# The Devil and Packet Trace Anonymization*

Ruoming Pang†, Mark Allman‡, Vern Paxson‡,¶, Jason Lee¶
†Princeton University, ‡International Computer Science Institute,
¶Lawrence Berkeley National Laboratory (LBNL)

## ABSTRACT

Releasing network measurement data—including packet traces—to the research community is a virtuous activity that promotes solid research. However, in practice, releasing anonymized packet traces for public use entails many more vexing considerations than just the usual notion of how to scramble IP addresses to preserve privacy. Publishing traces requires carefully balancing the security needs of the organization providing the trace with the research usefulness of the anonymized trace. In this paper we recount our experiences in ($i$) securing permission from a large site to release packet header traces of the site's internal traffic, ($ii$) implementing the corresponding anonymization policy, and ($iii$) validating its correctness. We present a general tool, `tcpmkpub`, for anonymizing traces, discuss the process used to determine the particular anonymization policy, and describe the use of metadata accompanying the traces to provide insight into features that have been obfuscated by anonymization.

## 1. INTRODUCTION

Sharing of network measurement data such as packet traces has been repeatedly identified as critical for solid networking research [4, 17]. Sharing datasets allows: ($i$) verification of previous results, ($ii$) direct comparison of competing ideas on the same data, and ($iii$) a broader view than a single investigator can likely obtain on their own. Various organizations do in fact release measurement data on a regular basis—e.g., NLANR's PMA packet traces [2] and CAIDA's *skitter* [3] measurements. However, when we recently endeavored to publicly release a set of packet header traces of LBNL's internal traffic, we unexpectedly encountered two key problems: ($i$) we found no carefully crafted guidance on anonymization policy for traces meant for public release above and beyond how to strip out payloads and transform IP addresses, and ($ii$) after developing an anonymization policy, we could not find tools we could adapt to transform our traces according to our particular policy or validate the results.

While there has been solid work devising techniques to anonymize IP addresses (e.g., [23]), we found these *just the beginning* of the work involved in preparing traces for release. Indeed, "the devil is in the details" regarding how to treat additional packet header fields, and, more generally, identifying and resolving the numerous considerations that arise when designing an anonymization policy. As an example, [12] demonstrates a technique that leverages TCP timestamps to fingerprint a physical host based on the host's clock drift. An attacker could use legitimate traffic to the site in question to fingerprint machines and then unmask the obscured IP addresses in the released traces by comparing the clock drift in their probes with the clock drift shown by the TCP timestamp options. (Our method for dealing with TCP timestamps is outlined in § 3.4.) While such devil-ish considerations can be readily dealt with by brusquely scrubbing detail from a trace, we know from experience that such scrubbing can often thwart researchers in their investigations due to the lack of key information in the traces. For example, *tcpdpriv* [15] removes TCP options from anonymized traces, thus closing the door to the physical fingerprinting threat mentioned above. However, this not only renders the trace useless to a researcher studying a given option, but also reduces the ability for other researchers to solve *puzzles* found in the traces (such as by using TCP timestamps to accurately pair up packets with their acknowledgments). Finally, we note that while we leverage previous work on IP address anonymization, we also contribute new wrinkles in terms of transforming enterprise addresses and also addresses probed by scanners (detailed in § 3.3).

In anonymizing our traces we endeavored to define a policy that balances the security and privacy needs of the organization providing the trace with the research value that is inevitably reduced with each transformation of the trace. As noted in [23], no perfect anonymization scheme exists and therefore as in much of the security arena, anonymization of packet traces is about managing risk. After arriving at an acceptable anonymization policy we looked for an appropriate tool with which to implement our transformations. None of the anonymization tools we found—including *tcpdpriv* [15], *ipsumdump* [10] and *tcpurify* [6]—were general enough to allow for the easy implementation of a multifaceted anonymization policy across protocol layers. Rather than inserting messy hacks into existing tools or creating yet another custom anonymizer to implement our own particular policy, we opted to develop a tool that provides a general framework for anonymizing traces that can accommodate a wide range of policy decisions and protocols. We describe our tool, `tcpmkpub`, in more detail in § 2 and have released it on our project web page (along with 11 GB of anonymized packet traces of LBNL's enterprise traffic) [1].

While our goal is to preserve as much as possible within the released traces, inevitably we had to obfuscate or completely strip out valuable information. In addition, analysis of packet traces often requires more contextual information than that found within the trace itself (e.g., the gateway IP address associated with a given subnet). Therefore, in addition

*Computer Communication Review, January 2006.

| Section | Meta-Data |
|---|---|
| § 3.1 | Packets found in the original trace with bad checksums are flagged in the meta-data, with a version of the packet with a bad checksum placed in the anonymized trace. |
| § 3.1 | Truncated packets found in the original trace are noted in the meta-data. The packet inserted into the anonymized trace has a corrected checksum based on the sanitized packet. |
| § 3.2 | The meta-data includes a rough frequency table of Ethernet vendor codes. |
| § 3.3 | The meta-data contains a list of the anonymized prefix and size of each internal subnet found in the trace, along with the subnet's gateway and broadcast addresses. |
| § 3.3 | The anonymized IP address of detected scanners is included in the meta-data. The anonymization maps addresses for the target in traffic involving scanners differently than addresses in non-scanning traffic. |
| § 3.3 | The meta-data lists addresses that are part of LBNL's address space, but not from a valid LBNL subnet. |
| § 3.4 | Hosts for which `tcpmkpub` could not determine the endianness of TCP's timestamp option are flagged in the meta-data. The order of the timestamps for these hosts is based on the order in which the packets arrive at the tracing location, rather than the time at which they were transmitted. |
| § 6 | The meta-data gives the number of packets completely removed from the traces due to policy considerations. |
| § 6 | The meta-data includes a tag indicating the anonymization key used to conduct the transformations. All traces with the same tag are uniformly anonymized. |
| § 6 | The meta-data includes a checksum digest of the anonymized packet trace to ensure that the traces and meta-data can be properly paired. |

**Table 1: Meta-data accompanying the anonymized traces.**

to a transformed packet trace we provide *meta-data* about each trace to inform further analysis. The meta-data is often *crucial* for understanding the traces and chasing down puzzles they may present. Table 1 gives a summary of the meta-data generated by our tool.

The problem of trace anonymization is broader than just preparing traces for public release. Some organizations require anonymization of any stored traces, even if kept internal. This can require on-line anonymization, which can introduce complexities. We do not address those complexities in this work, since for our task, off-line anonymization suffices. Furthermore, to retain as much research value as possible in the traces, our policy wound up requiring a multi-pass structure (for example, to identify rare items and map them to the same identifier to thwart fingerprinting based on their known scarcity). While on-line anonymization can leverage some of the techniques outlined in this paper, we believe that developing a solid system for on-line anonymization remains an area for future work.

The rest of this paper progresses as follows. In § 2 we outline the anonymization framework and tool we developed. In § 3 we address our analysis of the anonymization issues that arose and the policy developed in conjunction with LBNL's security staff. § 4 briefly examines the impact of anonymization on two particular packet header analyses. § 5 outlines the steps we took to validate that our anonymization process was in fact accurately transforming the trace without leaking information. § 6 discusses additional considerations that are broader than the contents of the traces. § 7 presents final thoughts.

## 2. METHODOLOGY

The precise method for anonymizing a packet trace fundamentally depends on *policy* decisions, which in turn depend on the purpose of transforming the trace and the concerns of those whose traffic appears in the trace. For instance, for use within an organization a policy may be as simple as removing the application payload from traces, while for traces released to the public, overwriting or transforming portions of the headers is also likely required.

The available anonymization tools we found focus on only the header fields to be changed, primarily the IP addresses. However, we wanted to achieve a balance between obscuring traces enough to provide security and privacy for the monitored network, while at the same time retaining as much information as possible in an effort to not unduly diminish the research value of the traces. We therefore needed an approach that allowed for rich policies that consider each portion of a packet header. To do so, we built `tcpmkpub`, an anonymization tool that provides a generic framework for transforming packet traces based on *explicit rules for each header field*. As illustrated below, `tcpmkpub` provides a platform for users to easily specify, implement, revise, and verify local anonymization policies for a large range of protocols.

Figure 1 shows an example specification for anonymizing an IP header according to a particular policy. The figure illustrates several aspects of our framework. First, note that the specification shown covers every field of an IP header, and thus provides `tcpmkpub` the entire mapping from fields to transformation actions.[1] In addition, all the fields must be specified with a name and a length (e.g., the "`IP_tos`" field is 1 byte long) because `tcpmkpub` has no built-in understanding of IP—the length fields are key to `tcpmkpub` being able to find its way through a given packet. `tcpmkpub` also supports variable length fields, such as individual IP or TCP options. The actual size of the variable length fields is determined by the corresponding action functions, which must understand specifics of the protocol in question. The current policy language is, however, not powerful enough for specifying recursive data structure, such as a linked list of protocol options; navigation through such structure is built into the `tcpmkpub` engine. Note that this limitation does not affect the property that the policy controls each data field. Besides providing a flexible platform for anonymization, the structure

[1]Our specification covers only IPv4. An anonymization policy that also wanted to deal with IPv6 [8] would require an additional specification of the IPv6 header format, as well as the anonymization policy for IPv6.

```
FIELD        (IP_verhl,   1,      KEEP)
FIELD        (IP_tos,     1,      KEEP)
FIELD        (IP_len,     2,      KEEP)
FIELD        (IP_id,      2,      KEEP)
FIELD        (IP_frag,    2,      KEEP)
FIELD        (IP_ttl,     1,      KEEP)
FIELD        (IP_proto,   1,      KEEP)
PUTOFF_FIELD (IP_cksum,   2,      ZERO)
FIELD        (IP_src,     4,      anonymize_ip_addr)
FIELD        (IP_dst,     4,      anonymize_ip_addr)
FIELD        (IP_options, VARLEN, anonymize_ip_options)
PICKUP_FIELD (IP_cksum,   0,      recompute_ip_checksum)
FIELD        (IP_data,    VARLEN, anonymize_ip_data)
```

**Figure 1: Specification for IP header anonymization.**

```
CASE (TCPOPT_eol,      0, 1,       KEEP)
CASE (TCPOPT_nop,      1, 1,       KEEP)
CASE (TCPOPT_mss,      2, 4,       KEEP)
CASE (TCPOPT_wsopt,    3, 3,       KEEP)
CASE (TCPOPT_sackperm, 4, 2,       KEEP)
CASE (TCPOPT_sack,     5, VARLEN,  KEEP)
CASE (TCPOPT_tsopt,    8, 10,      renumber_tcp_timestamp)
CASE (TCPOPT_cc,       11, VARLEN, KEEP)
CASE (TCPOPT_ccnew,    12, VARLEN, KEEP)

DEFAULT_CASE (TCPOPT_other, VARLEN, TCPOPT_alert_and_replace_with_NOP)
```

**Figure 2: TCP option anonymization specification.**

of `tcpmkpub` also helps guide data providers to precisely consider each header field, since an action must be assigned to each field.

Next, the user specifies an *action* for each field in the header. Two built-in actions are provided to retain the field's original value in the anonymized trace ("KEEP") and to clear the field's value in the anonymized trace ("ZERO"). The user can also specify C++ function names as actions for richer transformations, including those that require keeping state across multiple packets. For instance, the IP anonymization policy in Figure 1 shows that the "`IP_src`" and "`IP_dst`" fields are transformed by calling the *anonymize_ip_addr()* function. Given that the specification includes the entire packet, modifications are straightforward. For instance, studies have shown how to extract information from the IP ID field [5, 7]; therefore, while not a part of our particular policy, someone sharing a trace might want to obscure that field's value as part of their anonymization policy. This requires changing the action for the "`IP_id`" field from "KEEP" to "ZERO" to simply clear the field. Alternatively, the action could be set to the name of a function to execute to transform the field (e.g., *anonymize_ipid()*), coupled with developing a simple C++ function to randomize or change the IP ID field in whatever fashion the user deems appropriate.

In addition, `tcpmkpub` allows the anonymization process to "go back" to particular header fields. For instance, the "`IP_cksum`" field is initially zeroed and then, after all transformations have been applied to the packet, `tcpmkpub` comes back and computes a new IP checksum and inserts that checksum into the anonymized trace (see § 3.1 for more details about the checksumming process).

The framework also supports *case* statements when header fields can vary. For instance, Figure 2 shows the set of rules for processing TCP options, which may appear in arbitrary order, or not at all. `tcpmkpub` treats options much like standard header fields. In case statements the option name is followed by the "type" code for the option. If the option being processed matches the type code in the anonymization specification, the option is defined by a given length and processed using a given action. For instance, TCP option 2 is an MSS advertisement. The option is 4 bytes long and our policy simply retains the value in the original trace when placing the packet into the anonymized trace. As above, the action can be the name of a C++ function to execute to transform the option. For instance, the *renumber_TCP_timestamp()* function is called to sanitize the TCP timestamp option [9], as discussed further in § 3.4. Finally, a default case covers the situation when a particular option found in a trace is not enumerated in the anonymization policy. The policy employed in the example replaces such options with "NOP" options and inserts an alert into the `tcpmkpub` log file. These alerts are important to monitor because, if frequent, they may indicate a change to the anonymization policy is warranted. For instance, they could indicate increasing prevalence of some newly defined TCP option that could be better dealt with than by simply replacing the option with NOPs.

As the `tcpmkpub` engine possess little knowledge about protocols, a question is how one can check whether the protocol specification in anonymization policy is correct and complete. One way to catch such errors is through self-checking. The action functions can raise alerts when some field value looks suspicious, e.g., when encountering an undefined TCP option. Further, for constant (or constant-ranged) fields, one can employ a constant checker as the action (even if the field is not transmformed), as in the ARP policy (see Figure 3 at the end of paper)—in fact, this is how we caught the weird ARP packets discussed in the next paragraph.

Finally, `tcpmkpub` provides hooks for additional processing. These include static filtering based on BPF filters (e.g., for excluding a particular host or traffic involving a sensitive port) and *packet-specific* policies. For example, one policy we use contains entries that identify ARP packets with specific timestamps and payload contents. These packets contain the bizarre string "`Move to 10mb on D3-packet`," in a portion of

the ARP packet that is normally cleared by our default policy. However, these packets have been manually vetted and are not contrary to our anonymization policy; thus, we explicitly preserve the payload of these packets as in the original trace, since such real-life packet "crud" can be important for capturing the diversity present in actual network traffic.

# 3. ANONYMIZATION POLICY

In this section we sketch the anonymization policy we arrived at and the thinking that led to it. In the current work, our focus is on traces that include only packet headers,[2] though in the future our project intends to build on [16] and release traces with anonymized payloads. We do not advocate the policy outlined in this paper as *the correct policy*, but as *a possible policy*, with the goal being to discuss items to consider when determining policy. In addition, we discuss alternatives in this section that we considered and may well represent a better approach in some environments. Particular items that need thought when developing an anonymization policy are IP addresses, the IP ID field, TCP sequence numbers, length fields, and transport protocol port numbers, as discussed below.

We first consider the site's "threat model" for releasing such traces. It is crucial to prevent users of the trace files from determining: ($i$) identities of specific hosts such that an audit trail could be formed about particular users, ($ii$) identities of internal hosts such that a map could be constructed of which hosts support which services (which could be used in mounting an attack), and ($iii$) security practices of the organization that an attacker would not otherwise know and could leverage during an attack.

We next discuss our anonymization policy, starting with how to handle checksums across protocol layers; then we follow the protocol stack to examine policies for each protocol layer. This section provides examples of our anonymization policy files. See Figure 3 at the end of this paper for a listing of all the policy specifications used to implement our policy. The policy files will also be included with the `tcpmkpub` release at [1].

## 3.1 Checksums

One aspect of transforming packet traces that crosses layers and protocols is calculating various checksum fields. We re-calculate checksums in the anonymized traces for two reasons: ($i$) even when application-layer data is removed from packets the checksum can sometimes give away the contents of the data (e.g., for small packets) and ($ii$) since we remove application payloads and transform various header fields in the packets the users of the traces will not be able to determine if the original checksums were valid. As noted in [14], hunting for checksum failures in packet traces can be important when analyzing rare events.

Our technique involves replacing the original checksum, $C_o$, with a checksum $C_c$ calculated across only the transformed bytes that are being placed in the anonymized packet trace. There are two reasons we may not be able to verify $C_o$: ($i$) the packet has been corrupted while traversing the network or ($ii$) the original packet trace did not capture enough of the packet to allow us to independently compute the checksum (e.g., because some of the payload is missing). In the first case, we insert "1" into the appropriate checksum field to mark the packet as having a known failed checksum originally (unless $C_c$ happens to yield 1 itself, in which case we insert "2"). This guarantees that a researcher verifying the checksums in the anonymized trace will observe a failure, as in the original trace. On the other hand, for packets for which we cannot verify $C_o$ due to packet truncation in the trace, we assume valid checksums and include $C_c$ in the anonymized trace. We also note corrupted and truncated packets in the meta-data.

Finally, we need to consider the fact that UDP checksums are optional. If the checksum is zero in the original trace, we preserve this in the anonymized trace[3].

We note that an alternative method would be one of the approaches implemented in *tcpurify* [6], which replaces checksums with codes indicating "valid original", "invalid original", or "not enough of the packet captured to determine". That scheme has the advantage of not requiring separate meta-data, but requires analysis tools to understand the codes.

## 3.2 Link Layer

At first blush, the Ethernet header might not seem sensitive. On their own, Ethernet addresses do not give away much information since they are chosen essentially randomly by vendors. However, because Ethernet addresses are distinct to individual NICs, retaining them in the traces would allow attackers to uncover the actions of a given user if they separately obtain the MAC address of the user's NIC. If they also determine the associated non-anonymized IP address, they then can spot instances of the MAC address in the traces and use this information to work on unraveling the IP address anonymization scheme.

We consider three different methods of randomizing Ethernet addresses to counter these threats: ($i$) scrambling the entire 6 byte address, ($ii$) scrambling only the lower 3 bytes of the address, preserving the "vendor code" in the upper 3 bytes, or ($iii$) scrambling the vendor code and the lower 3 bytes independently. Mapping the entire 6 byte address would remove the ability of researchers to attribute various oddities (for example, replicated packets) to NICs from particular vendors. We could retain this facet of the trace data by preserving the vendor ID and scrambling only the lower 3 bytes. While this approach maintains potentially useful information about the NIC vendor, it fails to preserve anonymity if some vendors have only a small number of NICs in the site providing the trace—if the attacker separately learns about these rarely used devices, they can locate them in the trace based solely on their rare vendor ID.

These considerations led us to the third option, remapping the high- and low-order 3 bytes separately. This allows the trace user to find all hosts using the same NIC vendor, but not to identify that NIC or the original full address. Our specific scheme remaps the high-order 3 bytes and uses that value as the seed for remapping the low-order 3 bytes. Doing so produces a consistent mapping across multiple traces. Therefore, say the low-order 3 bytes $X$ map to $X'$ for vendor $Y$. For vendor $Z$ the same $X$ will map to some $X''$. Finally, we include in the meta-data a rough frequency table of unanonymized vendor IDs found in our traces (e.g., a list of vendor IDs with 1–20 hosts, 20–50 hosts, 50–200 hosts, etc.), in an attempt to pre-

[2]The only payloads we include are packet headers encapsulated within ICMP messages and ARP payloads (with renumbered addresses).

[3]Per the UDP specification [18], calculated values of zero are replaced with the equivalent 0xffff.

serve a profile of the diversity of NICs in use at the site. The bucket ranges are carefully chosen as to not finger particular machines by virtue of being the only address in a particular bucket.

Ethernet addresses not only appear in Ethernet headers, but also in the contents of ARP packets, and our framework understands the ARP packet format and consistently remaps these internal addresses, as well.

There are exceptions to the remapping policy. We preserve addresses that are all zeros (*unknown MAC* in ARP packets) or all ones (broadcast traffic), and also the "multicast bit" in the high-order 3 bytes.

Our analysis of the other Ethernet header fields concluded that they do not pose any anonymization issues. At this point, `tcpmkpub` inspects the type of header following the Ethernet header. The policy we use understands IP and ARP packets, so for these it proceeds to further anonymization. For all other packet types, it truncates the packet placed in the anonymized trace after the Ethernet header.

## 3.3 Network Layer

Obviously, a key aspect to our policy at the network layer is anonymizing IP addresses. If an attacker can tie traffic to a known IP address and thereby potentially to a user, they can attain a detailed accounting of the user's activities (violating privacy, and possibly embarrassing the site if the user's activities are inappropriate). In addition, an attacker could use information about services running on a particular host to develop an attack plan. We therefore seek to obscure the IP addresses. While IP address anonymization is well trod ground (e.g., based on [23]), we found that the devil again showed up and we needed to add a few wrinkles to implement a sound policy within our environment.

In particular, we remap addresses differently based on the type of address. The following details our anonymization policy for various types of addresses and distills the meta-data we record to retain as much research value as possible. For the purposes of our discussion, "internal" addresses are those allocated to LBNL and "external" addresses are non-LBNL addresses.

**External addresses:** remapped using the prefix-preserving address anonymization scheme given in [23]. While this scheme can be attacked, the site's view is that the difficulty of attacking it for external addresses, which have much less locality than internal addresses, suffices to reduce the threat to an acceptable level.

**Internal addresses:** processed in two steps: first, the prefix part is mapped to a prefix unused by the prefix-preserving scheme for external addresses and then the subnet and host portions of the address are transformed. It is important to note that *we do not retain the prefix-preserving relationship between internal and external addresses*. If we did, then because the organization from which the trace comes is known, the prefix-preserving property could be used to infer portions of external addresses adjacent to internal addresses. For instance, one of LBNL's address ranges is 128.3.0.0/16. However, since the trace is known to be from LBNL, even if we transformed "128.3", it seems safe to assume that it would not be difficult to determine which traffic is from LBNL. Therefore, by including LBNL's addresses in the prefix-preserving address anonymization used for external addresses, any address whose first octet is 128 would be partially unmasked.

Therefore, after the prefix-preserving algorithm has classified all external IP addresses in the trace we map the internal addresses to an unused part of the global address space.[4] The meta-data provides a list of internal network prefixes. This aspect of anonymization requires two passes at the original packet trace, first to construct a collision-free map of IP addresses, and second to actually anonymize the addresses. We note that given the multi-pass nature of our technique, this aspect of IP address anonymization would require a different approach for on-line anonymization. We also note that mapping internal addresses separately can lead to inconsistencies across traces. For instance, consider the case when we take a trace $T_0$ today, anonymizing and releasing it with internal addresses in prefix $P_0$. Further, assume we anonymize a second trace, $T_1$, at some point later, using the same key to provide uniformity across the traces (see § 6 for more on uniform anonymization). While anonymizing $T_1$, an external address may map onto $P_0$, and therefore we must use a different internal prefix, $P_1$, for internal addresses. Therefore, while most of the anonymization is uniform across the two traces, the consistency is marred by the fact that the internal prefixes differ across the two collections.

Second, the mapping of subnet and host portions of internal addresses is *not bitwise prefix-preserving*. Instead we remap the subnet and host portions of internal addresses independently and preserve only whether two addresses belong to the same subnet. Therefore, all hosts appearing in some subnet $X$ in the original trace will appear in the corresponding subnet $X'$ in the anonymized trace. This random mapping does not preserve the relationship between subnets in the internal network. For instance, if two /24 subnets share a /20 prefix in the original trace, they will not necessarily do so in the anonymized trace. The meta-data contains a list of the (renumbered) internal subnets. In addition, the meta-data contains the remapped gateway and broadcast addresses for each internal subnet. We remap the host portions differently for each subnet.

In remapping host portions within a subnet, we need to compute a pseudo-random permutation among addresses. With the algorithm described in [13], the permutations depend only on the cryptographic key, thus we can keep the mapping independent of the order in which the addresses appear and consistent across multiple traces, without having to store the mapping, analogous to the properties of the algorithm for prefix-preserving anonymization [23].

Remapping the subnets also involves computing a pseudo-random permutation, except that the subnets can have different prefix lengths. Thus we map bigger subnets (with shorter prefixes) before smaller subnets. The mapping likewise depends only on the cryptographic key.

**Multicast addresses:** preserved in the anonymized trace, as they do not identify any particular host.

**Private addresses:** preserved in the anonymized trace because they do not convey a sense of identity in LBNL's environment, due to how they are used and allocated. Note that in other environments, private addresses could very well convey a sense of identity. For instance, a particular portion of the network might employ a rarely used portion of private address space (e.g., 10.55.100.0/24) and therefore the private addresses could be easily linked with users.

[4] In practice, we use one of the organization's standard prefixes unless that prefix was used for some external address.

**Scanners.** A particular problem with our anonymization techniques concerns traffic from scanners that probe a wide swath of the IP address space. For instance, many organizations run a scanner to check various properties of the internal hosts as part of their security operation. These probes tend to hit addresses in a well established order such as $a.b.c.1$, $a.b.c.2$, $a.b.c.3$, etc. When we anonymize addresses, the host portion of the address is randomized. But because these sorts of scanners are easy to pick out by their rapid (and frequently unsuccessful) connection attempts, by observing the order hosts are probed by such scanners, an attacker might approximately derive the original host portion of the IP addresses, and also possibly the subnet prefix. Also note that the DNS is a readily accessible database of the live hosts at an organization, which an attacker may leverage to assist in unmasking relationships between populated addresses.

In addition to IP-level (or higher) internal or external scanners, we found another subtle scanner in the traces. The enterprise's routers sometimes ARP for an entire subnet in rapid-fire fashion, which we attribute to initializing the router's ARP table, or possibly "host discovery" activity within the subnet. As discussed above, such probes (and their responses) may be used to partially unmask IP addresses, given the timing of the requests. We appreciated this particular threat only late in the process of anonymizing our traces, which serves to (again) highlight the careful diligence required to anonymize packet traces.

Because of the potential threat from scanners, we decided to map addresses relating to scanner activity using a separate namespace than that of non-scanning activity, to break the structural relationship induced by sequential scanners. To do so, however, we need to find the scanners. We did so by looking for hosts that visited more than 20 distinct IP addresses, for which there was a window of 20 IP addresses in which at least 16 were (in the original trace) strictly in ascending or descending order. This is merely a heuristic; however, it has the property that an attacker is unlikely to find and leverage scanners in the anonymized trace that this heuristic misses.

As mentioned above, we renumber the IP addresses involved in scanning traffic separately. We keep the scanner's IP address uniform across the trace, and flag the scanner as such in the meta-data. However, we use a different mapping (resulting in a different subnet and host address) for the destination address of the scans. For instance, consider two hosts $X_1$ and $X_2$ in subnet $Y$ from the original trace file. In traffic *not involving the scanner*, these addresses will be mapped to $X_1'$ and $X_2'$ in subnet $Y'$. For traffic *involving the scanner* these addresses will be mapped to $X_1''$ and $X_2''$ in subnets $Z_1$ and $Z_2$, respectively. This unfortunate inconsistency in the resulting traces means that it becomes impossible to analyze a host's entire set of traffic for any internal address that was scanned. Finally, we note that Ethernet addresses of hosts being scanned also need renumbering, or an attacker can easily establish the mapping between IP addresses for scanning and non-scanning traffic.

The above discussion assumes that the adversary did not scan the network himself during trace collection and has to leverage existing scanning. As pointed out in [23], with *active probing* there are many opportunities for the adversary to "fingerprint" addresses and thus defeat any 1-to-1 address mapping. In that case one solution is to anonymize host identities (including IP and MAC addresses and IP-ID) with a 1-to-n mapping, for example, mapping an address depending on the communication peer's address.

**Invalid addresses.** Our packet traces contain several instances of data transactions involving a host belonging to an invalid subnet (i.e., the organization does not use the particular subnet). That is, the IP address is in the organization's address space, but that particular portion of the address space is meant to be dark. These might come from misconfigurations or users "borrowing" addresses they were not assigned. We anonymize such addresses as though the subnet existed, but note them in the meta-data as not belonging to a valid subnet.

In addition, we found packets in our packet traces that contain IP options that in turn contain IP addresses (e.g., the record route option). We remap the IP addresses contained within these options before placing the packets into the anonymized trace. Likewise, we must remap IP addresses contained within ARP replies.

We note that some of the complications in terms of anonymizing IP addresses come from the fact that we are sanitizing edge-network packet traces. Packet traces taken in the middle of the network would likely not have the same strong address prefix signature that enterprise traces have and therefore may be able to be anonymized without regard to address "type".

The last consideration at the network layer is ICMP traffic. Given ICMP's use for carrying all sorts of rich network status information, we must take care when including such packets in the anonymized traces. ICMP messages often contain the first bytes of the packet that triggered the ICMP message. Therefore, we recursively anonymize the included IP packet as we would any other packet in the original trace.

## 3.4 Transport Layer

Our anonymization policy deals with TCP [19] and UDP [18] at the transport layer. We truncate packets using other transport protocols after the IP header (we did not see significant amounts of such traffic). As outlined in § 2, implementing anonymization frameworks for new transport protocols (e.g., SCTP [21] or DCCP [11]) should be straight-forward.

The first consideration for transport protocols is whether to anonymize the port numbers. Our policy leaves the TCP and UDP port numbers intact, with the exception that we remove traffic involving one particular port used for an internal security monitoring application. A drawback of preserving port numbers is that they may be able to be used to identify a particular machine that runs a particular set of services, if that set is in some way unique (e.g., due to the make-up of the set, traffic volume, etc.).

Another aspect of TCP traffic that potentially leaks information is the sequence number (as well as IP/PCAP length). [22] shows that a motivated attacker can find traffic in an anonymized trace that involves a particular web site by comparing the length of TCP connections in the trace with a database of known object lengths on given web pages. This attack requires significant resources, and therefore for our environment it is not perceived to be a large threat.

Given that we preserve both port numbers and sequence numbers, the most significant transformation we perform at the transport layer is to rewrite TCP timestamp options [9]. Recent work has found that clock drift manifest in timestamp options can be leveraged to fingerprint a physical machine, enabling its unique identification in the future [12]. If a machine

could be fingerprinted using the anonymized traces, then an attacker who also probes the site's hosts directly could pair up the timestamp signatures they obtain from probing with those in the trace, undermining the IP address anonymization. On the other hand, timestamp options have significant utility in analyzing TCP dynamics, as they allow unambiguous matching of data packets with acknowledgments and can help detect packet duplication and reordering.

Therefore, to balance these concerns our policy is to transform the timestamps present in timestamp options into separate monotonically increasing counters with no relationship to time for each IP address appearing in the anonymized trace. We preserve timestamp echoes of zero, which indicate "no timestamp." Much of the research use of timestamps involves using them to determine the *uniqueness* and *transmission order* of segments. A per-host counter preserves this use. Of course, any use of the timestamp option for actual timing information (e.g., investigating TCP's retransmission timeout, or the jitter between packets) is lost. We considered "fuzzing" the timestamps by random amounts, instead of using a counter, to degrade the artifacts used by the fingerprinting scheme. However, since it is not clear how this would affect research relying upon timestamps for timing information, we decided to simply remove all timing information.

Using our approach, transforming a timestamp option requires two passes over the original packet trace, for two reasons. First, RFC 1323 does not specify the actual format of timestamps, nor their endianess. Therefore, to infer the ordering relationship between timestamps (and thus to correctly assign counter values when rewriting them), we need to observe multiple packets to determine endianess. Second, even if we can determine the order among timestamps, it is still problematic to renumber without knowing what timestamps may appear later, so we wait until observing all the timestamps before renumbering them sequentially. In those cases where we cannot determine the endianess of the timestamps, we simply reflect the order of the packets in the original trace. Doing so can aid a researcher interested in determining the uniqueness of packets, but the causal ordering becomes potentially misleading, so we note the failure to identify the endianess for the given host in the meta-data.

# 4. INFORMATION LOSS

As noted above, every transform applied to a trace can potentially perturb analysis of the transformed trace. Given our explicit goal to retain as much research value as possible, we analyzed the original and anonymized traces with two tools that perform packet header analysis and compared the output as one way to gauge how effective we were in preserving information. We stress that these are simply two *examples* and their performance may not be indicative of other uses of the traces.

We first used *p0f* [24] to do OS fingerprinting on the hosts in the trace.[5] We found two relevant differences between the analysis of the original and transformed traces: ($i$) transforming the TCP timestamp option into a counter rendered *p0f*'s "host uptime" analysis useless, and ($ii$) one connection showed a different OS signature in the transformed trace due to a corrupted packet in the original trace causing our anonymization process to change an invalid TCP option into a NOP option. Thus, we conclude that OS fingerprinting is in general still possible with the transformed traces; this is acceptable to our site.

We also used a custom tool, *tcpsum*, to crunch each TCP connection in the trace to find the number of packets and bytes sent in each direction, as well as a crude history of the connection ("saw SYN", "saw SYN+ACK", etc.). Except for IP addresses, the output from crunching the original and transformed traces matched, indicating no value was lost in the transformations for this particular type of analysis.

We again note that our simple tests are not exhaustive. Clearly, the transformations we applied to the traces can have an impact on certain forms of analysis. For instance, any analysis that involves digging into the contents of packets (e.g., for use in developing intrusion detection methodologies) would be rendered useless by our anonymization scheme. However, we believe that these simple tests show that within the realm of header analysis we have preserved much useful information while still protecting the security and privacy of the site and its users.

# 5. VALIDATION

We next turn to a key aspect of implementing an anonymization policy: *validation*. For the set of traces we prepared, we used several *ad hoc* methods to validate that the information we intended to mask was indeed transformed or left out of the anonymized traces:

- First we inspected the log created by `tcpmkpub` during the anonymization process. `tcpmkpub` flags all unexpected aspects of a packet trace it runs across, including, for example, incomplete IP headers or IP addresses (which are possible within ICMP unreachable messages), indeterminable byte order of TCP timestamps for a particular host, or illegal values for fields with constant or limited-ranged values. Examining illegal field values lead us to the discovery of the bizarre ARP packets mentioned in § 2 and TCP options with illegal length fields (e.g., "SACK permitted" options with length 253 instead of 2 and window scale options with length 1 rather than 3).

  While using the tool to verify itself is inherently insufficient, this is a prudent first step to ensure that `tcpmkpub` didn't get confused in a way that would lead to information leakage. We found nothing in our logs that indicated any problems. We base the remainder of our validation, however, on use of separate tools.

- We next used the standard Unix tool *strings* to look for sequences of at least six contiguous letters (case insensitive) in the anonymized traces in an attempt to ensure that packet payloads had been properly removed. When run across the original traces we found many strings that are clearly commands, filenames, etc. (e.g., "Documents", "Settings", "ConfirmFileOp"). However, in looking through the output produced from the anonymized trace we found little that was recognizable as obvious packet content. We manually checked the few strings that remotely resemble words (for instance,

[5] We note that this is an area where some sites may desire that the information *not* appear in the anonymized traces, in which case protocol scrubbing techniques [20] may be beneficial as part of the anonymization process.

"tkirtkis") and found them to be caused by simple coincidence.

- We wrote a small tool to pick through packets and look for 32 bits that looked like IP addresses to ensure that we removed all the LBNL addresses from the data. We first looked for "addresses" with LBNL's prefixes and appearing in both the original and anonymized packets (in either byte order). This procedure produced too many false positives due to a collision between the first octet of one of LBNL's prefixes with a common TCP offset value (which is preserved in anonymization, and thus identical in original and anonymized packets). Therefore, we refined our analysis to ignore certain regions of the packets that we preserve (for example, the TCP sequence numbers), which reduced the number of occurrences to nearly zero; we manually verified the remainder as due to coincidence (for example, in one case the destination address of a packet happened to be mapped to exactly the source address).

- We used *strings* to look for string versions of IP addresses (i.e., dotted-quads) that matched an LBNL prefix. We found no matches.

- We next focused on ensuring that `tcpmkpub` accurately transformed MAC addresses. First, we used *tcpdump* to generate a list of all MAC addresses found in our original traces. We wrote a small *flex* program to pick through the anonymized traces looking for the 6 byte MAC addresses found in the original trace files. We manually compared the hits from the anonymized traces with the original traces, which determined all were coincidence.

- Finally, we used *ipsumdump* to dump TCP options from our anonymized traces. From this we picked out the timestamps, produced sorted lists, and verified that all hosts started with a timestamp of zero and increased from that point. Therefore, we conclude that our timestamp re-numbering appears accurate.

The *ad hoc* validation we conducted convinced us that our anonymized traces are sufficiently safe to release. However, an area for beneficial future work is to write an independent tool that vets anonymized traces against a given policy, which would both improve the quality of the validation and make it easier to conduct.

# 6. ADDITIONAL CONSIDERATIONS

Along with the devil-ish details we describe above, there are several additional issues to consider.

**Traffic removal.** Some traffic in the traces could simply be too sensitive or unique to a particular institution to include in the anonymized traces. For instance, as mentioned above we removed all traffic on a particular TCP port because the traffic involves a custom application used for security operations within the site. For some analyses, the missing traffic will have little impact. However, for other analyses the missing traffic *could* lead to an invalid conclusion (e.g., that a network was not congested when it really was). We suggest that the characteristics of removed traffic be provided in the meta-data in high-level terms, so researchers using the data will at least be aware of the amount of traffic culled from the traces. At a minimum, the meta-data should contain an absolute count of the number of packets removed from the traces. (The number of removed packets in the LBNL traces is about 0.01% of total number of packets.)

An alternative to traffic removal would be to truncated packets after the ethernet or IP headers rather than completely removing the packets. Arguably, removal offers little additional benefit and some additional cost and diminishes the research value of the traces. However, we found that in getting approval for our anonymization scheme we needed to pick our battles and appreciate that removal is sometimes simply more appealing than scrubbing for extremely sensative information.

**Filenames.** The contents of a packet trace are not the only source of information leaks. While the particular naming used for the files of the traces seems like a mundane detail, naming conventions for can potentially leak information to an adversary, e.g., "server-room-trace.dmp".

**Uniform anonymization.** We suggest that traces anonymized in a uniform manner (e.g., the same IP address mapping) should contain a common tag in the various meta-data files to enable researchers to correlate information across the traces. In general, providing consistent anonymization across multiple traces is a two-edged sword: it preserves greater research utility, but at the cost of providing attackers with more data to use in attempting to subvert the anonymization process.

**Linking traces to meta-data.** We suggest a solid linking between a trace and its meta-data by inserting secure checksum digest of the trace in the meta-data, so that researchers can verify they are matching specific meta-data to the right trace.

**Performance.** On a FreeBSD system with a 2.2 GHz Intel Xeon processor and 2 GB of RAM `tcpmkpub` processes the LBNL traces we released in 2.9 hours, using a maximum of 331 MB of memory. The traces contain 165 million packets and the original files add to 48 GB.

**Detecting leakage.** Being able to detect if a trace's anonymization has been compromised after release could prove important. We have devised such methods; however, they either skew the traffic characteristics in the anonymized trace or could be trivially circumvented if the defense was generally known. The design of techniques to robustly detect anonymization compromise remains an interesting area for future work.

**Situational considerations.** Some of the aspects of packet trace anonymization discussed in this paper may be more or less important in certain situations. Different approaches may prove desirable depending on the traffic being traced, the vantage point of the traffic collector, or the portion of the network monitored. For instance, when anonymizing a backbone packet trace the special handling of scanning traffic discussed in § 3.3 is likely not required. This (again) underscores the importance of carefully considering all aspects of anonymization within the context of the local environment.

**The devil we have yet to meet.** If the attack in [12] had been discovered a year later, we would have preserved TCP timestamps in our released traces, leaving them potentially vulnerable. Unfortunately, it is not clear to us how to *systematically* defend against unknown attacks. Therefore, it is important that anonymization policies are periodically evaluated and evolve over time. We also note that applying future attacks to past traces may not be a fruitful endeavor. For instance, the TCP timestamp attack would be harder to mount if

there was some turnover in hosts or IP address renumbering.

## 7. SUMMARY AND FUTURE WORK

This paper endeavors to make four contributions: First, we enumerate and explore many of the devil-ish details involved in preparing packet traces for public release that go beyond the well-known topic of IP address obfuscation. Second, we sketch the use of meta-data to help researchers using anonymized traces to cope with the information lost during the anonymization process. Third, we developed a tool, `tcpmkpub`, and a framework for implementing arbitrary anonymization policy in a straightforward, comprehensible fashion. Our tools and traces are publicly available via [1]. Additionally, Figure 3 shows the complete anonymization specification for the policy we employ. Finally, we have introduced new wrinkles to address anonymization, such as mapping scanner traffic differently from non-scanner traffic, mapping internal addresses differently from external addresses, and mapping the two halves of Ethernet addresses separately. We stress that the decisions outlined in this paper should not be considered *the* right approach, but rather a heavily considered approach that currently meets the needs for releasing traces from a particular network.

There are a number of avenues for fruitful future work in the area of packet trace anonymization. As discussed above, tools to aid with validating that trace files have been appropriately scrubbed would be useful in increasing data provider's confidence in the anonymization process. In addition, studying the tradeoffs required to conduct on-line anonymization is an area that would likely have significant benefit. Also, robust schemes for detecting when a trace has been compromised would be highly useful in providing operators with situational awareness. Finally, there is a huge temptation to put together a system that can take high-level input from a user and produce an anonymization policy for `tcpmkpub`, given the complexity of the process of setting up and evaluating the procedures. It is not clear to us that this is possible to do if one actually cares about the quality of the results. However, a useful area of future work may be in exploring such a system, including both its value and its limitations.

## Acknowledgments

This work was supported as part of the DHS PREDICT project under grant HSHQPA4X03322 as well as NSF grant 0335214. Our thanks to the many LBNL staff members who made this work possible; in particular, Mike Bennett, Jim Mellander, Sandy Merola, Dwayne Ramsey and Brian Tierney. We thank Ethan Blanton for numerous discussions on the topics covered in this paper. Our thanks to Martin Casado and the anonymous IMC 2005 and CCR reviewers for providing useful comments.

## 8. REFERENCES

[1] Enterprise tracing project. http://www.icir.org/enterprise-tracing/.

[2] The Passive Measurement and Analysis Project. http://pma.nlanr.net/.

[3] The Skitter Project. http://www.caida.org/tools/measurement/skitter/.

[4] M. Allman, E. Blanton, and W. Eddy. A Scalable System for Sharing Internet Measurements. In *Passive and Active Measurement Workshop*, Mar. 2002.

[5] S. Bellovin. A Technique for Counting NATted Hosts. In *Proceedings of the Internet Measurement Workshop*, Nov. 2002.

[6] E. Blanton. tcpurify, May 2004. http://irg.cs.ohiou.edu/~eblanton/tcpurify/.

[7] W. Chen, Y. Huang, B. Ribeiro, K. Suh, H. Zhang, E. de Souza e Silva, J. Kurose, and D. Towsley. Exploiting the IPID Field to Infer Network Path and End-System Characteristics. In *Proceedings of the Passive and Active Measurement Workshop*, Mar. 2005.

[8] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specifi cation, Jan. 1996. RFC 1883.

[9] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

[10] E. Kohler. ipsumdump. http://www.cs.ucla.edu/~kohler/ipsumdump/.

[11] E. Kohler, M. Handley, and S. Floyd. Datagram Control Protocol (DCCP), Mar. 2005. Internet-Draft draft-ietf-dccp-spec-11.txt (work in progress).

[12] T. Kohno, A. Broido, and kc claffy. Remote Physical Device Fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[13] M. Luby and C. Rackoff. Pseudo-random permutation generators and cryptographic composition. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 356–363, New York, NY, USA, 1986. ACM Press.

[14] A. Medina, M. Allman, and S. Floyd. Measuring the Evolution of Transport Protocols in the Internet. *ACM Computer Communication Review*, 35(2), Apr. 2005.

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

[16] R. Pang and V. Paxson. A High-Level Programming Environment for Packet Trace Anonymization and Transformation. In *ACM SIGCOMM*, Aug. 2003.

[17] V. Paxson. Strategies for Sound Internet Measurement. In *ACM SIGCOMM Internet Measurement Conference*, Oct. 2004.

[18] J. Postel. User Datagram Protocol, Aug. 1980. RFC 768.

[19] J. Postel. Transmission Control Protocol, Sept. 1981. RFC 793.

[20] M. Smart, G. R. Malan, and F. Jahanian. Defeating TCP/IP Stack Fingerprinting. In *9th USENIX Security Symposium*, pages 229–240, 2000.

[21] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol, Oct. 2000. RFC 2960.

[22] Q. Sun, D. R. Simon, Y. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu. Statistical Identifi cation of Encrypted Web Browsing Traffi c. In *IEEE Symposium on Security and Privacy*, May 2002.

[23] J. Xu, J. Fan, M. H. Ammar, and S. B. Moon. Prefi x-Preserving IP Address Anonymization: Measurement-Based Security Evaluation and a New Cryptography-Based Scheme. In *Proceedings of the 10th IEEE International Conference on Network Protocols*, pages 280–289, Washington, DC, USA, 2002. IEEE Computer Society.

[24] M. Zalewski. p0f: Passive OS Fingerprinting tool. http://lcamtuf.coredump.cx/p0f.shtml.

```
// ether.anon
FIELD        (ETHER_dstaddr,                        6,       anonymize_ethernet_addr)
FIELD        (ETHER_srcaddr,                        6,       anonymize_ethernet_addr)
FIELD        (ETHER_lentype,                        2,       KEEP)
FIELD        (ETHER_data,                           VARLEN,  anonymize_ethernet_data)

// ether-data.anon
CASE         (ETHERDATA_ip,       0x0800,           VARLEN,  anonymize_ip_pkt)
CASE         (ETHERDATA_arp,      0x0806,           VARLEN,  anonymize_arp_pkt)
DEFAULT_CASE (ETHERDATA_other,                      VARLEN,  other_ethertnet_pkt_alert_and_skip)

// arp.anon
FIELD        (ARP_hrd,                              2,       const_n16 (0x0001, BREAK))
FIELD        (ARP_pro,                              2,       const_n16 (0x0800, BREAK))
FIELD        (ARP_hln,                              1,       const_n8 (6, BREAK))
FIELD        (ARP_pln,                              1,       const_n8 (4, BREAK))
FIELD        (ARP_op,                               2,       range_n16 (1, 2))
FIELD        (ARP_sha,                              6,       anonymize_ethernet_addr)
FIELD        (ARP_spa,                              4,       anonymize_ip_addr)
FIELD        (ARP_tha,                              6,       anonymize_ethernet_addr)
FIELD        (ARP_tpa,                              4,       anonymize_ip_addr)

// ip.anon
FIELD        (IP_verhl,                             1,       KEEP)
FIELD        (IP_tos,                               1,       KEEP)
FIELD        (IP_len,                               2,       KEEP)
FIELD        (IP_id,                                2,       KEEP)
FIELD        (IP_frag,                              2,       KEEP)
FIELD        (IP_ttl,                               1,       KEEP)
FIELD        (IP_proto,                             1,       KEEP)
PUTOFF_FIELD (IP_cksum,                             2,       ZERO)
FIELD        (IP_srcaddr,                           4,       anonymize_ip_addr)
FIELD        (IP_dstaddr,                           4,       anonymize_ip_addr)
FIELD        (IP_options,                           VARLEN,  anonymize_ip_options)
PICKUP_FIELD (IP_cksum,                             0,       recompute_ip_checksum)
FIELD        (IP_data,                              VARLEN,  anonymize_ip_data)

// ip-frag.anon
FIELD        (IPFRAG_data,                          RESTLEN, SKIP)

// ip-option.anon
CASE         (IPOPT_eol,          IPOPT_EOL,        1,       KEEP)
CASE         (IPOPT_nop,          IPOPT_NOP,        1,       KEEP)
CASE         (IPOPT_rr,           IPOPT_RR,         VARLEN,  IPOPT_anonymize_record_route)
CASE         (IPOPT_ra,           IPOPT_RA,         4,       const_n32 (0x94040000UL, CORRECT))
DEFAULT_CASE (IPOPT_other,                          VARLEN,  IPOPT_alert_and_replace_with_NOP)

// ip-data.anon
CASE         (TCP,                IPPROTO_TCP,      VARLEN,  anonymize_tcp_pkt)
CASE         (UDP,                IPPROTO_UDP,      VARLEN,  anonymize_udp_pkt)
CASE         (ICMP,               IPPROTO_ICMP,     VARLEN,  anonymize_icmp_pkt)
DEFAULT_CASE (IP_other,                             RESTLEN, SKIP)

// icmp.anon
FIELD        (ICMP_type,                            1,       KEEP)
FIELD        (ICMP_code,                            1,       KEEP)
PUTOFF_FIELD (ICMP_chksum,                          2,       ZERO)
FIELD        (ICMP_data,                            RESTLEN, anonymize_icmp_data)
PICKUP_FIELD (ICMP_chksum,                          2,       recompute_icmp_checksum)

// icmp-data.anon
CASE         (ICMP_echoreply,     ICMP_ECHOREPLY,     VARLEN,  anonymize_icmp_echo)
CASE         (ICMP_unreach,       ICMP_UNREACH,       VARLEN,  anonymize_icmp_context)
CASE         (ICMP_sourcequench,  ICMP_SOURCEQUENCH,  VARLEN,  anonymize_icmp_context)
CASE         (ICMP_redirect,      ICMP_REDIRECT,      VARLEN,  anonymize_icmp_redirect)
CASE         (ICMP_echo,          ICMP_ECHO,          VARLEN,  anonymize_icmp_echo)
CASE         (ICMP_routersolicit, ICMP_ROUTERSOLICIT, VARLEN,  anonymize_icmp_routersolicit)
CASE         (ICMP_timxceed,      ICMP_TIMXCEED,      VARLEN,  anonymize_icmp_context)
CASE         (ICMP_paramprob,     ICMP_PARAMPROB,     VARLEN,  anonymize_icmp_paramprob)
CASE         (ICMP_tstamp,        ICMP_TSTAMP,        VARLEN,  anonymize_icmp_tstamp)
CASE         (ICMP_tstampreply,   ICMP_TSTAMPREPLY,   VARLEN,  anonymize_icmp_tstamp)
CASE         (ICMP_ireq,          ICMP_IREQ,          VARLEN,  anonymize_icmp_ireq)
CASE         (ICMP_ireqreply,     ICMP_IREQREPLY,     VARLEN,  anonymize_icmp_ireq)
CASE         (ICMP_maskreq,       ICMP_MASKREQ,       VARLEN,  anonymize_icmp_maskreq)
CASE         (ICMP_maskreply,     ICMP_MASKREPLY,     VARLEN,  anonymize_icmp_maskreq)
DEFAULT_CASE (ICMP_other,                           VARLEN,  ICMP_alert_and_skip)
```

```
// icmp-echo.anon
FIELD        (ICMP_echo_id,                         2,       KEEP)
FIELD        (ICMP_echo_seq,                        2,       KEEP)
FIELD        (ICMP_echo_pyld,                       RESTLEN, SKIP)

// icmp-context.anon
FIELD        (ICMP_context_unused,                  4,       ZERO)
FIELD        (ICMP_context,                         RESTLEN, anonymize_ip_pkt)

// icmp-redirect.anon
FIELD        (ICMP_redirect_gateway,                4,       anonymize_ip_addr)
FIELD        (ICMP_redirect_context,                RESTLEN, anonymize_ip_pkt)

// icmp-routersolicit.anon
FIELD        (ICMP_rs_reserved,                     4,       const_n32 (0, CORRECT))

// icmp-paramprob.anon
FIELD        (ICMP_pp_pointer,                      1,       KEEP)
FIELD        (ICMP_pp_unused,                       3,       ZERO)
FIELD        (ICMP_pp_context,                      RESTLEN, anonymize_ip_pkt)

// icmp-tstamp.anon
FIELD        (ICMP_ts_id,                           2,       KEEP)
FIELD        (ICMP_ts_seq,                          2,       KEEP)
FIELD        (ICMP_ts_orig_ts,                      4,       KEEP)
FIELD        (ICMP_ts_recv_ts,                      4,       KEEP)
FIELD        (ICMP_ts_trsm_ts,                      4,       KEEP)

// icmp-ireq.anon
FIELD        (ICMP_ireq_id,                         2,       KEEP)
FIELD        (ICMP_ireq_seq,                        2,       KEEP)

// icmp-maskreq.anon
FIELD        (ICMP_maskreq_id,                      2,       KEEP)
FIELD        (ICMP_maskreq_seq,                     2,       KEEP)
FIELD        (ICMP_maskreq_mask,                    4,       KEEP)

// udp.anon
FIELD        (UDP_srcport,                          2,       KEEP)
FIELD        (UDP_dstport,                          2,       KEEP)
FIELD        (UDP_len,                              2,       KEEP)
PUTOFF_FIELD (UDP_chksum,                           2,       ZERO)
FIELD        (UDP_data,                             RESTLEN, SKIP)
PICKUP_FIELD (UDP_chksum,                           2,       recompute_udp_checksum)

// tcp.anon
FIELD        (TCP_srcport,                          2,       KEEP)
FIELD        (TCP_dstport,                          2,       KEEP)
FIELD        (TCP_seq,                              4,       KEEP)
FIELD        (TCP_ack,                              4,       KEEP)
FIELD        (TCP_off,                              1,       KEEP)
FIELD        (TCP_flags,                            1,       KEEP)
FIELD        (TCP_window,                           2,       KEEP)
PUTOFF_FIELD (TCP_chksum,                           2,       ZERO)
FIELD        (TCP_urgptr,                           2,       KEEP)
FIELD        (TCP_options,                          VARLEN,  anonymize_tcp_options)
PICKUP_FIELD (TCP_chksum,                           0,       recompute_tcp_checksum)
FIELD        (TCP_data,                             RESTLEN, SKIP)

// tcp-option.anon
CASE         (TCPOPT_eol,         0,                1,       KEEP)
CASE         (TCPOPT_nop,         1,                1,       KEEP)
CASE         (TCPOPT_mss,         2,                4,       KEEP)
CASE         (TCPOPT_wsopt,       3,                3,       KEEP)
CASE         (TCPOPT_sackperm,    4,                2,       KEEP)
CASE         (TCPOPT_sack,        5,                VARLEN,  KEEP)
CASE         (TCPOPT_tsopt,       8,                10,      renumber_tcp_timestamp)
CASE         (TCPOPT_cc,          11,               VARLEN,  KEEP)
CASE         (TCPOPT_ccnew,       12,               VARLEN,  KEEP)
DEFAULT_CASE (TCPOPT_other,                         VARLEN,  TCPOPT_alert_and_replace_with_NOP)
```

**Figure 3: Full anonymization policy.**