsers and its built-in language support for policy scripts.

Beside anonymization, our tool can also be used for generic trace transformations, providing a great degree of freedom and convenience for various types transformation. For example, we can take a trace of FTP traffic and remove from it all the connections for which the user name was not "anonymous"; or all the ones for which the FTP authentication was unsuccessful; or those that do uploads but not downloads. A different type of transformation is for testing network intrusion detection systems by inserting attacks into actual background traffic by slightly altering existing, benign connections present in a trace. Still another type of transformation is to remove large Web items from HTTP connections (including persistent sessions with multiple items) in order to save disk space (see Section 3.4 below).

In a sense, the tool spells the end of traces as being stand-alone evidence of any sort of activity, since it makes it so easy to modify what a trace purports to show.

We developed trace transformations for FTP, SMTP, HTTP, Finger, and Ident. As a test of the approach, we anonymized FTP traces from the Lawrence Berkeley National Laboratory (LBNL). Besides testing the technology, one of the important questions behind the exercise was to explore what sort of anonymizations a site might require, and being willing to abide, for public release of traces with contents. To this end, working with the site we devised an anonymization policy acceptable to the site and approved for public release. The corresponding traces are available from [6].

The rest of this paper is organized as follows. In the next section we present our goals. We describe generic packet trace transformation in Section 3, trace anonymization in Section 4, and unsolved problems and new directions in Section 5. We discuss related work in Section 6 and summarize in Section 7.

2. GOALS

We designed the transformation tool with the following goals in mind:

1. Policy scripts operate on application-protocol-level data values. This means that instead of operating on packets or TCP flows, a policy script sees typed and semantically meaningful values (e.g., HTTP method, URI, and version). Likewise, the trace transformation scripts also specify application-protocol-level data to the output trace, without needing to dictate the details of generating the actual packets.

2. The output traces contain well-formed connections: packets have correct checksums and lengths, TCP flows can be reassembled from the resulting packets, and application-protocol data has correct syntax[1], so that other programs can process the transformed traces in the same way that they handle original `tcpdump` traces.

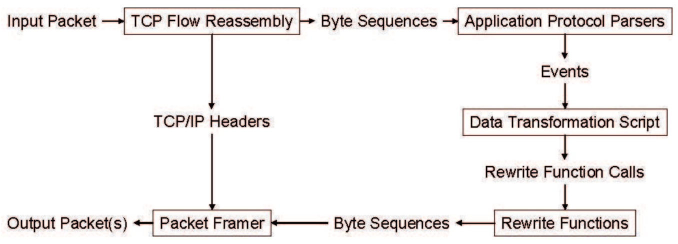

Figure 1: Data Flow in Trace Transformation

3. The mechanism supports generic trace transformations besides anonymization.

4. The anonymization is "fail safe" and amenable to verification. Fail-safety means that the privacy resulting from the anonymization does not depend on the tool and the policy script being completely correct. Being amenable to verification means it is easy to examine and validate the policy script, the anonymization process, and the output trace.

The first and third goal dictate where to separate mechanism and policy: 1) the mechanism part should parse the input trace to *expose* all application-protocol semantic elements, e.g., commands, reply codes, MIME header types; 2) the mechanism should not restrict how the values will be changed, but leave that to the policy script. We will discuss mechanism and anonymization policy in the next two sections, respectively.

3. GENERIC TRACE TRANSFORMATION

Trace transformation consists of three steps: parsing, data transformation, and composition. These are shown as the right-hand components of Figure 1. The parsing and composition parts do not depend on the type of trace transformation, and we have implemented them in Bro as built-in mechanisms. The second step (data transformation) is fully programmable, however, and so is implemented as a Bro policy script.

We will first look at the process from the viewpoint of the policy script, focusing on the trace input/output interface, and then discuss details of trace parsing and composition.

3.1 Policy Script Programming Environment

The Bro policy script language is procedural, with strong typing that includes support for several network-specific types (e.g., addresses and ports), as well as relative and absolute time, aggregate types (associative tables, records), regular expression matching, and string manipulation. More details about Bro language can be found in [16, 15].

From the point of view of a policy script, the parsing part is Bro's event engine, and the composer is a family of library functions, which we call "rewrite functions".

A policy script for a protocol usually contains several "event handlers", which are execution entry points of the script. Through event parameters, each event handler receives protocol-semantic data elements as well as a record corresponding to the particular TCP connection. An event handler may call other functions to process the data, and writes the transformed data to the output trace

[1]Or not, if the policy script decides to keep the "dirtiness" of the original trace.

by calling the rewrite functions. When calling a rewrite function, the policy script specifies a connection, and sometimes also direction of the flow, to write the data to. The destination connection is usually the same connection of the event, but can also be any other connection present in the input trace at the same time.

For example, a line in an SMTP message "`MAIL From:<alice@bob.org>\r\n`" arriving on connection C will generate the event:

```
smtp_request(
    conn: connection = C,
    command: string = "MAIL",
    argument: string = "From: <alice@bob.org>")
```

The policy script receives the command and argument and decides what to write to the output trace—e.g., it could call:

```
rewrite_smtp_request(
    C,
    "MAIL",
    "From: <name123@domain111>")
```

to change the sender in the trace from "`alice@bob.org`" to "`name123@domain111`".

There is usually a correspondence between protocol events and rewrite functions: e.g., for event `smtp_request`, there is function `rewrite_smtp_request`, and they have the same or very similar set of parameters.

Explicit Rewriting: Note that the trace composer API requires explicit rewrites, i.e., for a data element to get into the output trace, it must be explicitly placed there by the policy script calling a rewrite function. Alternatively, another style we could have chosen for the composer API would be to have the policy script only specify data elements that should be *changed*, and pass the rest through unmodified. With this style, we could implement a single generic interface by which scripts would directly specify the element to change. For example, the SMTP rewrite above would be specified as:

```
modify_element(
    smtp_request_arg,
    "From: <name123@domain111>")
```

and the composer would alter the location in output trace occupied by the variable `smtp_request_arg` to contain the new text rather than the original.

While appealing because a single rewrite function would suffice for all protocols (though the application parsers would have to annotate each script variable with its location in the connection's byte stream), instead of having a family of rewrite functions for various protocols, we choose the heavier API because it presents a safer interface for trace anonymization. First, requiring explicit rewrite forces the policy script writer to put consideration into every element, so it will be less likely that they overlook a privacy hole. Second, it is easier for other people to examine a policy script for privacy leaks, as the examiner only needs to look at elements written in the script (rather than having to keep in mind all the protocol elements that are implicitly not being changed because they don't show up in the script). This design choice shows how the "filter-in" principle affects our design. Additionally, this interface allows type-checking on trace-rewrite operations to catch inconsistency between output data elements.

3.2 Trace Parsing

Trace parsing usually consists of three steps: flow reassembling, (optional) line breaking, and protocol-specific parsing.

Flow Reconstruction: Bro's application parsing begins by reassembling IP fragments and then reassembling the TCP byte stream. (We ignore here Bro's UDP processing, though our techniques could be applied to it, too.) In case of TCP retransmission or packet reordering, the bytes that arrive first are not delivered until the gap is filled, at which point the bytes are delivered together. For example, suppose an SMTP command arrives in three packets with the last two in reverse order: "`MAIL Fro`", "`bob.org>\r\n`", and "`m:<alice@`". The reassembler will emit "`MAIL Fro`" on the first packet arrival, nothing on the second because it comes out of order, and "`m:<alice@bob.org>\r\n`" after processing the third packet.

Breaking into Lines: Many protocols (e.g., SMTP, FTP, the non-data part of HTTP) process application data one line at a time. For such protocols, there is an intermediate step that structures the bytes from reassembler into lines before protocol-specific parsing. Following the above example, the line divider will emit a line "`MAIL From: <alice@bob.org>`" after it sees `\r\n`.

Protocol-Specific Parsing: The parser takes plain bytes as input and emits typed and semantically meaningful data fields. It first divides the bytes according to protocol syntax, then converts bytes of each field to typed values—e.g., string, integer, boolean, record—and groups the values by events, finally placing the events in an event queue. (As event parameters, each data element carries a semantic meaning.) Currently Bro has parsers for DNS, Finger, FTP, HTTP, ICMP, Ident, MIME, NTP, Netbios, Rlogin, SMTP, SSH, and Telnet.

A major challenge in parsing is that the parser often cannot strictly follow the RFCs that define the application protocol, since in practice there are frequently deviations from the letter of the standards, or deficiencies in the traffic being analyzed. Two particular difficulties relevant for our discussion are:

Line Delimiters: Line-oriented protocols (e.g., SMTP, HTTP) generally are specified to use the two-byte sequence CRLF (`\r\n`) as the delimiter between lines. However, some end hosts also interpret single LF (`\n`) and/or CR (`\r`) as the end of the line. Ideally, we would like to identify which delimiter each host uses, and consistently apply that interpretation.

Content Gaps: For traces captured under high-volume traffic conditions, sometimes the packet filter fails to capture all of the packets. Such "content gaps" are generally unsolvable, but we found that most of them occur within the data-transfer section of an application dialog rather than in the command/reply exchange. We developed a content gap recovery mechanism for SMTP and HTTP that skips over gaps that appear consistent with being wholly contained within a data transfer. With this heuristic, we find that most content gaps no longer disrupt parsing. (We note that content gaps are also delivered as events, and the policy script may decide to eliminate them, keep them, or even insert new content gaps in the output trace.)

In summary, there can be some loss of fidelity when data goes through the trace parser. This is in fact a general problem for any network monitoring tools.

3.3 Trace Composer

The trace composer consists of rewrite functions and a packet generator. As discussed above, the rewrite functions are called during event processing. A rewrite function generates a byte string on each invocation and buffers the string for the packet generator. After processing events, Bro invokes the packet generator to process buffered bytes and generate output packets. Below we will look at the rewrite functions and packet generation in detail.

```
# Write a finger request to trace.
rewriter finger_request %(full: bool,
                username: string, hostpart: string%)
        %{
        const int is_orig = 1;
        if ( full )
                @WRITE@(is_orig, "/W ");
        @WRITE@(is_orig, username);
        if ( hostpart->Len() > 0 )
                {
                @WRITE@(is_orig, "@");
                @WRITE@(is_orig, hostpart);
                }
        @WRITE@(is_orig, "\r\n");
        %}
```

Figure 2: Source Code of a Rewrite Function

3.3.1 Rewrite Functions

A rewrite function performs the inverse of parsing: it prints the typed data elements to a byte string in a protocol specific format, placing them in the right order and adding proper delimiters. For example, `rewrite_finger_request` takes four parameters: `c` (the associated connection, of type `connection`, which is a record of connection information), `full` (a boolean flag indicating whether the Finger request was for the "full" format), `username` and `hostpart` (both strings). The rewrite function concatenates `username` and `hostpart`, adds `\r\n` to the end, and inserts "`/W `" to the beginning of the line when `full` is true. Thus, with parameters `(T, "alice", "host123")`, the function generates the string "`/W alice@host123\r\n`", and with parameters `(F, "bob", "")`, it generates "`bob\r\n`".

Rewrite Function Compiler: When implementing the rewrite functions for various protocols, we found a number of commonalities: they all need to convert Bro values to C++ native values and fetch the connection object, and for each built-in function we need to write a Bro-language prototype declaration and add initialization code to bind the Bro built-in function to the C++ function. So we looked for ways to facilitate code reuse to avoid the tedious and error-prone task of repeating the similar code at each place.

To do so, we developed a "rewrite function compiler". We write rewrite functions with Bro-style function prototypes and C++ bodies. The compiler inserts code for the value conversion and connection record fetch, extracts Bro function prototypes, and generates function binding code. With the rewriter compiler, most rewrite functions can be implemented with around 10 lines of code each. Figure 2 shows the source code of "`rewrite_finger_request`". Note that each rewrite function has a hidden first parameter: "`c: connection`", which is inserted into the C++ code and the Bro prototype during compilation.[2]

Currently we have implemented rewrite functions for FTP, HTTP, SMTP, Finger, and Ident.

3.3.2 Packet Generation: Framing

After rewriter functions emit byte sequences, the *packet framer* decides how to pack the bytes into packets. It cares about 1) whether the bytes should fit into a single packet or be split across multiple ones, and 2) what timestamp to attach to each packet.

The central concern of the packet framing algorithm is to keep the traffic dynamics as close to the original as possible and yet to remain transparent to the policy scripts. For example, an HTTP request can be transmitted line-by-line, one packet per line, or all in one packet; for each of these cases, we would like the rewritten request to maintain the original packet structure and the timestamps.

Note that we cannot directly reuse the packet structure present in the input trace because there is not necessarily a one-to-one mapping between bytes in the input and output traces, as a policy script can change data lengths, insert or remove objects, or change the ordering among objects. So in general it is only possible to *approximate* the original dynamics. Also, as the policy script does not have to specify the origin of data when it calls a rewrite function, the trace composer does not know an explicit mapping between original and new data objects and has to derive an implicit temporal relation to map bytes to packets, as follows.

In the common case, transformed data is written to the same TCP flow (i.e., same direction of a TCP connection) as the input packet currently being processed. The framer places the bytes in the current output packet. If the payload size exceeds the MTU, it generates another output packet with the same timestamp to hold the rest of the data.

Usually the data written by the policy script originates from data in the current input packet; thus, the output trace has a similar packet structure as the input trace. However, there are two cases in which the data to write actually comes from an earlier or later input packet:

1. When an event consists of data from multiple packets, the data may range across packet boundaries or appear in retransmitted packets. In this case, the transformed data will be written with respect to the last packet associated with the event, i.e., the packet whose arrival makes the trace parser generate the event.

2. When the policy script cannot decide immediately what to write before seeing later data. For example, when rewriting HTTP messages, the new Content-Length header for an HTTP entity cannot be decided until the entity is entirely transformed. In another example, when anonymizing FTP traces, user names in unsuccessful login attempts might be treated differently than user names in successful logins (because the unsuccessful ones can leak sensitive information, such as passwords mistyped for user names), so the script needs to see the server reply before it can decide how to anonymize the argument of the "USER" command.

For the first of these, we find it tolerable to simply associate the data with the event's last packet, because to do otherwise would require a great deal of work—tracing each event parameter's origin throughout the trace reassembly and parsing hierarchy in order to know from exactly which input packet the data originates.

Deferring Writes: The second case, of the policy script having to defer its transformation decision, presents a larger problem, because it not only leads to imprecise timestamps for output packets, but also causes inconvenience for transformation script programming: in the HTTP message case, the Content-Length header has to be written before the data entity, so the script must buffer up all the transformed data entity until it finishes processing the entire entity. To address this problem, we added support for deferring writes so that the script can essentially write packets out of order.

The trace composer supports deferring writes by allowing the policy script to reserve slots in current output packets. The script may then seek the reserved slot at a later point, write data to it, and release the slot. (See Figure 3)

3.3.3 Packet Generation: TCP/IP header fields

[2]The boolean variable "is_orig = 1" means the direction of the TCP flow is from the connection originator (the Finger client).

```
# when the original Content-Length header
# arrives on connection c
msg$header_slot = reserve_rewrite_slot(c);
...
# after the entire data entity is processed
seek_rewrite_slot(c, msg$header_slot);
rewrite_http_header(c, is_orig, "Content-Length",
                      fmt(" %d", data_length));
release_rewrite_slot(c, msg$header_slot);
```

Figure 3: Deferring Writes to HTTP Content-Length Header

Once packet payloads are determined, the trace composer attaches TCP and IP headers to output packets. Also, if no data is written in the current packet cycle, but the trace composer needs to construct a packet to carry a TCP flag (SYN, RST, or FIN) or simply an acknowledgment, it generates an empty packet and attaches the headers to it.

For every output packet, the trace composer first fetches the TCP and IP headers of the most recent input packet on the same TCP flow and generate the new headers by modifying the following header fields:

1. If the trace is being anonymized, the source and destination addresses in the IP header are anonymized, as discussed in Section 4.4.2.

2. As the output trace does not have IP fragments (Bro reassembles fragments early in its protocol processing, making it too difficult to track their contribution to the final byte stream), the composer clears fragment bits in the IP header.

3. The composer keeps the original IP identification field, unless the (source IP, ID) pair has already appeared in the output trace, in which case we increment the ID till no conflict is found.

4. TCP sequence/acknowledgment numbers are adjusted to reflect new data lengths, as is the IP packet length field. The composer then recomputes the TCP and IP checksums. (Note that, similar to the case of fragments in the input trace, because Bro discards packets with checksum failures early in its processing, it is too difficult to propagate checksum errors into the transformed output.)

5. Currently the composer discards IP options, because Bro lacks an interface to access them, and some of them would take significant effort to address. The composer keeps certain TCP options, such as maximum segment size, window scaling and SACK negotiation (but not SACK blocks, due to the ambiguity of the location of the SACK'd data in the transformed stream), and timestamps; and replaces other options with NOP.

6. TCP flags are propagated, except that the composer removes the FIN flag. This is because additional packets may be inserted after the last one present in the input stream, and these must still be numbered in the sequence space before the final FIN to comply with TCP semantics. We can imagine a "conceptual" FIN that is reordered together with the payloads and comes only at the end of the data flow. Therefore, the trace composer inserts a FIN flag only when the flow reassembler has delivered, and the transformation script has processed, the last chunk of the flow.

3.4 Trace Rewriters for Trace Size Reduction

As a demonstration of the utility of trace transformation in addition to anonymization, we implemented trace rewriters for HTTP and SMTP to reduce the *size* of traces rather than the privacy of their embedded contents. At LBNL, for example, the volume of HTTP traffic often exceeds 50 GB per day. The site wants to continuously record this traffic (for intrusion detection analysis), but the volume proves problematic.

HTTP trace rewriter: Replaces HTTP entities beyond a specified size with their MD5 hash values, changes the Content-Length header to reflect the new data length, and keeps the original Content-Length and the actual data length in an "X-Actual-Data-Length" header (see Appendix A for an example). Testing it on a 729 MB trace file, and setting the threshold to 0 bytes (so all entities are replaced by hashes), the rewriter reduces the trace size to 25 MB, a factor of 29. If we compare the *gzipped* sizes of the traces (which the site often does with traces, in order to keep them longer before the disk fills up), the reduction becomes a factor of 69 (from 377 MB to 5.5 MB). Alternatively, we can implement more selective size reductions, such as stripping out only non-HTML objects in order to keep the cross-reference structure intact. Operationally, the site keeps the first 512 bytes of each entity, and keeps those with a MIME type of "text" in their entirety; this results in about a factor of 10 in size savings, yet retains enough information to help analyze most HTTP attacks to determine whether they succeeded.

SMTP trace rewriter: Replaces mail bodies with MD5 hash values and size information, but keeps all SMTP commands/replies and mail headers.

4. TRACE ANONYMIZATION

In this section we discuss general issues in trace anonymization, analyze four types of possible attacks, and present our anonymization scheme for LBNL FTP traces. Although the scheme is inevitably dependent on the specific policy approved by the site, the general techniques are also applicable to other sites and protocols.

4.1 Objectives of Anonymization

The information we try to hide through anonymization falls into two categories: *identities*, including identity of users, hosts, and data; and confidential *attributes*, e.g., passwords, or specifics of sensitive user activity [4].

The first step of developing an anonymization scheme is to decide what information in the trace we need to hide. For example, in anonymizing FTP traces, we aim to hide *identities* of clients, private data (hidden files), and private servers; and sensitive *attributes*: e.g., passwords, authentication keys, and in some cases filenames.

Confidential information can be exposed via direct means, or *inferred* via indirect means. Therefore, to hide the identity of client hosts, it may not be enough to just anonymize their IP addresses. We will shortly analyze four kinds of inference attacks that may reveal confidential information through indirect means, but before doing so we first discuss the anonymization primitives, i.e., how we anonymize basic data elements.

4.2 Anonymization Primitives

Constant Substitution: One way to anonymize a data element is to substitute the data with a constant, e.g., replace any password with the string "`<password>`".

Constant substitution is usually used to anonymize confidential attributes. Applying constant substitution to identifiers (e.g., IP addresses), however, is generally undesirable, as we would then no longer be able to precisely distinguish objects from one another. Instead, identifiers are usually anonymized with a 1-1 mapping, such as sequential numbering or hashing, so that the anonymized identifiers are still unique, as follows.

Sequential Numbering: We can sequentially number all *distinct* identifiers in the order of appearance, e.g., mapping files names to "`file1`", "`file2`", etc.

Hashing: One shortcoming of *sequential numbering* is that we have to keep the whole mapping history to maintain a consistent mapping during the anonymization process and across anonymizations. Instead, we can use a hash function as the mapping. Doing so requires no additional state during the anonymization process, and in addition using the same hash function across anonymizations will render a consistent mapping (assuming that the range of the hashing function is large enough so that likelihood of collision is negligible). To preserve confidentiality, the hash function must be one-way and preferably resistant to chosen plain-text attack, so that an adversary can neither discover the input from the output nor compute the hash by themselves. HMAC-MD5 (with a secret key) satisfies these requirements. Assuming the adversary can neither reverse MD5 nor extract the secret HMAC key, *hashing* is as secure as *sequential numbering*.

Prefix-Preserving Mapping: Sometimes it is valuable to preserve some of the structural relationships between the identifiers, which *sequential numbering* and *hashing* cannot do. For example, IP addresses can anonymized in a prefix-preserving way [12, 21] such that any two IP addresses sharing a prefix will share a prefix of the same length in their anonymized form. Prefix-preserving mapping can be similarly applied on the directory components of file names. While being valuable for some forms of analysis, prefix-preserving mapping also reveals more information about the identifiers and thus is more vulnerable to attacks [22].

Adding Random Noise: We can add noise to numeric values, e.g., file sizes, to make the result more resistant to fingerprinting attacks such as matching file sizes in the trace with public files [19]. We have not applied this primitive in our experiments, however, so we do not have experience regarding how effective it is and the degree to which it diminishes the value of the trace.

4.3 Inference Attacks

Besides anonymizing certain identifiers and attributes to eliminate direct exposure of identities and secret data, we also consider rewriting other data fields to prevent *indirect exposure*. In order to understand which data should be anonymized, we need to analyze how an adversary might use additional data to infer confidential information. Below we will discuss four kinds of inference techniques and how they relate to our FTP anonymization efforts.

4.3.1 Fingerprinting

"Fingerprinting" is the notion of an adversary recovering the identity of an object by comparing its attributes to attributes of objects known by the adversary. In order to do so the adversary has to know the fingerprints of the candidate objects. Thus, they cannot, for example, discover a previously unknown FTP server through fingerprinting.

We present here a brief analysis of possible fingerprinting on our anonymized FTP traces, to convey the flavor of problem:

1. Fingerprinting files: possible for public files, by looking for matches in file sizes, similar in spirit to the techniques of Sun et al [19].
2. Fingerprinting servers: possible for public servers, by the structure of their reply messages (especially the `220` greeting banner), help replies, SITE commands, or through fingerprinting files on the server. It is unclear to us whether it is possible to fingerprint servers by analyzing response timing.
3. Fingerprinting clients: there are at least two possible ways to fingerprint clients: 1) when the client displays some peculiar behavior known to the adversary; 2) through "active" fingerprinting: the adversary inserts a fingerprint for a certain client by sending packets to the trace collection site with a forged source address of the client's host address, and then looks for how these were transformed in the anonymized trace.

While fingerprinting of files and servers can expose usage patterns, this does not appear to be a serious issue because *who* made the access is not exposed.

Fingerprinting clients, on the other hand, would in some circumstances pose a significant privacy threat. But this is generally difficult for the adversary to accomplish. For the first type of fingerprinting, the client's sessions must possess peculiarities that survive the anonymization process, and the adversary must discover these. For the second type of fingerprinting, the fingerprint has to be inserted during trace collection. We discuss a defense against active fingerprinting, "knowledge separation", in Section 4.3.4.

A particular threat is that a class of clients displaying certain peculiar behaviors will stand out from other clients. If we want to eliminate this threat, we should eliminate or blur the distinction among client behaviors—which might significantly reduce the value of the trace.

4.3.2 Structure Recognition

Similar to fingerprinting, the adversary may also exploit the structure among objects to infer their identities. For example, traces of Internet traffic will often include sequential address scans made by attackers probing for vulnerable hosts. By assuming that an anonymized trace probably includes such scans, an adversary can hunt for their likely presence, such as by noting that a series of unanswered SYN packets occur close together in one part of the trace, or that (when using *sequential numbering*) suddenly a group of new hosts appears in the trace. They can then infer the original addresses of other hosts by the sequence they occupy in the scan, given the assumption that the scan started at a particular base address and proceeded sequentially up from it [18]. In addition, if the adversary has identified a single host in the trace (say a well-known server), they can then calibrate their inference by confirming that it shows up in the scan in the expected sequence.

4.3.3 Shared-Text Matching

When attributes or identifiers of different objects share the same text, the unmasking of one can lead to exposure of the other. For example, if there is both a user name "alice" and a file name "alice", the user name will be exposed if the adversary can identify the file. To avoid this attack, we apply "type-separation": the user name "alice" should be anonymized as the string "user+alice", and the file name as "file+alice". Generally it is good practice to avoid using the same text for distinct objects (e.g., files with the same name on different servers) unless there is some trace analysis value in doing so. The attack on prefix-preserving IP anonymization also exploits shared-text matching for cascading effects, where the shared text is the prefix.

4.3.4 Known-Text Matching

When both the original text and the anonymized text are known to the adversary, they can identify all appearances of the anonymized text in the trace. The knowledge required for a known-text attack is often obtained through fingerprinting.

One example is a "known server log" attack: if the adversary obtains the log of a server present in the trace, they may be able to identify the mapping between client addresses and anonymized addresses through fingerprinting, and then unmask the clients' activities on other servers. (Obtaining such logs is sometimes not difficult—for example, occasionally a query to a search engine will find them, because the logs are maintained in a publicly accessible manner.)

Another example is if the adversary can insert traffic with given strings, such as a particular user ID, into the trace, similar to the "active fingerprinting" discussed above. They can then observe how the string was mapped, and look for other occurrences of the resulting text in order to unmask instances of the same original text.

A general method to counter known-text attacks is through "knowledge separation". This is similar to the type-separation defense against shared-text matching discussed above. For example, to counter a "known server log" attack, we can anonymize a client IP differently depending on the server it accesses. To counter the user ID insertion attack, we can anonymize user IDs differently depending on whether the login is successful or not (an alternative is to anonymize user IDs depending on the client's IP address). Similarly, "active fingerprinting" with forged source IPs can defeated by anonymizing addresses differently for connections that are never established, since the adversaries will fail to complete the TCP three-way handshake unless they can conduct an initial-sequence-number guessing attack.

When we apply "knowledge separation", a single object can have multiple identifiers in the anonymized trace, which reduces the value of the trace for some types of analysis. This is a basic trade-off, and the choice of the degree to incur it will be policy-dependent.

4.4 Case Study: FTP Anonymization

In light of these possible attacks and defenses, we now turn to the anonymization scheme we used for LBNL's FTP traces. Though the scheme is inevitably dependent on the specific policy approved by the site, and thus may not be directly applicable to other sites, we believe the considerations and techniques, for instance, the "filter-in" principle, will be also applicable to other site policies and other application protocols. Accordingly, we discuss in detail relevant points of the resulting anonymization process. The full scheme can be found at [6].

The FTP traces were collected at the Internet access point (Gigabit Ethernet) of LBNL, and contain incoming anonymous FTP connections to port 21. The traces do not include any of the transferred FTP items (files uploaded or downloaded, or directory contents corresponding to the FTP "LIST" command), but only requests and replies in the traces.

As stated above, our objectives are: 1) ensure that the anonymization hides the identity of clients, non-public FTP servers, and non-public files, as well as confidential authentication information;[3], and 2) the anonymization keeps the original request/reply sequence and other nonsensitive information intact.

In some ways, these goals and the resulting traces are quite modest. But we believe that the path to site's becoming open to releasing traces with packet contents is one that must be tread patiently, as sites quite naturally must develop a solid sense that trust in the anonymization process is warranted.

Self-Explaining: Besides the above objectives, we designed the anonymization scheme to be *self-explaining*: it should be easy for other people to examine and validate the scheme by merely looking at the scheme description or the policy script, without being familiar with every detail of FTP. We believe this is particularly important in order for the policy makers at a site to understand and accept trace anonymization.

4.4.1 The Filter-In Principle

The key to obtaining a robust and coherent anonymization scheme is to apply the "filter-in" principle, which is that the anonymization policy script explicitly specifies what data to leave in the clear, and everything else is anonymized (or removed). Thus, "filtering-in" implies using "white lists" of what is okay instead of "black lists" of what is disallowed. The design choice in our framework of "explicit rewriting" also reflects the "filter-in" principle.

It is critical to employ "filter in" instead "filter out". Anonymizing FTP traffic is complex enough that if we try to "filter out" private information by enumerating all the sensitive data fields, it is very likely that we will miss some of them. Also, a "filter-out" scheme would be hard to verify, unless the verifier can themselves enumerate all of the sensitive fields.

Following the "filter-in" principle, the difference between a crude anonymization script and a refined one is that the refined script will preserve more nonsensitive information in the output trace; but the two scripts should be equally privacy-safe (though we must keep in mind the maxim that complexity is the enemy of robust security). Also, a "filter-in"-style anonymization scheme is to some degree self-explaining—verification of the scheme does not require enumerating every possibility.

4.4.2 Selected Details of FTP Anonymization

IP addresses: (which appear in IP headers, PORT arguments, and some reply messages such as reply to the PASV command) are sequentially numbered, since the site views preserving client privacy as vital. (Recognizing IP addresses in reply messages is discuss in Section 4.4.4.)

User IDs: (arguments of USER/ACCT commands) are anonymized except for "anonymous", "guest", and "ftp". However, the anonymizer leaves a user ID in the clear if the login attempt fails and the user ID is one of the IDs defined as sensitive in Bro's default security policy (for example, "`backdoor`", "`bomb`", "`issadmin`", "`netphrack`", "`r00t`", "`sync`", "`y0uar3ownd`", and many others). This allows us to preserve one form of attack, namely attempted backdoor access, without exposing any actual account information.

When we anonymize a user ID, we apply HMAC-MD5, annotating the user ID prior to hashing with 1) the server IP to prevent "shared-text" matching, and 2) an indication of whether the login was successful to prevent "known-text" matching.

Password: we replace the arguments of PASS commands with the string "`<password>`". (An alternative would be to hash passwords for anonymous logins, with the email addresses annotated with the client IP address to achieve "knowledge separation".)

File/directory names: are replaced by the string "`<path>`" for non-anonymous logins. For anonymous logins, file names are left in the clear if they appear on a white list of well-known sensitive file names (e.g., "`/etc/passwd`"), in order to preserve occurrences

[3]Here, hiding a "non-public" server/file means that if an adversary does not know where to find the server/file beforehand, they will not be able to find it after looking at the anonymized traces.

of attacks; and anonymized with hashing otherwise. The hashing input is the absolute path annotated with the server IP to minimize shared-text matching across directories or servers. The reason to anonymize file names even for anonymous FTP traffic is that we cannot readily tell truly public files apart from private (hidden) ones that happen to be access using anonymous FTP, but only by users who know the otherwise unpublicized location of the file.

Arguments of commands with pre-defined argument sets: (TYPE, STRU, MODE, ALLO, REST, MACB): are left intact if well-formed. For example, a TYPE argument should match the regular expression `/([AE](␣[NTC])?)|I|(L[0-9]+)/` according to RFC 959. However, the anonymizer does not assume clients follow the RFC—it checks whether the argument matches the pattern, and leaves it in the clear only if that is the case, otherwise anonymizing the argument as a string.

We apply similar techniques for the "HELP" and "SITE" commands, for which we only expose the arguments if they match a manually determined "white list" of privacy-safe HELP/SITE arguments.

Unrecognized commands: are anonymized along with their arguments and recorded for optional manual inspection.

Timestamps/dates: are left in the clear. While timestamps could help an adversary match up known traffic (such as traffic they injected) with its occurrence in the trace, there are enough other ways the adversary can perform such matching (by making the injected traffic singular) that leaving them intact costs little. On the other hand, timestamps are valuable for various research purposes.

File sizes: are considered to be safe. As argued when analyzing fingerprinting, exposing file sizes may allow the adversary to identify public files. But this is not a concern for LBNL.

Server software version/configuration: is also considered to be safe, as the information that can be inferred from the trace can be readily obtained through other means (since the servers are public).

4.4.3 Refining with Manual Inspection

Whether data is to be left in the clear or anonymized, the anonymous script logs the decision and the reason for later inspection.[4] Identical entries are only logged once. Inspection of the log (with various text processing tools) helps us to discover 1) privacy holes (or to demonstrate the absence of holes), and also 2) overly conservative anonymization of nonsensitive information (important for working towards more refined scripts). We discuss log inspection techniques in detail below.

A "filter-in"-style script always makes conservative judgments on unknown data. Sometimes it can be too conservative, missing an opportunity to expose interesting, nonsensitive data, e.g., a mistyped command like "UUSER" or a user id like "annonymous". It is difficult to hardwire such commands and user names into the general anonymization script, as they may appear in unpredictable forms. Nevertheless, these special cases do not appear very often in traces, so we can afford to *manually inspect* each case by looking at the log after anonymization and then customizing the script to expose the nonsensitive ones. Figure 4 shows three log entries we have seen: the first entry records a common-case anonymization of a path name; while the other two, recording anonymizations of the "UUSER" command and user name "annonymous", are the kinds of entries we look for during manual inspection.

[4]Here we assume that the administrator of the trace anonymization can see the original trace—this helps in verifying results and generating better traces.

Note that the customization for special cases should be *optional*. The script should always first anonymize any unknown data, and should make no assumptions about whether the log will be manually inspected.

As most entries in the anonymization log record the anonymization of "common" cases, the trick to digging up special cases is to look for deviant entries through *text classification*. Here, we examine command arguments as an example to illustrate how we discover special cases:

First, we classify entries by the type of data being anonymized. The type can be, for example, a non-guest user name (e.g., "annonymous"), or a non-public file name, or the argument of a PORT command. Some types of anonymization, e.g., of path names and passwords, happen very often, while others rarely appear in the log. These rare types of anonymization often present interesting cases. For example, for a trace of an FTP server that only allows anonymous login, there can still be a few user names being anonymized. We have seen: "`anno`", "`anonyo\010`", "`anonymouse`", "`help`", and "`anamouse`", as well as a password mistyped for a `USER` command. Except for the password, all of the other user names actually do not reveal any private information. *But it's important to catch the password.* Note that none of these strange user names will appear in the output trace unless we modify the script to explicitly allow them, so the password will not appear without specific action to keep it.

Furthermore, we look for "malformed" path names—those do not match a heuristic pattern for well-formed path names. We find, for example: "`#`", "`\xd0\xc2\xce\xc4\xbc\xfe \xbc\xd0`", "`/n/nThis file was not retrieved by Tele-port Pro, because it did not meet the project`".[5]

In addition, applying similar techniques lets us find misspelled commands, or commands containing control characters: e.g., "`USE`", "`UUSER`", "`RETR<BS><BS><BS><BS>`", all of which we have seen in practice (these commands likely indicate users typing directly rather than using client software).

4.4.4 Reply Anonymization

An FTP reply consists of a reply code and a text message. We leave reply codes in the clear, as they do not reveal any private information. Reply messages, on the other hand, do often contain sensitive information and are hard to anonymize because there is no standard format for most reply messages—the format depends on the server implementation and its configuration.

One possibility is to discard the original text (except for replies to PASV, which are well-defined) and replace it with a dummy message. This has the virtue of being simple. On the other hand, reply messages do sometimes carry useful information that cannot be inferred from the reply codes. For example, a reply of code 530 (denial of login) usually explains why the login was rejected–it can be "guest login not permitted" or "Sorry, the maximum number of users from your host are already connected.". Such information can be valuable in some cases. So we explored methods to anonymize FTP replies.

As messages may contain variables such as file names/sizes, dates, and domain names, there can be countless distinct messages. However, we observe that there is only a limited set of message *templates*, as the number of templates is bounded by the number of different server software/configurations at the site. And we can extract templates (along with human assistance) by comparing messages against each other and distilling the common parts. Figure 5 shows a few example message templates. Once we have extracted

[5]Teleport Pro is the name of an offline browser.

```
anonymize_arg: (path name) [CWD] "conferencing" to "U42117b96U" in [xxx.xxx.xxx.xxx/xxxx > xxx.xxx.xxx.xxx/ftp]
anonymize_cmd: (unrecognized command) "UUSER" [anonymous] to "U7b402a69U" in [xxx.xxx.xxx.xxx/xxxx > xxx.xxx.xxx.xxx/ftp]
anonymize_arg: (user name) [USER] "annonymous" to "Ufb6db9afU" in [xxx.xxx.xxx.xxx/xxxx > xxx.xxx.xxx.xxx/ftp]
```

Figure 4: Anonymization Log Entries

the message templates, we can parse messages by matching them against the templates and thereby understanding the semantics of the data elements in the text.

Message templates are first automatically extracted by a script then manually sanitized before used for template matching. The automated template extraction is done in three steps: splitting, abstraction, and merging (as shown in Figure 6). We first *split* a message into parts—each part contains a word or a data element such as an IP address or a file name. Next, in *abstraction*, we try to guess whether each part is a variable or a constant part of the message template. Through *abstraction* we are able to find most of variable slots in message templates, and *merging* helps to reveal the rest of them. We merge two templates when they are identical on all but one part, and this process is iterated till no templates can be further merged.

The message extraction process is refined through the accumulation of experience. We found that the key issue in abstraction is to recognize the corresponding command argument echoed in the reply message. This is tricky because the echoed argument is sometimes different from the original argument, particularly when it is a file name. For example, the echoed argument can be the absolute file path or only contain the base file name with the directory parts. Therefore we need to recognize variants of the argument. The key for good message splitting is to know where *not* to split. By default we split at spaces and punctuation; however, we do not want to split an IP address or a file name, otherwise they cannot be recognized during abstraction.

Extracted message templates need to be examined and sanitized before being used for message matching. This can be a tedious process and we strived to minimize the required effort. Currently, when extracting templates from a set of ten-day long FTP traces, which contain more than 1.4 M lines of replies in 22.6 K connections to 318 distinct servers, we wound up with 461 message templates for 32 kinds of reply codes. Among the 461 templates, 25 require sanitization to remove server identity information. Examining a few hundreds of templates is feasible but still not easy—perhaps this is the price for processing free format text.

4.4.5 Verification

Verification is a fundamental step of the anonymization process. No matter how much thought we apply to the anonymization policy, the safety of the anonymization also depends on the correctness of the policy script and on the underlying Bro mechanisms. Therefore, besides inspecting the anonymization description and script, it is also important to examine the output trace directly.

Ideally, the verification process would guarantee that the transformed trace complies with the *intended* anonymization policy. This is a different notion that the *expressed* anonymization policy, due to the possiblity of errors occurring in coding up the expression. Our strategy therefore is to attempt to analyze the general properties of the transformed trace without tying these too closely to the anonymization script that was used to effect the transformation. As such, we cannot guarantee that there are no "hole" in the anonymized trace (but indeed doing so appears fundamentally intractable). Instead, we aim to provide another dimension of precaution. In general, it is particularly important to have a strong "verification story" in order to persuade sites that the anonymization process will meet their requirements.

For verification we do not use Bro to parse the output trace's packets—doing so would introduce a common point of failure across anonymization and verification. Instead, we look at the packets directly, using different tools. Automating the verification process remains an open problem—currently, it requires human assistance, although some of the steps can be automated to reduce the burden.

For packet headers, we inspect the source and destination IP addresses. As the anonymized addresses are sequentially numbered, verification that these lie in the expected range can be performed automatically.

For FTP requests in packet payloads, we enumerate all distinct commands and arguments present in the trace, except those which are already hashed (hash results follows a particular textual format and thus can automatically excluded). When the text parts of reply messages are discarded, it is straightforward to verify that FTP replies only contain reply codes and a placeholder of dummy text.

When we choose to anonymize reply messages, verification consists of two parts, checking vocabulary and numbers, respectively. Vocabulary checking is similar to message template extraction, but simpler and implemented separately. Messages are again split at blanks and punctuation, this time without worrying about special cases as in splitting for message template extraction. Next we abstract the parts by two rules: 1) if a part is a decimal number, substitute it with the string "`<num>`"; 2) if a part is a hashing output, substitute it with the string "`<hash>`". This way we can reduce 1.4 M anonymized messages to about 600 patterns. We then manually inspect these, which can be expedited by first sorting them so that similar patterns are clustered.

In checking numbers we are mainly concerned about numbers constituting IP addresses. Accordingly, we look for any four consecutive number parts in split messages and record each instance that does not fall within the range of anonymized addresses. Interestingly, such cases *do* appear, though they are quite rare, and safe—e.g., part of a software version string such as "wu-2.6.2(1)".

Verification helped us find a potential hole in an earlier version of our anonymization script. We found two suspicious command arguments: "`GSSAPI`" and "`KERBEROS_V4`". Though the strings themselves do not disclose any private information, their appearance is alarming because they are not defined anywhere to be "safe" in the script.

Looking into the logs revealed that they were arguments for two rejected "AUTH" commands. According to RFC 2228, the argument for the "AUTH" command specifies the authentication mechanism. Thus, a rejected mechanism seems safe to expose. However, doing so overlooks the possibility that a user might mistakenly specify sensitive information, such as a password, instead of an authentication *mechanism*. A "fail-safe" solution is to white list "`GSSAPI`" and "`KERBEROS_V4`" and anonymize any unknown argument for the "AUTH" command.

```
150 |opening| |ascii, binary| |mode| |data| |connection| |for| |˜ arg| |˜ ip| |˜ num| |˜ num| |bytes|
211 |connected| |to| |˜ domain, ˜ ip|
220 |welcome| |to| |˜ *| |ftp| |server|
550 |˜ arg| |not| |a| |directory|
```

Figure 5: FTP Reply Message Templates

```
message:     "150 Opening BINARY mode data connection for /def.pdf (123.45.67.89,50034) (156678 bytes)"
split →      "150 |opening| |binary| |mode| |data| |connection| |for| |/def.pdf| |123.45.67.89| |50034| |156678| |bytes|"
abstract →   "150 |opening| |binary| |mode| |data| |connection| |for| |˜ arg| |˜ ip| |˜ num| |˜ num| |bytes|"
merge →      "150 |opening| |ascii, binary| |mode| |data| |connection| |for| |˜ arg| |˜ ip| |˜ num| |˜ num| |bytes|"
```

Figure 6: Message Template Extraction

4.4.6 Discussion

Integrity of Output Trace: Besides the absence of private information, we also want to check whether the packets, TCP flows, and FTP requests and replies in the anonymized trace are all *well-formed*. To do so, we run Bro's FTP analyzer on the anonymized traces to see whether Bro can reassemble the TCP flows and parse the FTP requests and replies. We compare the FTP logs from both traces. Bro's FTP log records start and finish of FTP sessions and all requests and replies in the session. For a day-long FTP trace of 80 MB, 8,871 connections, and 86,908 request-reply pairs, we find that the two logs have the same FTP session starting timestamps,[6] request command sequences (not including the arguments) and reply code sequences, also at the same timestamps. For command arguments and reply messages, we cannot compare them directly as of course many of them are anonymized. We randomly picked a few sessions and manually checked the arguments and messages.

Anonymized Traces for Intrusion Detection: As mentioned earlier, packet traces are particularly valuable for research on network intrusion detection. So we very much want trace anonymization to preserve intrusion-like activities. This applies both to preserving actual attacks, but, even more so, unusual-but-benign traffic that stresses the false-positive/false-negative accuracy of intrusion detection algorithms. This latter is particularly important because it is often a key element missing from assessments of network intrusion detection mechanisms—it is easy for researchers to attain traces of actual attacks, because they can generate these using the plethora of available attack tools, but it is much more difficult today for researchers to attain detailed traces of background traffic.

Generally whether an attack survives anonymization depends on both its characteristics and how it is detected. Some FTP intrusions are recognized by signatures of files or user IDs the intruder tries to access or login as. For example, directory name "`tagged`" is often associated with FTP warez attacks; failed "`root`" or "`sysadm`" login attempts suggest server backdoor probing. Preserving these attacks requires leaving relevant identifiers in the clear. Fortunately the identifiers are mostly well-known and do not expose private identities, so they can kept through anonymization by establishing a white list for "sensitive" file names and user IDs to leave in the clear. To do so, however, requires knowing the attack signatures beforehand; thus, attacks with unknown signatures may still be lost in anonymization.

Other types of intrusions are recognized by activity patterns rather than identifier signatures. Most of these attacks can survive anonymization. For instance, port scanning is marked by unanswered (or responded by TCP-RST) TCP-SYN packets from the same source host to different destination hosts; successive failed attempts at creating directories on multiple servers may imply an FTP warez attack.

[6]In some cases, Bro's connection termination is triggered by a timer, which results in slightly different session finish timestamps.

FTP analyzer	131 seconds
FTP analyzer + anonymizer	1009 seconds
FTP analyzer + dummy rewriter	192 seconds

Figure 7: Execution time of various FTP policy scripts

Performance: Figure 7 shows the CPU time spent on a 1 GHz Pentium III processor running on the day-long trace mentioned above. We see that the FTP anonymizer, which also requires the FTP analyzer, is 7.7 times slower than the FTP analyzer. To understand where time is spent, we also tested Bro with a dummy FTP trace rewriter, which simply writes the original requests and replies to the output trace. We find that the execution overhead of the anonymizer script itself heavily dominates, comprising 81% of the total processing. The time is spent performing numerous hash table lookups, string operations, and regular expression matches, and generating a 3.8 MB anonymization log. We find this performance adequate, especially for off-line anonymization. It even suffices for on-line anonymization for FTP, though when extended to a higher volume protocol such as HTTP may prove problematic.[7]

5. CHALLENGES AND NEW DIRECTIONS

We view our work as an early push towards making richer packet traces available to the research community. There is still much to be done in this area. From our experience, we believe the main challenges include: 1) to formalize security considerations and the process of developing an anonymization scheme; 2) to automate the process of anonymization and verification; 3) to keep more packet dynamics in the transformed traces. Below we briefly discuss each of these.

Formalizing Anonymization: In Section 4 we described our methodology for trace anonymization and analyzed four types of inference techniques, but our analysis is far from being formal or complete. While accumulation of experience will help us have a better understanding of the relationship among various data elements, developing a formal model for anonymization would be a big step forward beyond the intuitive methods. A formal model would mean that users can have a complete view of the threats and rigorously deduce a detailed anonymization scheme from the objectives. However, a major difficulty in pursuing such models is the degree to which anonymization inherently involves knowledge of

[7]Note that the HTTP rewriter used to reduce HTTP packet traces as discussed in Section 3.4 runs on-line, processing nearly 100 times the daily data volume, though in a simpler fashion.

semantics, including sometimes quite high-level abstractions, and also corner cases that can inadvertently leak information.

Automating the Anonymization Process: Although the anonymization process has been much simplified by operating at the application-protocol level, currently we still need human assistance in tailoring scripts for traces (4.4.3), processing free-format texts (4.4.4), and result verification (4.4.5). The first two, though being optional, often largely improve the quality of the output trace. The last (verification) is an essential step which we cannot do without human interaction. On the other hand, fully automating anonymization will bring substantial benefits: 1) it will minimize human effort in releasing traces, making it easier for sites to make traces available; 2) it is critical for environments where the trace providers themselves are not allowed to see the original traces (e.g., for traces collected at some ISPs); 3) automated verification will foster a model of "script↔data" exchange, where users send anonymization scripts to data owners who use them to easily generate traces returned to the users [13].

The key for automating result verification is to make the anonymization scheme "understandable" to the verifier program. One way is to design a declarative (instead of procedural) language for the anonymization scripts. Being declarative, the anonymization scheme specification is also amenable to verification, which is necessary to ensure that the scheme is correctly specified.

Keeping Traffic Dynamics: One fundamental difficulty of keeping the original traffic dynamics is that lengths of data may be changed during transformation, and the new lengths must be reflected in TCP/IP headers to keep packets "well-formed". Therefore there is not a single best way to keep the original dynamics. We are investigating ways to retain as much of the dynamics as possible without dragging the user into low-level packet processing. One possibility is to create an out-of-band channel to convey information such as original packet lengths, fragmentation, retransmission, etc.

Also it is particularly difficult to process two parallel versions of the data, for instance, in the presence of inconsistent TCP retransmissions, because traffic parsing is stateful. So we have to remove at least one version from the anonymized stream, even though in some contexts (e.g., analyzing possible intrusion detection evasions seen in practice [16]) it would be very useful to have both copies of inconsistent retransmissions retained.

6. RELATED WORK

TCPdpriv [12] anonymizes `tcpdump` traces by stripping packet contents and rewriting packet header fields. One of its features is a form of "prefix-preserving" anonymization of IP addresses (the "-A50" option). [22] analyzes the security implications of this anonymization, proposing an approach that might be used to crack the "-A50" encoding by first identifying hosts with well-known traffic pattern (e.g., DNS servers). Xu et al proposed a cryptography-based scheme for prefix-preserving address anonymization [21]. The scheme can maintain a consistent anonymization mapping across multiple anonymizers using a shared cryptographic key. Peuhkuri presented an analysis of the private information contained in TCP/IP header fields and proposed a scheme to anonymize packet traces and store the results in a compressed format [17]. Peuhkuri's scheme for network addresses anonymization cannot be directly applied to our work because the scheme generates 96 bits instead of 32 bits for each address, and we are constrained by needing to generate output in `tcpdump` format. All of these works address only the anonymization of TCP/IP headers, with no mechanisms for retaining packet payloads.

NetDuDe (NETwork DUmp data Displayer and Editor) [9] is a GUI-based tool for interactive editing of packets in `tcpdump` trace files. NetDuDe itself does not parse application level protocols, but allows user to write plug-in's for packet processing, e.g., a checksum fixer plug-in can recompute checksums and update the checksum fields in TCP and IP headers.

There has also been considerable work on extracting application-level data from online traffic, though without significant applications to content-preserving anonymization. Gribble et al built an HTTP parser to extract HTTP information from a network sniffer [7]. Feldmann in [5] describes BLT, a tool to extract complete HTTP headers from high-volume traffic, and discusses various challenges in extracting accurate HTTP fields. Pandora [14] is a component-based framework for monitoring network events, which contains, among others, components to reconstruct HTTP data from packets. It is similar in spirit to Windmill [11]. Ethereal is able to reconstruct TCP session streams, and parses the stream to extract application protocol level data fields [3]. The fields can be used to filter the view of the trace. Ethereal has a GUI-based interface to display trace data. There are also numerous commercial network monitoring systems that can extract application-level information, e.g., EtherPeek[20].

There are also efforts on setting up honeypots [8] and break-in challenges [2] to collect traces of network intrusions. Such pure intrusion traces have the virtue of containing little private information, as the target hosts are not used for other purposes. For the same reason, however, the traces do not contain background traffic with various unusual-but-benign activities, and thus are very different from traffic at an operational site.

Finally, Mogul argues "Trace Anonymization Misses the Point" [13], proposing an alternative strategy to trace anonymization—instead of sharing anonymized traces, researchers send reduction agents to the site that has the source trace data. We believe our tool is in fact complementary to this sort of approach. Mogul raises the question: what kind of code should be sent to the source sites? Our answer is: "a Bro script for trace transformation."

7. SUMMARY

In this work we have designed and implemented a new tool for packet trace anonymization and general purpose transformation. The tool offers a great degree of freedom and convenience for trace transformation by providing a high-level programming environment in which transformation scripts operate on application-level data elements.

Using this framework, we developed an anonymization script for FTP traces and applied it to anonymizing traces from LBNL for public release. Unlike previous packet trace anonymization efforts, packet payload contents are included in the result. We discussed the key anonymization principle of "filter-in" as opposed to "filter-out", and the crucial problem of *verifying* the correctness of the anonymization procedure. We also analyzed a class of inference attacks and how we might defend against them.

We believe this tool offers a significant step forward towards ending the current state of there being *no* publicly available packet traces with application contents. As such, we hope to help open up new opportunities in Internet measurement and network intrusion detection research.

Acknowledgements

We would like to thank the Lawrence Berkeley National Laboratory, and Jim Rothfuss and Sandy Merola in particular, for working with us to realize the public release of traces of LBNL traffic that include packet contents; Larry Peterson for his support throughout this work; the anonymous SIGCOMM reviewers; our shepherd Greg Minshall; Lujo Bauer, Ed Felten, Brent Waters, Chi Zhang, and other colleagues at Princeton for their insights and suggestions; and the staff of the Princeton Department of Computer Science for providing us traces for testing. This work was supported in part by NSF grant ANI-9906704, DARPA contract F30602–00–2–0561, and the Intel Corporation.

8. REFERENCES

[1] S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security*, 3(3):186–205, August 2000.

[2] Capture the capture the flag. http://www.shmoo.com/cctf/.

[3] G. Combs. *The Ethereal Network Analyzer*. http://www.ethereal.com/.

[4] Federal Committee on Statistical Methodology. Report on statistical disclosure limitation methodology (statistical policy working paper 22), 1994. http://www.fcsm.gov/working-papers/spwp22.html.

[5] A. Feldmann. BLT: Bi-layer tracing of HTTP and TCP/IP. In *Proceedings of WWW-9*, May 2000.

[6] Anonymized FTP traces. http://www-nrg.ee.lbl.gov/anonymized-traces.html.

[7] S. D. Gribble and E. A. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proc. USENIX Symp. on Internet Technologies and Systems*, December 1997.

[8] The honeypot challenge. http://project.honeynet.org/misc/chall.html.

[9] C. Kreibich. *NetDuDe (NETwork DUmp data Displayer and Editor)*. http://netdude.sourceforge.net/.

[10] R. Lippmann, S. Webster, and D. Stetson. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. In *Proceedings of Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[11] G. R. Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proceedings of ACM SIGCOMM*, 1998.

[12] G. Minshall. *TCPdpriv: Program for Eliminating Confidential Information from Traces*. Ipsilon Networks, Inc. http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html.

[13] J. Mogul. Trace anonymization misses the point. Presentation on WWW 2002 Panel on Web Measurements.

[14] S. Patarin and M. Makpangou. Pandora: A flexible network monitoring platform. In *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, June 2000.

[15] V. Paxson. *Bro: A System for Detecting Network Intruders in Real-Time*. http://www.icir.org/vern/bro-info.html.

[16] V. Paxson. Bro: A system for detecting network intruders in real time. *Computer Networks*, December 1999.

[17] M. Peuhkuri. A method to compress and anonymize packet traces. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, November 2001.

[18] S. Savage. Private communication.

[19] Q. Sun, D. R. Simon, Y. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu. Statistical identification of encrypted web browsing traffic. In *Proceedings of IEEE Symposium on Security and Privacy, Oakland, CA, USA*, May 2002.

[20] WildPackets, Inc. *EtherPeek*. http://www.etherpeek.com/.

[21] 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 ACM SIGCOMM Internet Measurement Workshop*, November 2001.

[22] T. Ylonen. Thoughts on how to mount an attack on tcpdpriv's "-a50" option. http://ita.ee.lbl.gov/html/contrib/attack50/attack50.html.

APPENDIX

A. A SAMPLE HTTP TRACE TRANSFORMATION

The original trace was collected by `tcpdump` recording a retrieval of the www.google.com homepage. The `tcpdump` output (with wrapped packet summary lines and TCP payloads) of the original trace is shown on the next page.

We use our tool to transform the trace with a script that:

1. Replaces the data entity with its MD5 hash value (in this case, "`867119294265e3f445708c3fcfb2144f`");
2. Rewrites the `Content-length` field to reflect the length of the MD5 hash value;
3. Adds the header: "X-Actual-Data-Length: 2709; gap=0, content-length= 2709" to record the original Content-length field and how many bytes are actually transferred.

The `tcpdump` output of the transformed trace is also on the next page.

Note that "Write-Deferring" is applied here: the new headers are written at the position of the original `Content-length` header, even though the actual data size is not determined until all of the data is seen. The script defers writing the headers until the end of the message and then writes back to the reserved position.

Furthermore, by changing only one line of the script, from:

```
msg$abstract = md5_hash(data);
```

to:

```
msg$abstract =
 subst_string(data, "Google", "Goooogle");
```

the script then replaces every occurrence of "Google" in the data entity with "Goooogle", instead of replacing the whole data entity with its MD5 hash value. Next page shows part of the transformed trace. (There are four occurrences of "Google" in the original message, thus the Content-length increases from 2709 to 2717.) Note that sequence and acknowledgment numbers between the traces differ due to packet reframing and the addition of X-Actual-Data-Length headers.

Original trace:

```
1044328495.549695 192.150.187.28.1472 > 216.239.51.101.80:
    S 1352447574:1352447574(0) win 57344
    <mss 1460,nop,wscale 0,nop,nop,timestamp 92919815 0> (DF)
1044328495.632608 216.239.51.101.80 > 192.150.187.28.1472:
    S 3009119707:3009119707(0) ack 1352447575 win 1460
    <mss 1460,nop,nop,timestamp 752104543 92919815,nop,wscale 0> (DF)
1044328495.632647 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 1 win 57920
    <nop,nop,timestamp 92919823 752104543> (DF)
1044328495.632966 192.150.187.28.1472 > 216.239.51.101.80:
    P 1:81(80) ack 1 win 57920
    <nop,nop,timestamp 92919823 752104543> (DF)
0x0030  2cd4 345f 4745 5420 2f20 4854 5450 2f31 ,.4_GET./.HTTP/1
0x0040  2e30 0d0a 5573 6572 2d41 6765 6e74 3a20 .0..User-Agent:.
0x0050  5767 6574 2f31 2e35 2e33 0d0a 486f 7374 Wget/1.5.3..Host
0x0060  3a20 7777 772e 676f 6f67 6c65 2e63 6f6d :.www.google.com
0x0070  3a38 300d 0a41 6363 6570 743a 202a 2f2a :80..Accept:.*/*
0x0080  0d0a 0d0a                               ....
1044328495.716691 216.239.51.101.80 > 192.150.187.28.1472:
    . ack 81 win 30660
    <nop,nop,timestamp 752104551 92919823> (DF)
1044328495.737787 216.239.51.101.80 > 192.150.187.28.1472:
    P 1:1449(1448) ack 81 win 31856
    <nop,nop,timestamp 752104553 92919823> (DF)
0x0030  0589 d80f 4854 5450 2f31 2e30 2032 3030 ....HTTP/1.0.200
0x0040  204f 4b0d 0a43 6f6e 7465 6e74 2d4c 656e .OK..Content-Len
0x0050  6774 683a 2032 3730 390d 0a43 6f6e 6e65 gth:.2709..Conne
0x0060  6374 696f 6e3a 2043 6c6f 7365 0d0a 5365 ction:.Close..Se
0x0070  7276 6572 3a20 4757 532f 322e 300d 0a44 rver:.GWS/2.0..D
0x0080  6174 653a 2054 7565 2c20 3034 2046 6562 ate:.Tue,.04.Feb
0x0090  2032 3030 3320 3033 3a31 343a 3535 2047 .2003.03:14:55.G
0x00a0  4d54 0d0a 436f 6e74 656e 742d 5479 7065 MT..Content-Type
0x00b0  3a20 7465 7874 2f68 746d 6c0d 0a43 6163 :.text/html..Cac
0x00c0  6865 2d63 6f6e 7472 6f6c 3a20 7072 6976 he-control:.priv
0x00d0  6174 650d 0a53 6574 2d43 6f6f 6b69 653a ate..Set-Cookie:
0x00e0  2050 5245 463d 4944 3d31 6538 6337 3538 .PREF=ID=1e8c758
0x00f0  6231 6632 3965 3836 643a 544d 3d31 3034 b1f29e86d:TM=104
0x0100  3433 3238 3439 353a 4c4d 3d31 3034 3433 4328495:LM=10443
0x0110  3238 3439 353a 533d 6638 344d 6753 7948 28495:S=f84MgSyH
0x0120  3347 452d 3439 5070 3b20 6578 7069 7265 3GE-49Pp;.expire
0x0130  733d 5375 6e2c 2031 372d 4a61 6e2d 3230 s=Sun,.17-Jan-20
0x0140  3338 2031 393a 3134 3a30 3720 474d 543b 38.19:14:07.GMT;
0x0150  2070 6174 683d 2f3b 2064 6f6d 6169 6e3d .path=/;.domain=
0x0160  2e67 6f6f 676c 652e 636f 6d0d 0a0d 0a3c .google.com....<
0x0170  6874 6d6c 3e3c 6865 6164 3e3c 6d65 7461 html><head><meta
0x0180  2068 7474 702d 6571 7569 763d 2263 6f6e .http-equiv="con
0x0190  7465 6e74 2d74 7970 6522 2063 6f6e 7465 tent-type".conte
0x01a0  6e74 3d22 7465 7874 2f68 746d 6c3b 2063 nt="text/html;.c
0x01b0  6861 7273 6574 3d49 534f 2d38 3835 392d harset=ISO-8859-
0x01c0  3122 3e3c 7469 746c 653e 476f 6f67 6c65 1"><title>Google
0x01d0  3c2f 7469 746c 653e 3c73 7479 6c65 3e3c </title><style><
...
0x0360  3237 3620 6865 6967 6874 3d31 3130 2061 276.height=110.a
0x0370  6c74 3d22 476f 6f67 6c65 223e 3c2f 7464 lt="Google"></td
...
1044328495.737951 216.239.51.101.80 > 192.150.187.28.1472:
    P 2897:3025(128) ack 81 win 31856
    <nop,nop,timestamp 752104553 92919823> (DF)
0x0030  0589 d80f 6f6e 743e 0a3c 703e 3c66 6f6e ....ont>.<p><fon
0x0040  7420 7369 7a65 3d2d 323e 2663 6f70 793b t.size=-2>&copy;
0x0050  3230 3033 2047 6f6f 676c 653c 2f66 6f6e 2003.Google</fon
0x0060  743e 3c66 6f6e 7420 7369 7a65 3d2d 323e t><font.size=-2>
0x0070  202d 2053 6561 7263 6869 6e67 2033 2c30 .-.Searching.3,0
...
1044328495.737987 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 1449 win 57920
    <nop,nop,timestamp 92919833 752104553> (DF)
1044328495.738022 216.239.51.101.80 > 192.150.187.28.1472:
    F 3025:3025(0) ack 81 win 31856
    <nop,nop,timestamp 752104553 92919823> (DF)
1044328495.738054 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 1449 win 57920
    <nop,nop,timestamp 92919833 752104553> (DF)
1044328495.739267 216.239.51.101.80 > 192.150.187.28.1472:
    P 1449:2897(1448) ack 81 win 31856
    <nop,nop,timestamp 752104553 92919823> (DF)
0x0030  0589 d80f 2f66 6f6e 743e 3c2f 613e 3c2f ..../font></a></
0x0040  7464 3e3c 7464 2077 6964 7468 3d31 353e td><td.width=15>
0x0050  266e 6273 703b 3c2f 7464 3e3c 7464 2069 &nbsp;</td><td.i
0x0060  643d 3320 6267 636f 6c6f 723d 2365 6665 d=3.bgcolor=#efe
0x0070  6665 6620 616c 6967 6e3d 6365 6e74 6572 fef.align=center
...
0x0370  7562 6d69 7420 7661 6c75 653d 2247 6f6f ubmit.value="Goo
0x0380  676c 6520 5365 6172 6368 2220 6e61 6d65 gle.Search".name
...
1044328495.739318 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 3026 win 56344
    <nop,nop,timestamp 92919833 752104553> (DF)
1044328495.741006 192.150.187.28.1472 > 216.239.51.101.80:
    F 81:81(0) ack 3026 win 57920
    <nop,nop,timestamp 92919834 752104553> (DF)
1044328495.823516 216.239.51.101.80 > 192.150.187.28.1472:
    . ack 82 win 31856
    <nop,nop,timestamp 752104562 92919834> (DF)
```

Replacing data entity with MD5 hash value:

```
1044328495.549695 192.150.187.28.1472 > 216.239.51.101.80:
    S 1352447574:1352447574(0) win 57344
    <mss 1460,nop,wscale 0,nop,nop,timestamp 92919815 0>
1044328495.632608 216.239.51.101.80 > 192.150.187.28.1472:
    S 3009119707:3009119707(0) ack 1352447575 win 1460
    <mss 1460,nop,nop,timestamp 752104543 92919815,nop,wscale 0>
1044328495.632647 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 1 win 57920
    <nop,nop,timestamp 92919823 752104543>
1044328495.632966 192.150.187.28.1472 > 216.239.51.101.80:
    P 1:130(129) ack 1 win 57920
    <nop,nop,timestamp 92919823 752104543>
0x0030  2cd4 345f 4745 5420 2f20 4854 5450 2f31 ,.4_GET./.HTTP/1
0x0040  2e30 0d0a 5553 4552 2d41 4745 4e54 3a20 .0..USER-AGENT:.
0x0050  5767 6574 2f31 2e35 2e33 0d0a 484f 5354 Wget/1.5.3..HOST
0x0060  3a20 7777 772e 676f 6f67 6c65 2e63 6f6d :.www.google.com
0x0070  3a38 300d 0a41 4343 4550 543a 202a 2f2a :80..ACCEPT:.*/*
0x0080  0d0a 0d0a 582d 4163 7475 616c 2d44 6174 ....X-Actual-Dat
0x0090  612d 4c65 6e67 7468 3a20 303b 2067 6170 a-Length:.0;.gap
0x00a0  3d30 2c20 636f 6e74 656e 742d 6c65 6e67 =0,.content-leng
0x00b0  7468 3d0d 0a                            th=..
1044328495.716691 216.239.51.101.80 > 192.150.187.28.1472:
    . ack 130 win 30660
    <nop,nop,timestamp 752104551 92919823>
1044328495.737787 216.239.51.101.80 > 192.150.187.28.1472:
    P 1:371(370) ack 130 win 31856
    <nop,nop,timestamp 752104553 92919823>
0x0030  0589 d80f 4854 5450 2f31 2e30 2032 3030 ....HTTP/1.0.200
0x0040  204f 4b0d 0a43 6f6e 7465 6e74 2d4c 656e .OK..Content-Len
0x0050  6774 683a 2033 320d 0a58 2d41 6374 7561 gth:.32..X-Actua
0x0060  6c2d 4461 7461 2d4c 656e 6774 683a 2032 l-Data-Length:.2
0x0070  3730 393b 2067 6170 3d30 2c20 636f 6e74 709;.gap=0,.cont
0x0080  656e 742d 6c65 6e67 7468 3d20 3237 3039 ent-length=.2709
0x0090  0d0a 434f 4e4e 4543 5449 4f4e 3a20 436c ..CONNECTION:.Cl
0x00a0  6f73 650d 0a53 4552 5645 523a 2047 5753 ose..SERVER:.GWS
0x00b0  2f32 2e30 0d0a 4441 5445 3a20 5475 652c /2.0..DATE:.Tue,
0x00c0  2030 3420 4665 6220 3230 3033 2030 333a .04.Feb.2003.03:
0x00d0  3134 3a35 3520 474d 540d 0a43 4f4e 5445 14:55.GMT..CONTE
0x00e0  4e54 2d54 5950 453a 2074 6578 742f 6874 NT-TYPE:.text/ht
0x00f0  6d6c 0d0a 4341 4348 452d 434f 4e54 524f ml..CACHE-CONTRO
0x0100  4c3a 2070 7269 7661 7465 0d0a 5345 542d L:.private..SET-
0x0110  434f 4f4b 4945 3a20 5052 4546 3d49 443d COOKIE:.PREF=ID=
0x0120  3165 3863 3735 3862 3166 3239 6538 3664 1e8c758b1f29e86d
0x0130  3a54 4d3d 3130 3434 3332 3834 3935 3a4c :TM=1044328495:L
0x0140  4d3d 3130 3434 3332 3834 3935 3a53 3d66 M=1044328495:S=f
0x0150  3834 4d67 5379 4833 4745 2d34 3950 703b 84MgSyH3GE-49Pp;
0x0160  2065 7870 6972 6573 3d53 756e 2c20 3137 .expires=Sun,.17
0x0170  2d4a 616e 2d32 3033 3820 3139 3a31 343a -Jan-2038.19:14:
0x0180  3037 2047 4d54 3b20 7061 7468 3d2f 3b20 07.GMT;.path=/;.
0x0190  646f 6d61 696e 3d2e 676f 6f67 6c65 2e63 domain=.google.c
0x01a0  6f6d 0d0a 0d0a                          om....
1044328495.737987 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 371 win 57920
    <nop,nop,timestamp 92919833 752104553>
1044328495.739267 216.239.51.101.80 > 192.150.187.28.1472:
    FP 371:403(32) ack 130 win 31856
    <nop,nop,timestamp 752104553 92919823>
0x0030  0589 d80f 3836 3731 3139 3239 3432 3635 ....867119294265
0x0040  6533 6634 3435 3730 3863 3366 6366 6232 e3f445708c3fcfb2
0x0050  3134 3466                               144f
1044328495.739318 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 404 win 56344
    <nop,nop,timestamp 92919833 752104553>
1044328495.741006 192.150.187.28.1472 > 216.239.51.101.80:
    F 130:130(0) ack 404 win 57920
    <nop,nop,timestamp 92919834 752104553>
1044328495.823516 216.239.51.101.80 > 192.150.187.28.1472:
    . ack 131 win 31856
    <nop,nop,timestamp 752104562 92919834>
```

Substituting “Google” with “Goooogle”:

```
1044328495.737787 216.239.51.101.80 > 192.150.187.28.1472:
    P 1:373(372) ack 130 win 31856
    <nop,nop,timestamp 752104553 92919823>
0x0030  0589 d80f 4854 5450 2f31 2e30 2032 3030 ....HTTP/1.0.200
0x0040  204f 4b0d 0a43 6f6e 7465 6e74 2d4c 656e .OK..Content-Len
0x0050  6774 683a 2032 3731 370d 0a58 2d41 6374 gth:.2717..X-Act
0x0060  7561 6c2d 4461 7461 2d4c 656e 6774 683a ual-Data-Length:
0x0070  2032 3730 393b 2067 6170 3d30 2c20 636f .2709;.gap=0,.co
0x0080  6e74 656e 742d 6c65 6e67 7468 3d20 3237 ntent-length=.27
0x0090  3039 0d0a 434f 4e4e 4543 5449 4f4e 3a20 09..CONNECTION:.
...
1044328495.739267 216.239.51.101.80 > 192.150.187.28.1472:
    P 373:1821(1448) ack 130 win 31856
    <nop,nop,timestamp 752104553 92919823>
...
0x0080  3838 3539 2d31 223e 3c74 6974 6c65 3e47 8859-1"><title>G
0x0090  6f6f 6f6f 676c 653c 2f74 6974 6c65 3e3c oooogle</title><
...
0x0230  743d 3131 3020 616c 743d 2247 6f6f 6f6f t=110.alt="Goooo
0x0240  676c 6522 3e3c 2f74 643e 3c2f 7472 3e3c gle"></td></tr><
...
1044328495.739267 216.239.51.101.80 > 192.150.187.28.1472:
    F 1821:3090(1269) ack 130 win 31856
    <nop,nop,timestamp 752104553 92919823>
...
0x0230  7574 2074 7970 653d 7375 626d 6974 2076 ut.type=submit.v
0x0240  616c 7565 3d22 476f 6f6f 6f67 6c65 2053 alue="Goooogle.S
0x0250  6561 7263 6822 206e 616d 653d 6274 6e47 earch".name=btnG
...
0x04c0  7079 3b32 3030 3320 476f 6f6f 6f67 6c65 py;2003.Goooogle
0x04d0  3c2f 666f 6e74 3e3c 666f 6e74 2073 697a </font><font.siz
...
```