Packets with Provenance

Anirudh Ramachandran, Kaushik Bhandankar, Mukarram Bin Tariq, and Nick Feamster
School of Computer Science, Georgia Institute of Technology
{avr,kaushikb,mtariq,feamster}@cc.gatech.edu

ABSTRACT

Traffic classification and distinction allows network operators to provision resources, enforce trust, control unwanted traffic, and traceback unwanted traffic to its source. Today's classification mechanisms rely primarily on IP addresses and port numbers; unfortunately, these fields are often too coarse and ephemeral, and moreover, they do not reflect traffic's provenance, associated trust, or relationship to other processes or hosts. This paper presents the design, analysis, user-space implementation, and evaluation of *Pedigree*, which consists of two components: a trusted tagger that resides on hosts and tags packets with information about their provenance (*i.e.*, identity and history of potential input from hosts and resources for the process that generated them), and an arbiter, which decides what to do with the traffic that carries certain tags. *Pedigree* allows operators to write traffic classification policies with expressive semantics that reflect properties of the actual process that generated the traffic. Beyond offering new function and flexibility in traffic classification, *Pedigree* represents a new and interesting point in the design space between filtering and capabilities, and it allows network operators to leverage host-based trust models to decide treatment of network traffic.

1. Introduction

Enterprise and transit networks must be able to classify and differentiate network traffic to enable provisioning and keep networks secure. Ideally, operators would be able to differentiate traffic according to expressive features, such as the application that generates the traffic; the host or user that generated the traffic, and the associated privileges of that user; whether or not that host might be infected, and so forth. Differentiating traffic on such features would allow operators to upgrade or downgrade the service seen by particular traffic flows based on flexible attributes and properties and would, as a result, facilitate much more expressive policies (*e.g.*, filtering traffic based on whether the process that generated the traffic had talked to a known infected host or not). Operators might also want to control traffic based on the properties of the process that generated it (*i.e.*, the application that generated it, or what other hosts or files may have affected the process).

Today, traffic classification is coarse and imprecise. Network operators typically classify traffic using port numbers or IP addresses. This approach is often too coarse or ephemeral. It is also indirect: IP addresses carry no information about the *provenance* of the traffic, such as the process (or group of processes, or host) that generated the traffic. Providing this type of function is difficult: The large volume of traffic that traverses the network makes inspecting each packet's contents infeasible, and the packets have no markings that bind them to a particular process or group of processes on any particular host. We believe that significant gains in network traffic monitoring require means to bind that traffic back to a process group (and corresponding level of trust) on a host.

This paper presents *Pedigree*, which allows network devices to classify traffic based on the privileges and provenance of that traffic, rather than a coarse identifier like an IP address that carries no semantics. *Pedigree* allows network operators to express policies based on (1) what *container*—a persistent identifier for a resource (*e.g.*, process group, virtual machine)—generated the traffic; (2) what inputs the process that generated the traffic has received ("taint set"). *Pedigree* has two components, as shown in Figure 1. The first component is the *tagger*, a trusted module that resides on the host and tags traffic with the identification of the container ("container ID") and taint set of the process that initiated the traffic. Users in an enterprise or customers of an ISP who want to receive better service (*e.g.*, provisioning, stronger security guarantees) may install such a tagger. The second module, the *arbiter*, resides on a network element and acts on the traffic according to these tags and the network operator's policy. Such actions might include filtering or shaping the traffic, shunting the traffic to a deep packet inspection device, or re-routing the traffic to a better provisioned network with stronger performance guarantees. Network elements can either upgrade or downgrade traffic based on these tags, which essentially blurs the distinction between the two extreme design points of capabilities (*i.e.*, keeping the network "off by default" and permitting only certain traffic) and filtering (*i.e.*, keeping the network "on by default" and discarding undesirable traffic).

The tags that each packet carries map to a host container from which the traffic originated, and reflect the specific properties of that container, such as whether it has access to certain keys, whether it has been affected by other processes or files (even across hosts), etc. Network elements can then associate the tags that the traffic carries with a particular container that generated the traffic (or to the commu-

Georgia Tech CSS Technical Report GT-CS-08-02

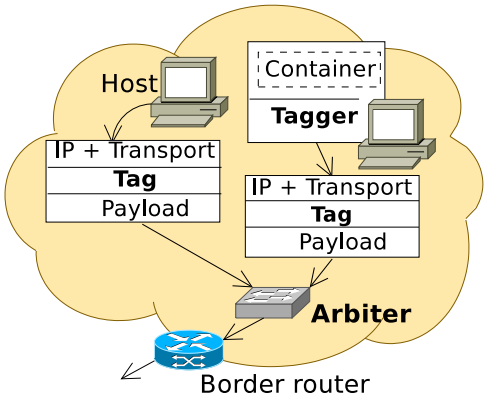

Figure 1: High-level design of *Pedigree*.

nication history of the process that generated the traffic) and take specific actions based on those tags, rather than acting a on coarse-grained, ephemeral handle such as an IP address or port. Unlike IP addresses, these container IDs and taints are persistent: Even if a host changes its location or IP address, or if it reboots, the markers on traffic that originate from a particular host can always be tracked back to the process group that generated them.

Pedigree marks packets with *tags* that have two distinct components: *container ID*, which is deterministic (and, in some cases, cryptographic) and essentially acts as an attestation regarding what container generated the traffic on a particular host; and *taints*, which reflect the set of tags that a container accumulates by taking inputs from other files, processes, and network sockets. Each packet carries such a tag all the way to its destination; in principle, any network element along the path could take action (*e.g.*, filter, provision) based on this tag. A network element that wants to make a provisioning, forwarding, or filtering decision (an *arbiter*) can then make forwarding decisions using policy based on the container IDs, the taints, or a conjunction of the two (as, reflected in the examples above). Although the arbiter could certainly also reside in a trusted domain on the host, classifying tagged traffic with an arbiter that is separate from the host itself has two main advantages: First, *Pedigree*'s tags allow an arbiter to classify traffic based on relationships and correlations that may exist *across hosts*. Second, updating traffic policies may be more manageable than doing so at individual hosts.

Implementing *Pedigree* entails several challenges. First, the tags must be resilient to forgery, minting, and replay: A host or element that observes the tags on one packet should have no mechanism or incentive to "steal" those tags to gain a higher privilege. Similarly, a host should not be able to mint a new set of tags to evade blacklisting or attain higher privileges. Second, tagging must be fast: A host must be able to tag packets as quickly as an application can send them, to avoid degrading application service. Similarly, arbitration must also be fast: network elements must be able to perform arbitration at line rate.

This paper presents the following contributions. First, we present a new framework for classifying traffic that leverages a trusted component on the host to help operators attribute semantics and trust to network traffic. This framework has two conceptual contributions: extending host-based security models into the network and highlighting a new point in the design space between filtering ("on by default") and capabilities ("off by default"). Second, we design and implement *Pedigree*, a system that operates within this framework to help both enterprise networks and ISPs classify traffic based on provenance and trust levels. Third, we present a user-space prototype implementation of *Pedigree* and show that packet and storage overhead is negligible for all but short connections.

The rest of this paper is organized as follows. Section 2 discusses motivating applications. Section 3 describes the trust model that *Pedigree* assumes. Section 4 presents the main idea of *Pedigree* and presents the corresponding threat model. Section 5 describes the detailed design of the tags and taints, as well as how these taints propagate across the processes within a host, as well as, across the hosts as messages traverse the network. Section 7 describes the prototype implementation of *Pedigree*. In Section 6, we analyze the security of *Pedigree* under various threats; in Section 8, we discuss the performance overhead of *Pedigree*. Section 9 discusses various optimizations and extensions, Section 10 discusses related work, and Section 11 concludes.

2. Motivation

Whereas today's traffic classification and filtering methods attempt to classify traffic based on second-order effects, the ideal scenario would allow operators to track first-order effects, such as filtering traffic that comes from processes that are known to be infected, or communicating with other infected hosts. Although much previous work has been done to track interactions and assign trust levels to processes within hosts, all of this information is lost when network traffic leaves the host and enters the network. Network devices currently have no way to exercise complex policy because traffic is completely devoid of any information that can be tied to provenance or semantics. The main goal of *Pedigree* is to extend trust levels and semantics that can be attributed host-level processes into the network. Today, filtering and classification devices rely on port numbers and IP addresses to attribute semantics to traffic. These approaches are inadequate; network devices could implement more expressive policies if traffic carried additional information about the host process that originated the traffic.

2.1 Today's Classification: Indirect and Brittle

Relying chiefly on heuristics based on IP address and port numbers to classify network traffic is both *indirect* and *brittle*: Classifying traffic based on an IP address or port *indirectly* expresses an operator's intent to perform some higher level action (*e.g.*, rate-limiting peer-to-peer traffic, filtering spam). In addition, these classification rules are brittle as

they require change when traffic sources change. In this section, we elaborate on these shortcomings.

Why not IP addresses? Many filters, blacklists, and access control lists are based on IP addresses, such as DNS-based blacklists (DNSBLs) for spam prevention. Unfortunately, the IP addresses that send spam continually change (*i.e.*, incidentally due to dynamic addressing and intentionally, due to techniques such as route hijacking [24]), making it difficult to keep these lists up to date. Although statistical and behavioral blacklisting methods (*e.g.*, [14, 23]) can help counteract the dynamism of IP addresses by tracking behavioral invariants, these methods rely on indirect inference. A better method would allow operators to track the privilege and provenance of traffic based on deterministic properties that are contained in the packets themselves.

Why not port numbers? Operators also use port numbers to attribute traffic to applications. For example, operators rate-limit peer-to-peer traffic, or assign higher priority to real-time traffic, using destination port numbers that correspond to well-known services. For example, because many botnets use Internet Relay Chat (IRC) for their rallying behavior, inspecting IRC traffic used to be a reliable method of detecting such traffic. Today, however, many applications use non-standard ports, which makes it difficult to identify such applications without expensive deep packet inspection (DPI). As malicious applications are increasingly attempting evasion of port-based filtering by rallying using common ports (*e.g.*, HTTP [2] or peer-to-peer protocols [26]), even applying DPI becomes impractical.

2.2 Pedigree: Robust and Direct

Pedigree's tags (described in detail in Section 4) provide the following semantics to traffic: (1) the trust associated with the process (or group of processes) that generated the traffic ("What is the entity that generated the traffic?"); (2) the provenance of the process or group of processes that generated the traffic ("Who has this entity been 'talking to'?"). These semantics enable several new functions and applications because they *directly* reflect the semantics an operator might want to express. Because these tags are persistent (Section 4, these semantics are also *robust*. The rest of this section describes functions that these tags facilitate. *Pedigree* allows all of these applications to be implemented within a single, coherent framework.

Provisioning Operators may sometimes prioritize for certain types of applications (*e.g.*, prioritize VoIP, de-prioritize peer-to-peer traffic, etc.) or between certain customers (*e.g.*, enterprise sites, specific users). To prioritize traffic for certain applications today, operators must write policies that classify traffic based on brittle identifiers such as IP addresses or port numbers, which suffer from a number of drawbacks. Instead, operators might want to classify traffic using a tag that directly links network traffic to the entity that generated it and provides information about whether or not that principal belongs to a certain group. Certain tags could, for example, provide access to better provisioned, less loaded, or lower latency paths over "standard" paths (in this context, upgrades are analogous to capabilities [36]).

Blacklisting Unwanted traffic such as spam, denial of service attacks, and illegal content distribution are taxing network and system resources. Eradicating such traffic is challenging because of their similarity to legitimate traffic, and operators must often resort to some form of deep packet inspection, which may be impractical on high-speed links; even with fast DPI technology, encrypted traffic can still evade being downgraded [33, 31].

Instead, tags that identified the application that originated the traffic and tracked the history of the application that generated the traffic (*e.g.*, whether that application had communicated with other known malicious processes) could be used to rate-limit or filter traffic from these applications.

Exfiltration Enterprise network administrators must protect internal hosts from compromise and prevent private data or resources from leaving certain designated areas of the network ("data leakage", or exfiltration). Current solutions for exfiltration involve either routing all traffic that leaves the network through a middlebox that performs expensive (and often ineffective) DPI-based watermarking techniques, or filtering certain ports, which both benign and malicious processes can evade using encryption or tunneling. Using tags that could associate network data transfers from a host to files that are being read on the host, operators can reliably and cheaply ensure that only data pertaining to non-critical files leave the enterprise network.

Secure network regions Applications such as online banking authenticate customers at the application layer (or above) but have no way to ensure that traffic is not being routed to (or through) malicious entities; such subversion may be performed even by a keylogger on the host that has no privileges. A network operator may wish to enforce that traffic destined to such "secure regions" originate only from trusted applications. In order to implement this policy, the operator can require that destined to certain parts of the network from a user's computer *necessarily* contain tags that indicate that the application that generated the traffic possesses a valid credential (such as a private-key signature); an egress router could authenticate the credential before forwarding the traffic.

3. Trust Model and Assumptions

Because arbiters in the middle of the network make decisions about how to forward traffic based on *Pedigree*'s tags, malicious processes or network entities have an incentive to try to manipulate these tags. Accordingly, we make some assumptions about the capabilities of various entities in the network to observe and manipulate the tags. This section discusses two important assumptions: First, that network elements are trusted and, as a result, do not manipulate the tags. Second, that the host based *tagger* (*i.e.*, the entity that annotates the packet with tags that conveys information

about provenance and privilege) is trusted. The rest of this section discusses these two assumptions in more detail.

Assumption 1: *Network elements do not modify tags.*

Pedigree requires that network elements on the path (*i.e.*, routers, switches, intrusion detection systems, proxies, or firewalls) will: (1) forward packets from sender to recipient without modifying *Pedigree*'s tags; (2) not use information gleaned from tags for malicious purposes. Because network elements are usually operated by ISPs or enterprises, and because users already implicitly trust network elements to forward packets to modifying higher-layer packet headers or payloads, we believe that this assumption is reasonable.

Assumption 2: *The end-host has a trusted component that is always at a higher privilege level than any untrusted resource.*

Pedigree requires that the component on the host that maintains provenance information for all resources on the host, attaches tags to outgoing packets, and incorporates tag information from incoming packets into the reading process's provenance, is *trusted*. In order to deploy the tagger at the correct trust level, we divide all entities (processes, kernel, Virtual Machine Monitor (VMM), hardware, etc.) on general purpose operating systems into the five distinct trust levels listed below. Also shown is the tagger deployment option assuming a given trust level is the highest for all untrusted components in the system.

1. *User-level unprivileged processes*: These processes cannot modify resources owned by other users (unless explicitly granted permission). *Tagger Deployment:* As a privileged process or daemon.
2. *User-level privileged process*: These processes have super-user rights and may modify any resource at the user-level. They, however, must issue system calls to the kernel to gain access to physical resources. *Tagger Deployment:* As a protected kernel module accessible only to the system administrator (*e.g.*, using passphrase protection) [1].
3. *Kernel-level process, either as a module or as built-in code*: These processes have unrestricted access to all virtual (*e.g.*, vfs) and physical resources. *Tagger Deployment:* If the kernel is vulnerable, the administrator must execute the operating system in a virtual machine, and deploy the tagger secure code within the hypervisor [11].
4. *VMM level*: If the OS is executed within a virtual machine, the virtual machine monitor (or *hypervisor*) runs at a higher privilege level than the guest operating system (*e.g.*, Xen [1]). *Tagger Deployment:* If the VMM is susceptible to compromise, the tagger must be run within a trusted hardware module.
5. *Trusted computing platform*: If the platform is equipped with a hardware chip that executes trusted code, the Trusted Platform Module (TPM) becomes the highest level of trust for the platform. *Tagger Deployment:* If the hardware on a host cannot be trusted, the tagger must be deployed outside the host (*e.g.*, a home router or cable modem).

The administrator of a end-host can decide the highest threat level to assign for untrusted processes based on anticipated threats: a general-purpose desktop computer may execute arbitrary code at user-level, so deploying the tagger at kernel level (level 3) is viable. Section 6.1 deals with this issue in greater detail.

[1] In current operating systems, the distinction between trust level 2 (user-level privileged process) and trust level 3 (kernel-level process or module) is blurry: any process that has super-user privileges on an operating system can install arbitrary code (as a module) to the kernel, or even change the kernel itself. *Pedigree* requires that processes at one trust level cannot affect processes at higher trust levels; enforcing this separation between levels 2 and 3 requires a modification to the kernel to ensure that only certain privileged processes *explicitly authorized by the user* (*e.g.*, using a passphrase that is known only to the user and the running kernel) can change kernel parameters or load or unload modules. With this modification, executing the tagger as a kernel-level process will be sufficient most general purpose computers. Surveys show that for most operating systems are exploited due to vulnerabilities present in user-level process or services (privileged or unprivileged); kernels are largely free of exploitable vulnerabilities [32].

4. Pedigree: System Design and Function

This section presents the design of *Pedigree*. Section 4.1 discusses the portion of *Pedigree* that resides on end hosts, and tracks provenance and interactions between resources within a host: the *tagger*. Section 4.2 discusses the the *arbiter*, which resides in the network and uses the information contained tags to classify traffic.

4.1 On The Host: Tags

Pedigree tracks interactions between resources (*i.e.*, files, processes, and sockets) in an operating system and attributes persistent tags to each resource. *Pedigree* annotates outgoing traffic with *tags*. When a process sends data on the network, *Pedigree*'s tagger annotates outgoing packets with a tag that represent the provenance of a packet: essentially, the process that generated the traffic and where it has taken input from. When a process reads data from the network, the tagger updates the reading process's tags with tags on incoming packets. We describe the semantics and structure of these tags and how they are used to track interactions.

4.1.1 *Tag semantics and structure*

Semantics. Tags convey two types of information: (1) *local properties* about the process that generates the traffic (*e.g.*, information about the application generating the traffic, whether or not the user of some process possesses a key); and (2) the *history of interactions* of that container with other local and remote containers. The tags that are assigned to a given application's processes (as well as its derivatives, such as child processes) are unique and survive reboots, but the tags for the same kind of application on different machines are distinguishable from each other. The tags also serve as audit trails about interactions between resource ("who-talks-to-whom"), thus allowing a network device to filter traffic based on the who or what the process that generated the traffic has interacted with.

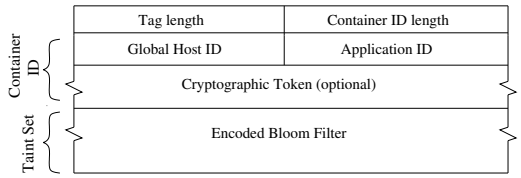

Figure 2: Structure of a tag in *Pedigree*.

Structure. Figure 2 illustrates the structure of *Pedigree*'s tags. Tags have two parts: (1) a *container ID*, which allows network elements to unforgeably identify the sender's unique host ID and application details (*e.g.*, whether the sender is using an approved Web browser program); (2) a *taint set*, which provides information about a container's interaction with other applications and hosts.

The **container ID** identifies a virtual "container" within which all of the resources of an application are constrained, analogous to the terminology used in virtualization technologies like OpenVZ [21]. The container ID has two parts: (1) a *global host identifier* (GHID), which is a unique integer that is either tied to the installed operating system or to a unique hardware serial number on the device[2], and (2) an *application ID* (AppID), is a unique, persistent, integer associated with an application program, such as the binary file used to start the application, its configuration files, etc. All of the application's resources—the application binary files, files created by the application, the application's processes and process group, etc.—have the same persistent application ID.

The **taint set** for a resource stores the identifiers of all the other resources (local or remote) that it has interacted with in the past. In a fresh install of a typical operating system, *Pedigree* initializes the taint set of *all* the resources with the GHID and the respective AppID.

Pedigree maintains the taint set as a compact set data structure; our implementation uses a Bloom filter [3]. The size of the Bloom filter depends on the expected number of unique members and acceptable false positive rate for lookups. *Pedigree* compresses the taint sets while transmitting them as part of tags in the packets. *Pedigree* further reduces the overhead by including full taint set only at the start of the connection and using a hash on the subsequent packets.

Some processes may use the cryptographic **token** to indicate authorization to receive, say, better quality of service or access to a trusted network region. Depending on the key distribution mechanism and the nature of authorization required, this token might be an HMAC over fields in the packet and tag, a signature, or even a network capability [36].

4.1.2 *Tracking interactions with tags*

The tagger maintains each resource's current tag, either within the structure that the operating system uses to maintain resource metadata (*e.g.*, in the Linux kernel, the tag can be added as an extra field in the inode for a file, or as a field in `task_struct` for a process), or at a central location that contains tag informations of all resources (*e.g.*, a database).

The tagger updates the tags as processes are executed and interact with resources. Figure 3 shows the three ways that tags are propagated

1. *Resource A reads from Resource B or Resource B writes to Resource A* In this case, because both A and B already exist (*e.g.*, a process reads from a file or a network socket), *Pedigree* updates the *taint set* of A to include all of B's taints. The container IDs of both processes are unchanged.
2. *Resource A creates Resource B.* Because only processes can create resources, A will be a process. B may be another process (created using `fork(2)` in POSIX), a file, or a network socket. In this case, all fields of A's tags are copied as B's newly created tags.
3. *Resource A executes Resource B.* In this case, A replaces its own process image by executing the binary file B, using a system call such as `execve(2)` in POSIX. *Pedigree* sets the new process's AppID to the hash of the file B. The new process's taint set is set to the taint set of the file B. If A passes data to the new process through `execve`, *Pedigree* also updates the taint set of the new process with A's taint set.

Case 1 includes taints acquired by reading data from the network, as well as from files and other processes. If incoming network packets are carrying tags, the tagger updates the reading process's tags with tags from network packets; the contents of these tags will be propagated to other resources that the reading process creates or writes.

Case 2 includes new files or sockets created by processes (using `open(2)`, or new processes (using `fork(2)`). Child processes inherit the parent process's full tag when they are created, but changes to the taint sets of either the parent or child after creation are not propagated unless there is an explicit read between the two (using IPC or shared memory).

In Case 3, the tagger replaces the AppID with a hash to retain consistent AppIDs for the same application across reboots or multiple instances of the same application: irrespective of the process executing the application, the AppID of a process created using a specific binary file will remain the same. Enterprises can use the AppID to ensure whether a program generating traffic is authorized or not.

These three rules allow *Pedigree* to propagate tags for any type of interaction between resources, *both within and across hosts*. *Pedigree*'s taint propagation mechanism is transitive: if resource A sends data to resource B, who in turn sends data to resource C, resource C's taint set will include resource A's set as well. As we show in the next section, this important property allows *Pedigree* to track the history of interactions between files, processes, and hosts, which is useful in a variety of classification scenarios. Taint sets are

[2]The tagger on a machine can choose the set of serial numbers it wants to use to construct the GHID, as long as it can construct it again. The function may even be different per host.

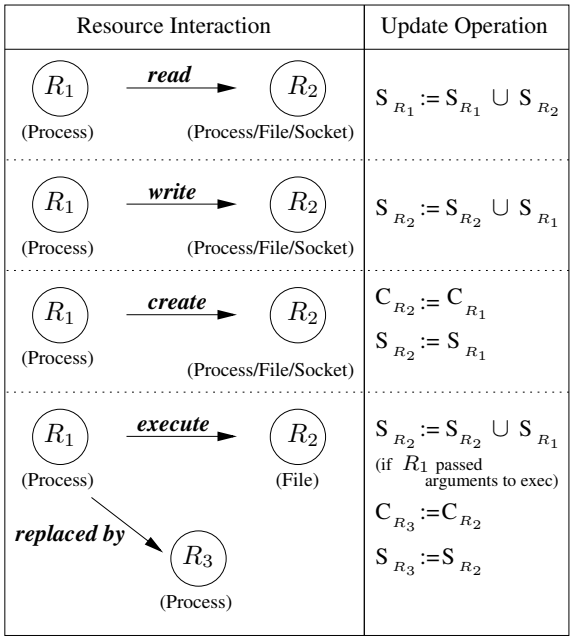

Figure 3: The system calls that cause *Pedigree* to update tags of involved resources. C_R refers to the Container ID of resource R, and S_R refers to the taint set of R. [4]

also susceptible to overflow; we discuss this issue in Section 5.3.

Although we have designed and implemented *Pedigree* around a POSIX-based Linux system, all general-purpose operating systems offer equivalent interfaces to applications.

4.2 In The Network: Arbitration

Once packets have been tagged, any network element along the path to a destination can perform arbitration to decide how each packet should be treated. We use the term *arbiter* to refer to a network element (*e.g.*, proxy servers, routers, authentication servers, intrusion detection systems, and load balancing systems) that inspects some part of the packet's tag (*i.e.*, the container ID, the taint set, or both) and takes some action based on its value (*e.g.*, filtering, rerouting, etc.).

Pedigree's tags allow network elements to either upgrade or downgrade the level of service that some traffic sees based on the tags carried in the packets. Operators can provide improved levels of service to applications that present appropriate authorization (*i.e.*, as a cryptographic token in the container ID). On the other hand, an arbiter can drop, rate-limit, or de-prioritize the low-priority or unwanted traffic. The rest of this section describes those actions in more detail.

Controlling access to resources. An arbiter could inspect the fixed fields on packet tags such as the AppID or signature to perform authentication for access to a restricted resource or secure region of the network (as described in Section 2). This scenario is most applicable in an enterprise network, where key management and distribution can be controlled and users can be asked to run common software (*e.g.*, the tagger). An arbiter could use the AppID to ensure that a user's traffic was generated using authorized software. It can also use the token to control access to more critical services, (*e.g.*, online banking portals).

Curtailing unwanted traffic. An arbiter that merely prevents malicious application traffic from passing through will only need to check whether a certain packet's taint set contains one or more taints belonging to known malware. For example, Intrusion Detection Systems (IDSes) or firewalls may inspect the taint set to check for membership of taints of known malicious applications. To blacklist traffic, an IDS might maintain a blacklist of taints known to belong to malicious files or servers and the taint sets of incoming packets to determine whether it contains any blacklisted taint. *Pedigree*'s taint set is a Bloom filter, so the arbiter only needs to perform a simple bitwise AND between the Bloom filters corresponding to a packet and a Bloom filter containing the sought malware taint. The query returns true (*i.e.*, the packet on the wire contains the taint of the sought malware) if the resulting Bloom filter is equivalent to the Bloom filter containing the malware. Upon discovering the presence of a taint, the IDS can take further action (*e.g.*, filter, drop, rate-limit, inspect payload).

5. Practical Considerations

Deploying *Pedigree* on a real network faces several practical concerns such as increased packet size, overhead in storing tags on each host, potential for misclassification due to overflow of the taint set of a resource's tag, etc.; in addition, *Pedigree* also needs to be hardened against a variety of attacks. This section addresses the practical considerations in deploying *Pedigree*, while Section 6 discusses attack defenses.

5.1 Packet Overhead

The taint set portion of a process's tag is a significant overhead if sent uncompressed with every packet in a connection. To reduce packet overhead, *Pedigree* performs two optimizations: (1) *Pedigree* sends the full tag only at the beginning of a connection, with subsequent packets carrying only a hash computed on the tag; and (2) because the Bloom filter (*i.e.*, taint set) that makes up the bulk of the tag is typically sparse, *Pedigree* compresses it using a fast algorithm.

Avoiding full tags on every packet. To reduce the per-packet overhead due to *Pedigree*'s tags for a network connection, the tagger sends the full tag only once, during the connection setup phase (*e.g.*, the three-way handshake of TCP-based protocols). Subsequent packets only contain a well-known hash function computed on the tag. Arbiters may compute the hash independently for verification, and even use this hashed string as the key to quickly look up the classification decision for later packets in the same connection. If the taint set of the process generating packets

[4] *Pedigree* does not treat network sockets separately as a socket normally maps one-to-one to its controlling process.

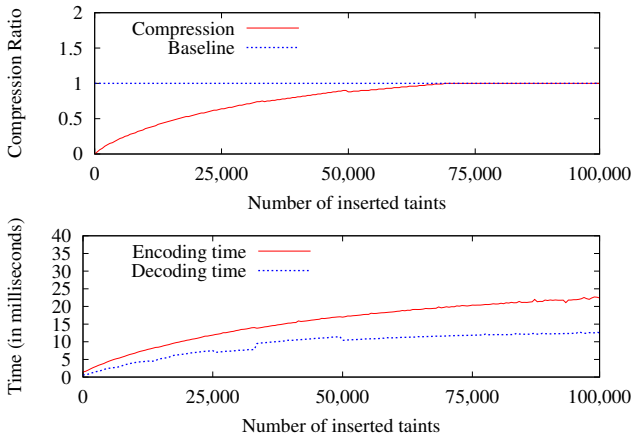

Figure 4: Golomb-Rice compression gain and latency evaluated using a Bloom filter (of maximum capacity 100,000 taints) as more taints are inserted.

changes after the beginning of the connection (*e.g.*, because the process acquired more taints by reading from another resource), only the newly set bits (and the hash of the new tag) are sent on subsequent packets.

Compressing Bloom filters. The uncompressed size of the Bloom filter depends on the expected number of taints a resource can acquire during its lifetime, and the maximum false positive rate for the chosen maximum number of taints. Using standard Bloom filter calculations [3], a Bloom filter that supports 100,000 insertions with a false positive rate of 0.01 or less requires approximately 127 KB.

To reduce the in-transit size of the filter, *Pedigree* compresses it before attaching it to network packets. The compression scheme used must be fast enough to not hamper packet processing, but effective enough to provide significant compression gain. Because the run-lengths of 0 bits in a Bloom filter follow a geometric distribution, *Pedigree* uses *Golomb-Rice* codes that are optimal for such distributions [25]. Golomb-Rice encoding is parametric: it does not require the computationally expensive sliding window look-ahead used by algorithms such as Deflate [22].

Figure 4 plots the variation in compression ratios, as well as the latency in encoding and decoding measured using our implementation of the Golomb-Rice algorithm, as more taints are inserted into an initially-empty Bloom filter. For sparse Bloom filters, Golomb-Rice encoding achieves high compression ratios, but understandably degrades as the Bloom filter reaches its capacity (because the filter approaches a uniform random binary string). The encoding and decoding latencies—especially if performed only once per a new connection—are also reasonable.

With the two optimizations above, the packet overhead due to *Pedigree*'s tags is acceptable. For a typical sparse Bloom filter, compression ratios of one-tenth are possible, enabling *Pedigree* to represent a 128 KiB taint set in less than 10 KiB. The container ID is also small, amounting to 32 bytes including a signature in our implementation. The full tag (approximately 10 KiB) needs to be sent only once per connection, and subsequent packets only carry a 20-byte hash of the tag, provided the tags of the sending process do not change mid-connection. Thus, for a 10 MB file transfer using 1500 byte packets, the overhead due to *Pedigree*'s tags amounts to only 1.5%.

5.2 Storage Overhead

Tags should use as little storage as possible. *Pedigree* stores only the tags of active resources (open files and running processes) in physical memory; the in-memory storage is of fixed-size and these entries are replaced (by writing through to disk) using a least-recently used (LRU) policy.

The tags of all permanent resources (*i.e.*, files), as well as the tags of transient resource which cannot be accommodated in physical memory, are stored on disk. To reduce the overhead due to the permanent store, *Pedigree* could maintain only one copy of the taint set (which accounts for the bulk of the tag) between all resources that have the same taint set; this optimization would especially be useful in representing the taint sets of all resources that have the initial, default taint. Yet another optimization involves maintaining only "diff" of the taint set for a resource if its taint set is similar to that of another resource (*e.g.*, two copies of the same file which have been written to by two different processes). Even without optimization, the number of permanent tags that *Pedigree* will accumulate on a host is linearly bound to the number of regular files on the disk.

5.3 Taint Set Overflow

Host processes continuously accumulate taints as they communicate with other processes (local or across the network) and read files, but their taint sets can accommodate only a fixed number of taints before the false positive rates increase beyond acceptable levels; we call this occurrence "taint set overflow". If false positive likelihood of a taint set is high, arbiters will not act upon the presence of a malicious taint in the taint set, due to the risk of misclassifying traffic from processes that may have *legitimately* overflowed their taint sets. Malicious processes may attempt to capitalize on the arbiter's inability, and intentionally create an overflow of their taint sets in order to "hide" their malicious taints among many others (*e.g.*, by contacting arbitrary hosts on the Internet, and acquiring at least one new taint per host).

To prevent a resource's taint set from overflowing, *Pedigree* clears portions of full (or nearly full) taint sets. Clearing bits, however, induces false *negatives* in Bloom filter lookups. In this section, we first argue that most resources on a typical end-user operating system are unlikely to ever overflow their taint sets We then present two schemes to unset portions of a Bloom filter as fills up, as well as a mechanism for clustering hosts that exchange resources based on common taints.

5.3.1 *Likelihood of Taint Set Overflow*

Most *legitimate* applications on end-hosts accumulate only a fixed number of taints in their taint set. Even for applications such as Web browsers and P2P software, the taints

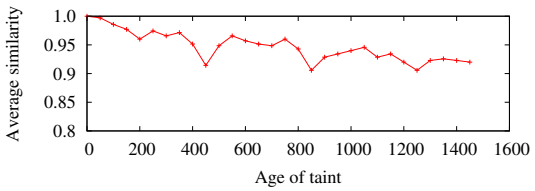

Figure 5: Evaluation of bit resetting Scheme 1: The taints that are inserted last to the Bloom filter have a higher average similarity value than earlier taints.

they accumulate in one session (*i.e.*, the time that a particular process executing the application binary is alive) are usually not carried over to the next session, as these applications never read files that they wrote to disk in a previous session. For example, if a Web browser process P on a host downloads the file F from a remote host, the taint set of P, S_P, will be updated as $S_P := S_P \cup S_F$, which will also be the taint set of the copy of file F written to the host's disk. The taint set of the browser's binary file itself is not affected, and the next process executing the binary file will not acquire file F's taints. Contemporary browsers such as Firefox fork child processes for each download which die after the download completes; such forking further segregates the taints of the downloaded files from the parent browser process.

Admittedly, the number of taints that a legitimate process can acquire in a session depends on usage patterns, and representative figures for taints acquired by end-user applications cannot be obtained without an extensive user study. Even so, the number of taints required to fill up a Bloom filter—approximately 100,000 in our implementation—requires the host process to contact at least the same order of different remote processes *within one session*, which we believe is unlikely.

5.3.2 *Probabilistic Bit Resetting*

In order to prevent taint set overflows, we instrument the tagger to reset one or more bits from a resource's Bloom filter if the filter is filled beyond a threshold. Unfortunately, if bits in a Bloom filter are reset, the filter loses the property of never returning a false negative: even the keys (*i.e.*, taints) that were inserted into the filter may not have *all* of their corresponding bits set. To address this issue, the arbiter must modify its technique for querying a taint set for membership of a particular taint: instead of verifying that *all* bits corresponding to the k hashes for a certain key are set, the arbiter must use a *similarity* metric, which may be the fraction of bits (out of k) that are set for any given key (*i.e.*, taint). Depending on the type of taints that are preserved, we present two strategies for resetting bits.

Scheme 1: Preserve later taints. The first scheme involves resetting one or more bits *with equal probability* each time a new taint is inserted. This scheme preserves recently inserted taints. To verify that taints inserted later are likely to have higher similarity scores than earlier taints, we intentionally cause a Bloom filter to overflow, and measure the average similarity of inserted taints. Figure 5 shows the average similarity of buckets of taints, sorted by age (*i.e.*, a taint with lower age is inserted later), for 1,500 taints inserted into a Bloom filter. We instrumented the Bloom filter to allow bit resetting to kick in after very few taints are inserted. In Figure 5, the last inserted 100 taints have average similarity of 1, indicating that these taints are hardly affected by resetting.

This scheme is suitable for legitimate application processes that acquire taints over the course of their normal operation, such as Web browsers: the taints of newly read (*i.e.*, downloaded) files—which could potentially be malicious—remain unmodified until they are written to disk as separate files, at the expense of taints of files read earlier.

Scheme 2: Preserve earlier taints. The second scheme requires storing a snapshot of the Bloom filter for each resource *at the time the resource was first created on the host*. When bits need to be reset, the current Bloom filter is compared with the snapshot, and common bits are given highest priority against resetting. The scheme thus protects older bits at the expense of newly set bits, and is suitable for potentially untrusted application binaries: in the case of malware, the taints in the snapshot likely include the malicious taints, and the bits corresponding to these taints will be preserved *even if the malicious program later attempts to acquire a large number of taints*.

5.4 Automatically Identifying Taints

Using *Pedigree*, a network administrator who knows the taints associated with the malware can filter traffic generated by malicious programs, where these taints are presumably acquired through another source (*e.g.*, by directly contacting the malware hosting server, from a honeypot, etc.). This process, however, requires the administrator to acquire the taints of these malware in order to perform lookups on real-time traffic. These requirements are hard to meet: miscreants release many new variants of malware each day [5], and there will be significant delays before security researchers can obtain and classify these taints. In fact, it is precisely because of this large influx of new malware variants (many of them auto-generated, such as polymorphic worms) that manual classification is hard. *Pedigree*'s resource tracking mechanism ensures that even these polymorphic variants of worms inherit the taints of the original malware binary; thus, although the malware processes on each host will have taints specific to that host, they will all posses a subset of *common* taints that were embedded in the first copy of the malware executable (*e.g.*, from the malware hosting Web site). [5]

Arbiters in the core of the network are in a position to observe traffic generated by a malicious program from different hosts, and can apply clustering on the taint sets of traffic from these hosts in order to discover: (1) groups of host processes that possess common subsets of taints; and (2) the bits in the Bloom filter corresponding to the common subset of taints for a group. In this section, we present

[5] This statement assumes that all copies of a malware have the same root, but this assumption is not necessary: clustering will merely group the subsets of malware processes with different root taints in separate clusters, which is equally beneficial to the operator.

Varying number of clusters (C) Fixed params: $B = 50$; $b = 2000$; $a = 1000$; $N = C \times B$			
Clusters, C	*Clusters Found*	*Avg. FNs*	*Avg. FPs*
5	5	0	1.6
10	10	0	3.6
15	24	0	1.45
20	20	0	1.6
25	28	0	1.6
Varying number of noise taint sets (N) Fixed params: $C = 5$; $B = 50$; $b = 2000$; $a = 1000$			
Noise, N	*Clusters Found*	*Avg. FNs*	*Avg. FPs*
50	8	0	0
100	8	0	1.4
150	7	0	0
200	5	0	0
250	5	0	1.6
...	...	...	...
600	7	0	2
Varying noise within clusters (a) Fixed params: $C = 5$; $B = 50$; $N = 250$ $b = 2000$			
Cluster Noise, a	*Clusters Found*	*Avg. FNs*	*Avg. FPs*
500	7	0	2.29
1000	8	0	1.75
1500	6	0	0.5
2000	6	0	0.83
2500	5	**6.6**	1.2

Notation

b – Initial taints per cluster
B – Taint sets per cluster
C – Number of clusters
a – Random taints per taint set
N – Number of "ambient noise" taint sets

Table 1: Clusters identified using different simulation parameters. Avg FNs indicate the average False Negatives per cluster, and Avg. FPs indicate average False Positives per cluster. All experiments were performed using a Bloom filter that supports 10,000 insertions.

a clustering-based algorithm that arbiters can deploy to *automatically* identify taint sets that have subsets of taints in common, which will likely be the case for a particular malware variant that runs on many infected hosts on the Internet.

Our algorithm uses the idea that, although some clusters are likely to be non-malicious (*e.g.*, different host processes using a particular software downloaded from the same Web site will cluster on the subset of taints corresponding to the original copy of the software), the operator needs to manually verify only one host process's traffic per cluster; once one process's traffic is identified as malicious, the operator can automatically filter other traffic that clustered with the identified traffic. In addition, the operator can also instrument the clustering algorithm to return the smallest set of '1' bits that define a cluster of malicious traffic; he can then use this set as a direct filter on the taint sets of future traffic.

Evaluation. In this section, we evaluate how well the clustering scheme described above identifies groups of related processes (even from different hosts) that have a common subset of tags. In our simulation, we create C clusters of B taint sets (*i.e.*, Bloom filters) each. All the taint sets within any given cluster are initialized with the same b taints (but with different taints across clusters). The initial taint set of a cluster is analogous to the "root" taint of a malicious executable before it spreads in the wild.

In order to simulate copies of a malware spreading across different hosts and acquiring new taints, we now add a random taints to each of the B taint sets of each cluster. All taint sets are large enough to accommodate $a+b$ taints. Each taint within a cluster represents a different host process running the malware: they all possess the original b root taints, but also posses a taints acquired due to later interactions between hosts and processes.

We also create N noise taint sets that each have $a + b$ random taints in order to simulate traffic with unrelated taint sets that an arbiter will observe when attempting to cluster taint sets in the network core. Finally, we input all taint sets—$C \times B$ taint sets which have some taints in common, and N noise taint sets—to Eigencluster [6], an unsupervised spectral clustering algorithm, giving equal weight to each taint set. Table 1 tabulates the number of clusters identified by Eigencluster, the number of taint sets belonging to one of the C clusters that were grouped with taints from another cluster (False Negatives, or FNs), and the number of taint sets from ambient noise, N, that were grouped with one of the output clusters (False Positives, or FNs).

In each case, we expect: (1) Exactly C output clusters; (2) No FNs (*i.e.*, no element in a cluster was misclassified); and (3) No FPs (*i.e.*, no ambient noise was accidentally identified as belonging to a cluster). Our simulation shows that the clustering does pick out *all* $C \times B$ taints and cluster them (result not shown in Table 1), but the number of detected clusters is sometimes slightly greater than the number of input clusters. There are no false negatives (except when the common b taints within a cluster is overshadowed by the b random taints added later): taint sets of processes with common history will never cluster with another set of processes. The existence of false positives (*i.e.*, potentially legitimate taint sets grouped with a malware cluster) hampers blind filtering merely using the clustering output, but the bulk of the noise—potentially legitimate taint sets—*do not form clusters*. Because of the large reduction in "ambient noise", it may be practical for the operator to channel *all* taint sets from identified clusters to a deep packet inspection device to weed out any false positives.

6. Protecting Pedigree from Attack

This section discusses host and network-based attacks against *Pedigree* and defenses against them.

6.1 Host-based Threats

Pedigree is vulnerable to several host-based threats that might allow a process to interfere with tagging. We discuss threats in increasing order of the capabilities of a malicious process.

Threat 1: *Malware is a user-space, unprivileged process.*

For this threat, an administrator assumes that the only untrusted components of the operating system are unprivileged processes running in user-space. The tagger can then be deployed as a user-space, privileged process through which all outgoing (or incoming) packets are routed before they cross to (or from) kernel-space. Trusting all privileged processes may be reasonable for OS architectures where the amount

of trusted code is minimized (*e.g.*, embedded OSes, L4 Microkernel [18], Exokernel [7], HiStar [37], etc.) or special-purpose devices where the user cannot typically run processes with super-user privileges.

Threat 2: *Malware is a user-space, privileged process.*

If the kernel is trusted, the tagger can be a kernel module. Still, most operating systems also allow processes with super-user privileges to change various parameters of the running kernel, or even load and unload modules from the kernel; thus privileged malicious process might bypass the tagger by interfering with its parameters or even unloading the module entirely. To counter this attack, we suggest even a privileged process should present credentials (*e.g.*, a password) to the kernel to change kernel settings.

Threat 3: *Malware is a kernel module.*

If malware can gain kernel rights (*e.g.*, by inserting a kernel module), it can typically bypass, corrupt, or completely eradicate a software-based tagger. There exist two methods to deploy the tagger at a level below the kernel. First, the operating system, and all its processes, can be run inside a virtual machine [28, 29, 34]. In this case, the tagger will reside either in the hypervisor, or in the trusted domain (*e.g.*, Dom0 for Xen, the host OS in VMWare or UML). All untrusted programs will only be run in one of the untrusted domains. The tagger maintains the tags of processes and files of the untrusted OS outside the OS, and is therefore protected even if the untrusted OS is compromised. Second, the tagger can be deployed as a hardware chip (*e.g.*, within a Trusted Computing Platform) through which all incoming and outgoing packets are routed.

6.2 Network-Based Threats

Only hosts that do not have a tagger installed can attempt to mount attacks against *Pedigree*'s tags. Such hosts may eavesdrop for tags on local networks, or attempt to manufacture tags to gain access to some resource. Of course, if a host has not deployed *Pedigree*, then all information being carried in the tags will be lost when processes on this machine take inputs from tagged packets and generate untagged traffic of their own. ISPs can encourage users to deploy taggers in various ways (*e.g.*, providing preferential treatment of traffic).

Threat 4: *Eavesdropper attempts to spoof or replay tags (or credentials in tags).*

Although no host with a tagger can eavesdrop on tags, malicious hosts may eavesdrop on packets on the wire and mount attacks such as denial of privileges, replaying another host's packets, etc. To prevent valid tags from being replayed by eavesdroppers, *Pedigree* includes a re-initializable hash chains[9] as a cryptographic credential in the tag. Re-initializable hash chains (RHC) are a variant of finite-length one-way hash chains [17] that can be securely re-initialized once the hash chain is exhausted. For each connection, *Pedigree* sets up a new RHC. For the first packet in a connection, the tag includes the first value of the RHC. Additionally, the cryptographic signature included with the first packet is performed over the self-identifying message (comprising the GHID, AppID, taint set, source IP address, and source port) *concatenated with the first entry of the RHC*. As before, every subsequent packet carries the hash of the first packet's tag, as well as the next entry in the RHC. When the arbiter receives the first packet, it verifies the signature using the fields in the tag. After the signature is verified, the arbiter stores the current value of the RHC, and uses it to verify that new packets purported to belong to the same connection are not replays.

The eavesdropper may spoof other portions of the tag, including the global host ID (GHID) and the application ID (AppID). These fields are not cryptographically meaningful, and thus the arbiter will not typically use them to authenticate traffic. Still, if the tagger knows the arbiter's public key, it may encrypt *all* fields in the container ID portion of the tag. This approach requires the arbiter to have certified key pairs and key distribution services.

Because the taint set portion of the tag is typically a compact set representation that uses a probabilistic data structure (*Pedigree* uses a Bloom filter), membership queries to the taint may return a false positive. As result, we do not envision that the arbiter would use the taint set as a credential; thus, there would be very little incentive for an eavesdropper to replay a taint set.

Threat 5: *Eavesdropper changes tags to evade blacklisting, filtering, or other service degradation.*

Using container ID fields to filter or blacklist is reliable only when the network administrator is assured that all hosts on a network have a trusted tagger installed; otherwise, a malicious host may simply change the values of fields to evade the filter. This is a reasonable assumption in enterprises where all hosts run a trusted OS that has the tagger pre-installed.

In the wide-area, filters based on container IDs can be easily evaded since these IDs can be minted or forged. Thus, arbiters in the wide-area must use the taint set. A malicious process may still try to modify a taint set that was blocked by the arbiter in order to construct a different taint set that will be allowed by the arbiter. As a defense, the ISP might encourage or require its users to necessarily install taggers to receive normal service, and instrument "verified" taggers with a credential that the ISP's arbiter can verify; this credential may be a scheme to add a new taint to the taint set of *each* packet that exits the host. The arbiter can first verify credentials on the taint sets of packets before performing service decisions using the set. To prevent eavesdroppers from guessing the credential taint (perhaps by collating taint sets of many packets), the credential generator changes the credential taint in some fashion.

7. Prototype Implementation

This section describes the prototype implementation of

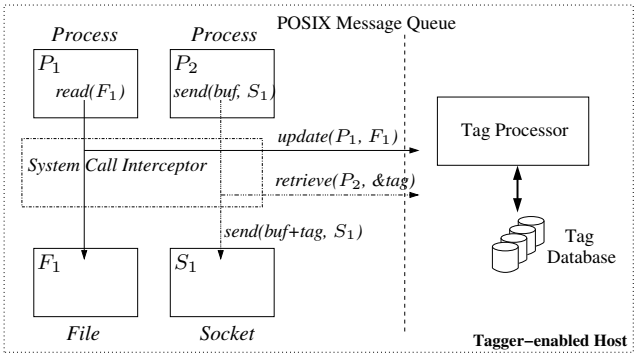

Figure 6: Architecture of the user-space *Pedigree* prototype. System calls that propagate tags are intercepted using a pre-loaded shared library, and the tag information of all active resources is stored in a central database.

Pedigree. Our prototype is geared towards testing the performance and overhead of *Pedigree* and to serve as a proof of concept; it does not attempt to be immune to malicious attacks. Section 7.1 presents an overview of the prototype, Section 7.2 describes the tag processor, Section 7.3 describes tag database structure.

7.1 Overview

We implemented a proof-of-concept prototype of *Pedigree* in user-space in approximately 10,000 lines of C code with 2890 semicolons; we chose a user-space implementation for ease of prototyping and testing.

Our prototype causes system calls made by applications (*e.g.*, `read(2)`, `write(2)`, `fork(2)`, `execve(2)`, `send(2)`, `recv(2)`, etc.) to be interecepted by a pre-loaded shared library, using which we track and record the propagation of tags between resources. The shared library also attaches tags of a process to the packets it generates, and strips tags from packets the process reads and incorporates them into the process's own tag.

7.2 Processing Tags

Figure 6 presents the architecture of our prototype. Because the prototype runs in user space, the tagger must be notified whenever a system call that might propagate taints is invoked. One way to implement this functionality is to augment the kernel system call handler to notify the tag-tracking process whenever such a system call occurs, but this technique requires two user-kernel crossings, which are expensive. Instead, we use the functionality offered by the dynamic linker on Unix-like OSes (*e.g.*, *ld-linux.so* in Linux) to pre-load additional shared libraries before programs are run (called "library interposing"), using the LD_PRELOAD environment variable.[6]

The interposed shared library wraps all system calls that propagate taints according to the rules specified in Section 4.1.2. When a process invokes one of the wrapped system calls, the code in the shared library sends a message to a special process, the *Tag processor* with details of the calling and called resource (*e.g.*, in Figure 6, when process P1 reads file F1, the library informs the Tag Processor of the read, including the process ID of the process and the inode number of the file). For some system calls, the interposed library may also expect a response from Tag Processor: for *e.g.*, during a `send(2) call`, packets must be affixed with the tags of the process that invoked `send(2)` before they are sent out to the network. At a high-level, the Tag Processor offers two functions: 1) `update (dst, src)`: updates the tags of resource *dst* with those of *src*; this operation is performed during `read`, `write` or `execve`; 2) `retrieve (src, &tag)`: retrieves the tag of the resource *src* into the structure *tag*, which is performed during *send(2)*. A variant of `update` is invoked when a network packet is received using `recv(2)`: instead of using a resource ID as *src*, the library uses the tag stripped from an incoming network packet.

[6]Our implementation does not protect against malicious programs that anticipate pre-loaded libraries and unset the LD_PRELOAD variable. Strictly speaking, there is no way to completely avoid "help" from the kernel in tracking system calls: even if we resorted to building the tag-tracking functionality into libc itself, a malicious program is free to link against its *own* copy of libc.

7.3 Tag Database

The tag processor uses a tag database to maintain the tags of all active resources in the system. The database is implemented as a two-level hierarchy consisting of an in-memory cache backed by an on-disk database. The on-disk database structure is maintained as two separate GDBM [8] databases, one for files (*i.e.*, permanent resources), and one for processes (transient resources). The files are indexed by the ID of the resource (*i.e.*, inode number for files and process ID for processes). Because process IDs are not related to to the *application* being run as the process, the database for processes does not have any permanent entries; the entry for a process is deleted after the process dies, and the whole file is cleared at system boot. The database for files, on the other hand, persists across reboots.

The in-memory cache is used to reduce the I/O overhead associated with accessing the on-disk database. The number of tags cached is a configurable parameter. In case of cache overflow, an LRU scheme is used to write through with the on-disk database. When the system halts/reboots, cached tags for files are transferred to the file database, while those of processes are purged. The tag processor communicates with the interposed library of a process making a syscall using POSIX message queues. Message queues automatically handle issues of concurrency due to multiple processes writing to it.

8. Performance Evaluation

This section presents the evaluation of our prototype of *Pedigree* implemented in user-space using library interposition. We evaluated the prototype on a 2.4GHz dual core machine running a POSIX-based GNU/Linux system with Linux kernel 2.6.22 and glibc version 2.6. We study the latency overhead of the prototype when interposing both local system calls (*e.g.*, `read`, `write`) and remote ones (*e.g.*,

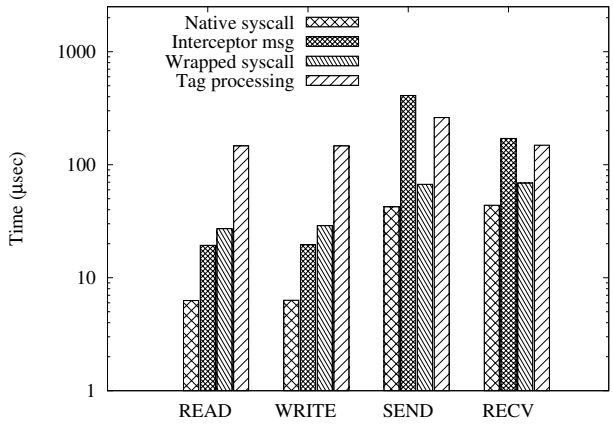

Figure 7: Latency overhead. *Interceptor msg time* **indicates the time taken by** *system call interceptor* **to enqueue a message.** *Tag processing time* **is the time taken by** *tag processor* **to process one message.**

`send`, `recv`), in order to demonstrate the feasibility of *Pedigree* on general purpose computers.

Latency Overhead. To evaluate intra-host system call overhead, we created two test applications — *test-read* and *test-write* — that perform 10,000 `read(2)` and `write(2)` system calls in a loop on an open file descriptor. To evaluate network overhead, we created a client/server file-transfer application to upload a 67MB file from client to server over a 1Gbps switched LAN, with both client and server using a 4096-byte buffer for sending or receiving data. We assume that tags of the client and server do not change for the duration of the file upload experiment, which is typically the case with most processes performing network data transfers.

Figure 7 depicts the latency overhead calculated for these test programs with and without library interposition. Note that *Process syscall time* does not include the *Tag processing time* as the tag messages are *asynchronous* (except `retrieve`). The latency overhead for wrapped syscalls is usually low, ranging from 1.5 to 4.5 times that of the native syscall, which is reasonable for a user-space implementation; an in-kernel implementation is likely to be several times faster. For `send(2)`, the *Interceptor msg time* exceeds that for other syscalls because `retrieve(src, &tag)` is synchronous, *i.e.*, it must wait for the tag processor to reply with the retrieved taint.

Message Queue Overflow. We noticed that the tag processor processes incoming messages slower than application processes enqueue them, resulting in the POSIX message queue becoming full. A full queue causes an application process attempting to enqueue messages to block (*i.e.*, the process is moved to the operating system scheduler's wait queue), leading to higher observed latency for the application process. Because blocking misrepresents the real overhead of the prototype, we performed measurements using a message queue large enough to not overflow during the course of the experiment.

Assuming an arbitrarily large message queue is, however, not practical: message queues remain in non-swappable kernel memory and their size is thus limited by physical memory. On the other hand, if the application process enqueues messages at a latency comparable to the rate at which the Tag processor processes them, even a small message queue would suffice for IPC. The test programs in our evaluation invoke system calls as fast as the operating system and hardware can field them without performing any processing on the data, while a real application would presumably process data before performing syscalls on the data.

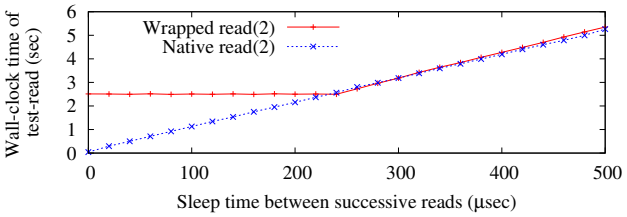

Figure 8: Experiment to measure message queue overhead in *Pedigree* prototype. The sleep time between successive read(2) is varied and the wall-clock time seen by wrapped and native *test-read* **is measured.**

To evaluate the impact of blocking on perceived application latency, we repeat the 10,000 loop `read(2)` using *test-read* with a smaller message queue of merely 100 messages. To simulate processing, we add a small delay after each `read` using the `usleep(3)` library routine. Figure 8 plots the time perceived by the *test-read* application as the delay is varied from 0 μsec to 500 μsec.

The latency experienced by native *test-read* begins at 0.05 seconds and increases linearly with delay, but the latency for wrapped *test-read remains the same at 2.5 seconds* from 0 μsec through 240 μsec. The reason for wrapped *test-read* not reacting to the explicit delay is because the the sleep time *overlaps* with the blocking time: wrapped *test-read* spends approximately 240 μsec per cycle on the scheduler's wait queue, and any delay below 240 μsec does not affect its overall wall-clock time. Beyond delays of 240 μsec, both wrapped and native *test-read* show linear increase in wall-clock time, with the difference between the two (approximately 0.09 seconds, or approximately 180% of native *test-read*) reflecting the *actual* overhead of library interposition.

The blocking latency of 240 μsec per `read` is comparable to (or less than) the processing time for many typical operations (*e.g.*, a graphical application takes milliseconds to redraw the screen; a file transfer application sending data from disk at 1 Mbps using a 4096 byte `read` buffer performs a `read` only once every 4 milliseconds), making even the overhead of our prototype (with small message queues) feasible for real applications [7] Implementing the tagger using shared memory will not involve message queue overhead, while an in-kernel implementation will completely eliminate the need for IPC.

9. Discussion

[7] The 240 μsec figure is the average blocking time between successive enqueues for the message queue across *all* processes, which may imply higher delays per process if many processes simultaneously perform syscalls in quick succession.

This section discusses some additional benefits that *Pedigree* provides, as well as various limitations.

9.1 Benefits

Better anomaly detection. *Pedigree* focuses only on tagging traffic, which we consider to be distinct from the well-studied area of *labeling* traffic (*i.e.*, as in traffic classification and anomaly detection systems) [14, 16, 10]. We believe that these systems could be much more powerful if the traffic itself carried taint sets indicating relationships among traffic coming from different groups of hosts. For example, the taint set might help an anomaly detection system realize that a group of flows were related because the sources had all communicated with one another, traffic flows coming from a group of hosts were all generated by a process that was running a common binary file, or that the traffic generated by some group of processes that were running with a certain level of trust.

Stronger host security. Administrators may want to track files and processes that may have affected any given process. Using *Pedigree*'s tags, system administrators can implement host-based more expressive security rules than they can today. An administrator who knows the taints corresponding to a certain malicious application can construct a host-based rule that prevents any process whose traffic carries the malicious application's taints from accessing critical resources, irrespective of the process' privilege level.

9.2 Concerns and Limitations

Connectionless protocols and route changes. Sending the hash of the tag to reduce packet overhead (as described in Section 5.1) will not work if either (1) the protocol is connectionless or (2) the route from the sender to receiver changes in the middle of a connection. It also imposes significant overheads on short connections. The taint set is large, but *Pedigree*'s container ID is small and can be used independently of the taint set; applications that were sensitive to this overhead could mark packets with only the container ID and still gain some of the benefits of *Pedigree*. To combat route changes, arbiters that receive unrecognized hashes of tags could send a challenge (*e.g.*, via an ICMP message) to the sender asking the sender to re-send the original tag.

Taint accumulation. A potential application of *Pedigree* is tracking worm outbreaks. Unfortunately, using the container ID (specifically, the AppID) will likely be infeasible for this purpose because most malware is polymorphic (*i.e.*, each copy of the binary takes a slightly different form). When a worm creates a slight variant of a copy of itself, however, the taint set for the new copy of the worm will contain the taint set of the old worm, which may make tracking easier. Additionally, *Pedigree*'s tagger could be augmented with a special function that copies a hash of the old binary's image into the new binary's taint set in cases where an application creates a file that is similar to its own image.

Many applications, particularly those that maintain persistent state in configuration files across sessions, may acquire a very large number of taints over time. As such a taint set fills up, performing certain types of operations, such as tracking malware, become difficult: if such an application writes malware to disk, the malware will acquire all of the taints of that application. We believe the best defense against this is to mandate that taint sets do not become too full using the techniques we described in Section 5.3.

Issues with partial deployment. As discussed in Section 6, when traffic that is carrying tags is passed through a host that has not been instrumented with a tagger, the traffic's "audit trail" is lost. In these cases, a common worm outbreak (or communicating group of hosts) might appear to an arbiter or network monitor as distinct groups of hosts. Examining the taint sets of each of the subgroups, however, might allow an operator to recognize that taint sets from each group either share many taints in common. In our future work, we plan to study various methods for recovering taint sets in cases of partial deployment.

10. Related Work

Resource interaction tracking. *Pedigree* is inspired by ideas related to tracking interactions between resources at the host level. Many research operating systems have attempted to secure the system against exploits or to prevent security breaches (*e.g.*, exfiltration). Early research goes back many decades (*e.g.*, the Hydra operating system [4]), and researchers continue to tackle related problems (*e.g.*, Taos [30], and more recently, HiStar [37]). TaintCheck [20] also monitors information flow, albeit at the instruction level, in order to detect potential exploit code.

Access tracking systems such as Tripwire [27] perform interaction tracking *and* detection; in contrast, *Pedigree* leaves classification decisions to a separate arbiter. Perhaps the closest work to *Pedigree*'s resource tracking scheme is "process coloring" [12], where the authors propose tracking interactions between resources ("color diffusion"). Their work, however, uses interaction tracking for early detection of resources on a host that possess "colors" of a vulnerable process, (much like *Pedigree* can be used to strengthen host security as discussed in Section 9); *Pedigree* is unique in extending resource tracking *across* hosts using taint sets.

Traffic classification. *Pedigree* is primarily a traffic classificiation system, and attempts to address some of the concerns that similar systems address: (1) application identification agnostic to IP or port using systems such as BLINC [14], or using statistical techniques [19]; (2) identifying session structure in traffic [13], etc. In contrast to these systems, *Pedigree* does not use network-level characteristics to identify applications, but instead relies on the trusted tagger on hosts to record and transmit the tags of processes generating packets. However, *Pedigree*'s clustering scheme (Section 5.4) can use the hints that these systems provide in order to tune the input to the clustering algorithm (*i.e.*, cluster the taint sets only of unknown applications).

Tracking worm outbreaks and intrusions. Tracing the origin of worm outbreaks can help identify miscreants. Ran-

dom moonwalks [35] is representative of this body of work; the authors use random walks to traverse edges of the graphs constructed from traffic logs backwards in time in order to identify the origin and propagation of worms. Many systems such as BackTracker [15] use dependency graphs to backtrack attacks. In *Pedigree*, explicit backtracking or tracing worm origins is not possible because a tag's taint set cannot enumerate the taints that were inserted into it; however, an arbiter that sees a wide range of traffic can extract the minimal taint sets using clustering (Section 5.4) to identify groups of related hosts.

11. Conclusion

We presented *Pedigree*, a system for expressive, semantically-rich traffic classification. *Pedigree* relies on the cooperation of trusted *taggers* on end-hosts to affix tags to network packets. Tags extend the provenance information that already exists within a host to the network. These tags may comprise fixed identifiers of the end-host, the application generating packets, the credentials of the process generating packets, and the *taint set* indicating the process's history (*i.e.*, information about other processes, files, or remote hosts that may have affected the process), These tags can then be used by network elements (*arbiters*) for a variety of applications, including: access control and authentication, provisioning, preventing exfiltration, and tasks for reducing unwanted traffic such as filtering, rate-limiting, or dropping. *Pedigree* lets network operators use the same tag structure for both upgrading and downgrading service, blurring what we see as an artificial distinction between "on-by-default" and "off-by-default" network capability design paradigms. Our evaluation of a user-space prototype of *Pedigree* shows that packet and storage overhead is negligible for all but short connections.

REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[2] Bobax Trojan Analysis. `http://www.secureworks.com/research/threats/bobax/`.

[3] Burton H. Bloom. Space/time Trade-offs in Hash Coding With Allowable Errors. In *Communications of the ACM*, volume 13, pages 422–426, 1970.

[4] E. S. Cohen and D. Jefferson. Protection in the hydra operating system. In *ACM Symposium on Operating Systems Principles*, pages 141–160, 1975.

[5] Dark Reading. Malware Quietly Reaching Epidemic Levels. `http://www.darkreading.com/document.asp?doc_id=143424`.

[6] David Cheng and Ravi Kannan and Santosh Vempala and Grant Wang. A divide-and-merge methodology for clustering. *ACM Transactions on Database Systems*, 31(4):1499–1525, 2006.

[7] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *ACM Symposium on Operating Systems Principles*, pages 251–266, 1995.

[8] GNU dbm. http://www.gnu.org/software/gdbm/.

[9] V. Goyal. How to re-initialize a hash chain. IACR 2004, 2004.

[10] Haakon Ringberg and Augustin Soule and Jennifer Rexford and Christophe Diot. Sensitivity of PCA for Traffic Anomaly Detection. In *Proc. ACM SIGMETRICS*, San Diego, CA, June 2007.

[11] IBM Secure Hypervisor. `http://www.research.ibm.com/secure\_systems\_department/projects/hypervisor/`.

[12] X. Jiang, A. Walters, F. Buchholz, D. Xu, Y.-M. Wang, and E. H. Spafford. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. In *ICDCS*, June 2006.

[13] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-automated discovery of application session structure. In *Proc. ACM SIGCOMM Internet Measurement Conference*, pages 119–132, Oct 2006.

[14] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: multilevel traffic classification in the dark. In *SIGCOMM*, pages 229–240, 2005.

[15] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, 2005.

[16] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[17] L. Lamport. Password Authentication with Insecure Communication. *Communications of the ACM*, 24(11):770–772, 1981.

[18] J. Liedtke. On micro-kernel construction. In *ACM Symposium on Operating Systems Principles*, pages 237–250, 1995.

[19] A. W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. In *Proc. ACM SIGMETRICS*, pages 50–60, 2005.

[20] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[21] OpenVZ Server Virtualization. http://openvz.org/.

[22] P. Deutsch. *DEFLATE Compressed Data Format Specification Version 1.3*. Internet Engineering Task Force, May 1996. RFC 1951.

[23] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006. An earlier version appeared as Georgia Tech TR GT-CSS-2006-001.

[24] A. Ramachandran, N. Feamster, and D. Dagon. Revealing Botnet Membership with DNSBL Counter-Intelligence. In *2nd USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, San Jose, CA, July 2006.

[25] Simon W. Golomb. Run-Length Encodings. In *Transactions on Information Theory*, volume 12, page 399, 1966.

[26] Storm Worm DDoS Attack. `http://www.secureworks.com/research/threats/storm-worm/`.

[27] Tripwire Configuration Audit. http://www.tripwire.com/.

[28] User Mode Linux Kernel. http://user-mode-linux.sourceforge.net/.

[29] VMware virtual machine. http://www.vmware.com/.

[30] E. Wobber, M. Abadi, and M. Burrows. Authentication in the taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[31] ISPs fight against encrypted BitTorrent downloads. `http://arstechnica.com/news.ars/post/20060831-7638.html`.

[32] 2006 Operating System Vulnerability Summary. `http://www.omninerd.com/articles/2006_Operating_System_Vulnerability_Summary`.

[33] Storm Botnets Using Encrypted Traffic. `http://www.securitypronews.com/insiderreports/insider/spn-49-20071016StormBotnetsUsingEncryptedTraffic.html`.

[34] XenSource Home. http://www.xensource.com/.

[35] Y. Xie, V. Sekar, D. A. Maltz, M. K. Reiter, and H. Zhang. Worm origin identification using random moonwalks. In *IEEE Symposium on Security and Privacy*, pages 242–256, 2005.

[36] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[37] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazires. Making information flow explicit in histar. In *Symposium on Operating*

Systems Design and Implementation, pages 263–278. USENIX Association, 2006.