Neon: System Support for Derived Data Management

Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable
Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage
University of California, San Diego

Abstract

Modern organizations face increasingly complex information management requirements. A combination of commercial needs, legal liability and regulatory imperatives has created a patch-work of mandated policies. Among these, personally identifying customer records must be carefully access-controlled, sensitive files must be encrypted on mobile computers to guard against physical theft and intellectual property must be protected from both exposure and "poisoning." However, enforcing such policies can be quite difficult in practice since users routinely share data over networks and derive new files from these inputs – incidentally laundering any policy restrictions. In this paper, we describe a VMM system called Neon that transparently labels derived data using byte-level "tints" and tracks these labels end-to-end across commodity applications, operating systems and networks. We demonstrate that this mechanism allows the enforcement of a variety of data management policies, including data-dependent confinement, intellectual property management, and mandatory I/O encryption.

1 Introduction

Information wants to be free because it has become so cheap to distribute, copy, and recombine—too cheap to meter. It wants to be expensive because it can be immeasurably valuable ... That tension will not go away.
– Stewart Brand, 1987.

Two decades after Brand's prescient statement, the value of information, the ease of manipulating it and the complexity of managing its use have only become greater. Today commercial corporations, non-profits and governmental bodies alike mandate a wide range of information management policies that govern who may access data, how it may be accessed, and for what it may be used. However, in today's interconnected computing environment such policies are generally far easier stated than they are enforced. Indeed, entire industries have emerged around providing and validating different kinds of information "policy compliance" within the enterprise.

Consider the simple policy "*any customer records should be encrypted on disk.*" Implicit in this statement is the assumption that customer records can be easily identified and that there is a policy control in place that forces the encryption of such information. However, in practice, neither is usually true. Most control mechanisms mediate access to objects (such as files) and not the information contained therein. Moreover, commodity applications, operating systems and networks provide little means for tracking the flow of information *between* such objects. Returning to the policy example, while some operating systems do provide interfaces to specify that a particular file should be transparently encrypted (*e.g.*, Windows XP's EFS), this policy does not carry over to *derived data*. Thus, if the file is compressed, if records are "cut and pasted" into a new file, if it is sent over the network, if it is attached to an e-mail, or if it is subject to any of a myriad of normal data manipulations, the resulting data will be be laundered of any connection to the original encryption policy. Put succinctly, operating systems and applications generally provide controls only over information *containers* – such as files or network connections – but not over the actual data they contain. This failing is the Achilles heel that undermines the practical enforcement of virtually all information management policies.

Solving this problem – without requiring changes to existing binary applications or operating systems – requires a transparent system-wide mechanism for tracking information flow. Moreover, for today's distributed environment, it is insufficient to simply track information flow on a single machine, but this capability must apply transitively across the network as well. In this paper we describe how such a mechanism can be integrated within a virtual machine monitor (VMM) and used to enforce a wide range of practical information management policies in a distributed enterprise environment. We demonstrate this approach using a prototype system, called *Neon*, that implements byte-level policy labels, called "tints," that are transparently propagated and combined as part of normal instruction execution. Critically, this capability requires no changes to applications or operating systems.

Moreover, Neon propagates tints across the network and to and from storage, thus maintaining the binding between a policy and any derived data. For example, consider the following scenario: a user opens a file over NFS using emacs, edits its contents, selects a portion and then pastes it into a separate mail application – in a different window – then encrypts the message and sends it to a third-party via SMTP. If the source file was tinted "red" (an arbitrary designation) then packets containing the message will be tinted red as well (at least in part). Thus, if the red tint was meant to indicate confinement, the policy can be enforced by dropping red packets. We have used Neon to easily implement a variety of such information management policies, including network-level confinement and access control, mandatory encryption, enforced virtual private network use and licensing compliance for compiled code.

This paper is structured as follows. We motivate the need for transparent information flow tracking and describe closely related work in Section 2. Next we describe our overall design in Section 3 followed by a description of Neon's implementation in Section 4. The remainder of the paper, Section 5, describes the baseline overhead added by the Neon prototype and describes how the system is used in a variety of applications. We summarize our findings and conclude in Section 6.

2 Background and Related Work

The need for information management policies is driven by a range of concerns. These include protection of trade secrets, third-party liability, and corruption of intellectual property. Moreover, compliance with regulations such as the U.S. Health Insurance Portability and Accountability Act (HIPAA), the U.S. Sarbanes-Oxley Act and the European Parliament's Directive 95/46/EC all require restrictions on the use and disclosure of particular forms of data and their derived counterparts. Finally, the combination of data loss disclosure laws (such as California's SB 1386) and increased attention to attendant issues such as identity theft and data piracy have focused considerable attention on these issues.

Moreover, these are not mere hypothetical concerns, but reflect growing reports of misappropriation, loss, or misuse of data that should have been appropriately protected. For example, in one highly publicized case, a staff member in the office of the U.S. House Majority Leader was able to exfiltrate Judiciary Committee documents from the opposing political party by encapsulating them within e-mails to a separate account [26]. Even more common are accidental exposures of private data, either from employees who misunderstand the policy governing such data or through inadvertent sharing or accidental sharing. For example, many popular applications implement versioned file formats that may include information from previous versions even if it has been deleted, or blacked out, using the application. Similarly, many file sharing applications or indexing services are easily misconfigured to export potentially sensitive data without user knowledge.

Physical data mobility presents additional challenges and stolen laptop computers have exposed a wide range of sensitive data stored therein, including millions of personnel records [21], classified intelligence documents [22] and credit card information [25]. Indeed, in a 2006 CSI/FBI survey of Fortune 1000 companies, nearly half of all losses due to cyber security incidents were the result of data loss/leakage of some type [14].

Finally, unmanaged data infiltration can sometimes cause as many problem as data leakage. In particular, the copyright of a software program is derived from the provenance of its constituent source files. However, the success of the open source movement has made it easy to share source code across the Internet – potentially undermining the qualities of derived works, open source and proprietary alike. Indeed, violations of the Free Software Foundations Gnu Public License are commonplace in commercial products and yet most of these violations are accidental [13].

In general, simply determining who is allowed access to which information and how they may use it is difficult enough, but enforcing such policies in a distributed environment poses significant challenges. Underlying this challenge is the impact of networked computers and the client-server architecture that together decentralize control over information. A document is rarely stored in any single location, but may also be attached to e-mails, posted to Wikis or Web pages, copied onto employee computers and laptops, indexed and copied into a database, and so on. Thus, there are naturally a wide range of points over which a policy must be enforced.

2.1 Existing Solutions

The commercial marketplace has responded to these problems individually. For example, a new market – including companies such as Vontu, Port Authority, Oakley Networks and Fidelis – has emerged around "data leakage" protection. Users of these systems typically identify confidential information, which is then transformed into content fingerprints used to lexically scan network traffic or application I/O requests to the operating system. Thus, these systems typically operate under the assumption that derived data will be textually similar to a corpus of strings being protected. These approaches have the benefit of detecting violations independent of how they were generated (thus even data manually entered from paper documents can be detected). However, these solutions are inherently limited in dealing with derived data and data transformation. For example, a file that is encrypted will bear no lexical similarity to the source file. [1] Similarly, simple actions such as exerting small amounts of data (*e.g.*, a single cell in a spreadsheet) or converting a spreadsheet into an image or Adobe PDF file, may make such analysis impossible. These systems also rely upon a representative set of sensitive content and fuzzy fingerprinting to identify content which may not detect lexically innocuous content.

To address the problem of laptop theft, organizations are increasingly requiring sensitive data to be encrypted on mobile drives. This policy can be implemented either as a software layer in file system (as with Windows EFS) or block device interface (as with WinMagic's SecureDoc) or in hardware in the hard drive itself (as with new drives being manufactured by Seagate, LaCie, HDD and Stonewood electronics) [17]. The former represents a protection that is too lenient (encrypted files can be easily laundered of their encrypted status) or too severe (unimportant files are subject to the overhead of encryption and the data loss risks associated with user-managed authentication keys).

A similar situation exists for use of the network. Organizations are increasingly configuring laptops that enforce mandatory use of corporate virtual private networks (VPN). While this ensures that private data on a laptop will be protected on the Internet and will be subject to a corporations data inspection mechanisms (as above) it can also create undue burden for access to public data sources. Mandatory VPN use can be incompatible with the firewall rules of local networks and incurs unnecessary latency (and hence reduction in throughput due to the congestion control behavior of the Transmission Control Protection (TCP)).

Finally, a completely different market has emerged around validating the IP provenance of source files in software development efforts. For example, companies such as Black Duck and Palamida provide "IP compliance" software that lexically matches source lines against large corpora of known open source projects. The strengths, and weaknesses, of this approach are similar to data leakage products using similar techniques.

In general, the weaknesses in all of these approaches relate to their inability to track the flow of information independent of its representation. To wit, if it could be determined unambiguously that a given data object was *derived* from a source that requires confinement, encryption or a compatible copyright license, enforcing the associated policy would be vastly simplified.

2.2 Information Flow Tracking

The notion of tracking information flow arose over thirty years ago in the context of security policy enforcement [8]. One major thrust of subsequent work, first proposed by Denning and Denning, has been the use of static analysis to provide efficient, high-precision flow tracking [9]. In this community, one of the best known examples of this approach is Myers and Liskov's model for static type-checking of program information flow labels [19]. The subsequent Jif compiler, which operates on an annotated version of Java, allows the combination of both static and run-time checking to support dynamic information flow labels [18].

However, despite the benefits of native language and compiler support for information flow, virtually all systems and applications are written using languages and compilers without such capability. Thus, another major body of work has focused on the use of binary rewriting to provide transparent dynamic information flow tracking for existing binaries. Most of this work has been specifically motivated by control flow hijacking attacks, such as buffer overflows, and typically involves the "tainting" of program inputs, the dynamic propagation of taint through program execution, and the trapping of control transfers to tainted target addresses [5, 16, 20]. These approaches add significant overhead since each operation that *potentially* propagates information flow must access an ancillary data structure. Even the fastest of such systems introduces typical slowdowns of over 500% [3]. Moreover, binary rewriting systems typically only track data-dependent information flow within a *single* program and are not trivially extended to track information flow between applications and operating systems.

Recently, several systems have implemented whole-system taint tracking, using either instruction emulation [6, 4], dynamic translation [23], or combinations of these with virtual machine monitors [15]. This last approach, which is the basis for our own implementation (described more fully in Section 4), can potentially run at full speed when accessing untainted data and only slows to propagate taint information. Thus, the total slowdown is a function of the workload and is typically below a factor of two (although the worst case can be more than fifty times worse).

To address the overhead of dynamic tainting, the computer architecture community has investigated the use of dedicated hardware support for dynamic information flow tracking. Typically these designs can reduce overhead to roughly 1% by adding an individual taint bit to each register and storage location and automatically propagating taint during memory and arithmetic operations [6, 27, 30]. A critical evaluation of such approaches can be found in [7].

As with the binary rewriting systems, most architectural proposals have focused singularly on detecting control hijacking attacks (hence only necessitating a single taint bit and permitting a variety of propagating heuris-

[1] As a fallback most of these systems are equipped to detect high-entropy data transfers.

tics for pointer arithmetic). One exception is the RIFLE architecture which, similar to our system, anticipates the value of a range of information flow labels combined through computational dependencies [28]. RIFLE further tracks implicit information flow that arises through control dependencies, such as conditional branches, on tainted operands. We view all of these architectural approaches as complementary to our own and the availability of even limited hardware support for information flow could clearly be used to improve the performance of our system.

Finally, a number of systems have explored making information flow a first-class operating system abstraction. For example, the Asbestos and Histar systems export first-class information flow labels that are explicitly managed by programmers and directly interpreted by the operating system [12, 31]. This permits both a very efficient implementation of information flow tracking and allows application semantics to be tightly and precisely bound with the information flow policies of interest. This part of the design space is of great interest in influencing the construction of future operating systems, but is not well-suited to enforce fine-grained information flow between legacy applications and operating systems.

3 System Design

As we have discussed, the design space for tracking information flow is large and mirrors the range of applications and constraints faced by system builders. We have no illusions of providing a fully general approach, but instead have designed our system around what we believe are the needs of the existing enterprise environment.

3.1 Design Goals

Our design is driven by the following goals and constraints:

- *Transparency.* While there is a range of approaches for tracking information flow, many of them require changes to applications, operating systems or both. By contrast, we focus exclusively on supporting legacy environments and thus only consider information flow mechanisms that are transparent to the applications and systems they monitor.

- *Fine-grained tracking.* Since our need for transparency precludes access to application semantics, we cannot precisely name objects or data structures. To make up for this limitation, we track information flow at the granularity of individual bytes. We could have chosen a more coarse-grained approach, such as file-level tracking, but this would not track shared-memory communications (*e.g.*, such as to cut buffers or to window managers) and would require excessive conservativism for applications that manage multiple files (*e.g.*., editors or mail transfer agents).

- *I/O enforcement.* In a traditional reference monitor, an enforcement predicate can be evaluated at arbitrary granularity (*e.g.*, each instruction). However, such an approach is overkill in the enterprise environment, since each client host is typically used by a single user at a time and data is only shared via the network or storage. Thus, in our design, we enforce information flow policies exclusively during I/O. This approach also simplifies any implementation since low-level I/O actions typically use canonical representations (*e.g.*, disk block, IP datagram) that are straightforward to interpret.

- *Orthogonal policies.* While traditional taint tracking systems focus on enforcing a single policy – preventing control flow hijacking – we believe that there are a range of distinct information management policies that would offer real-world value if enforceable. Moreover, many of these policies are potentially orthogonal – a piece of data may require encryption, network confinement, reference counting (who currently has this data), limited data lifetime, and so on. Thus, we require the ability to track multiple information flows and combine policies for data derived from multiple sources.

Equally important as our design goals, are our non-goals – what we do not hope to accomplish in this paper. In general, we do not hope to enforce policy against sophisticated adversaries. As Butler Lampson is fond of saying publicly, “In computer security, the perfect is the enemy of the good”. Our focus is on tracking information flow within conventional applications, systems and networks, being used in the manner they were intended for. In particular, we do not strive to prevent covert or hidden communication channels that might defeat information flow tracking (trivially an insider might write down information and then re-type it from notes). Moreover, we are not currently concerned with adversaries who write code to launder data dependencies into control dependencies, although we recognize that information flow tracking purely based on data dependencies is subject to this threat. [2] Given that one recent survey of 145 information security breaches implicated employee negligence in over 75% of incidents, our focus seems appropriate [10]. Finally, while acceptable performance is always desirable, performance optimization is not a goal of this paper. Our focus is to show the general applica-

[2] For example, an adversary able to introduce new code could trivially launder tint using a simple program: `foreach bit b in the input; if (b == 1) then output 1 else output 0;`

bility of our approach rather than to demonstrate the best method for implementing it.

3.2 Abstract Design

In our design, each byte of memory is associated with a separate 2^n bit label called a "tint" (a pun on taint). Each bit position is used to represent a distinct policy and thus a tint can represent any combination of n policies. Data is originally tinted using a special administrative tool. We envision this tool will be restricted to carefully managed file servers, which will subsequently make data available to other clients.

Upon loading any tinted memory location, the associated tint is automatically propagated to any target register or memory location. Similarly, stores from a tinted register propagate the tint value to the target address. As well, stores from untinted registers untint the target, ensuring that tint does not grow forever. The target operand of arithmetic instructions is tinted with the logical OR of its source operands. Colloquially, if one adds the contents of a *blue* register and a *red* register, the target is tinted *purple*. As described earlier, our initial design does not not attempt to handle implicit control flow arising from conditional branches on tinted registers. This is a difficult problem to handle precisely and the naive approach – assigning the tint of the branch operand to all subsequent instructions – is needlessly conservative. However, we have not found situations where this sort of transformation occurs within our design target of normal application and system use.

When a tinted buffer is provided to the network or disk interface, its corresponding tint should be associated with the data. For example, each network packet could carry a header specifying the tints of its encapsulated data bytes. Similarly, an inbound I/O labeled with tint must be correctly propagated to the receiving buffer.

We assume that the memory holding tint values is protected from the operating system and application software being run and thus is only changed through normal information flow activity. Moreover, we assume that the network is secure and protected from tint manipulation (this restriction can be relaxed if we permit cryptographic integrity checks on each packet). Finally, we assume that all systems on the network participate in monitoring information flow. This assumption can be relaxed if each packet is authenticated and encrypted with a secret key known only to participating systems. In this case, non-participating machines would be able to exchange untinted data, but would be unable to receive tinted packets or generate valid instances of newly tinted data.

4 Implementation

In this section we describe the Neon prototype system, its construction and the tradeoffs we made for ease of implementation. We specifically describe how memory tinting is represented, how it is transparently propagated both on local hosts and across the network, how we introduce tint to the system and how we enforce tinting policies.

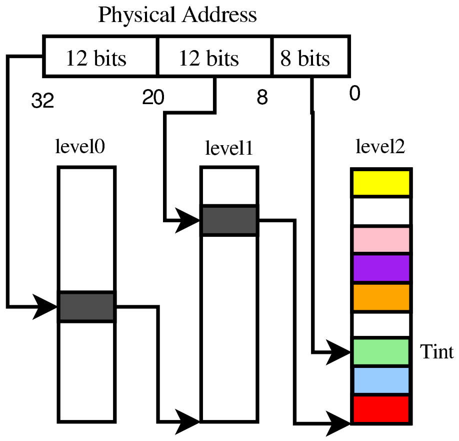


Figure 1: Tint Table Data Structure

Our implementation is based on the 3.0 release of the Xen Virtual Machine Monitor (VMM) [11] combined with the demand emulation modifications of Ho et al. [15]. In this environment, applications normally execute natively on the raw hardware as does the operating system with minor modifications to efficiently interact with the VMM. Later versions of Xen, working in concert with architectural extensions such as Intel's VT and AMD's SVM, provide support for a fully–virtualized hypervisor. Our use of paravirtualization is incidental and does not bear on the system implementation in any significant way.

A processor emulator provides demand emulation – QEMU [1] – that executes in a privileged VM context (dom0). Thus, when individual instructions must be emulated to propagate tint, control flow is redirected through the VMM to the emulator along with a comprehensive description of the current processor state (roughly 350 bytes in total). Exiting the emulator occurs in the same fashion, vectoring through the VMM to restore the updated processor context.

4.1 Neon Data Structures

Neon maintains byte-level tint for each *machine byte* (*i.e.*, a byte interpreted as a physical address by the guest OS) that is tinted. We allocate this memory dynamically using a multi-level table, similar to a page table, to maintain a compromise between space efficiency and lookup overhead for sparse allocations. Thus, as shown in Figure 1, the first 12 most significant bits of a byte address index into a first level table, which in turns uses the 12 bits to reference a second-level table each entry of which points to an array of 256 tint elements. Each tint element is 32 bits wide, allowing 32 distinct tints to be represented orthogonally – a number that significantly exceeds our present need for distinct policies. Thus, in the worst case – if every byte were tinted – tint overhead could be four times that of allocated memory. However, it is likely rare in practice that all tint combinations are in use. Thus, an obvious optimization is to distinguish between a canonical tint representation (32 bits) and a compact label that is simply an index into specific tint combinations that are in use. In our experiments such an optimization could reduce memory overhead by a factor of five or more. However, for ease of debugging the system we have solely used the larger canonical representation.

To speed lookups and help implement *tint faults* in the guest VM, we also maintain a per-machine page bitmap indicating which machine pages contain at least one tinted value.

Finally, we provide a fixed array data structure to track 32 bits of tint for each data register. We do not track precise tint through control registers since these should never be loaded with tinted data; thus a single bit – corresponding to any non-zero tint value – is sufficient to detect this case.

All of these data structures are implemented in *Domain 0*, within the address space of the QEMU emulator. We choose this location for convenience since it did not require any interfacing with the VMM; however it imposes unnecessary overheads when performing I/O and handling tint faults, as we will discuss later.

4.2 Propagating Tint Locally

Propagating tint requires the invocation of the emulator and thus a means to involuntarily transfer control from the VM when a tinted address is accessed – a *tint fault*. While the Intel architecture does provide a means for byte-level address traps through the *DR0-DR3* registers, this mechanism is limited to only four addresses at a time and thus is only appropriate in very limited settings (in which less than 16 distinct bytes require monitoring).

Instead, the demand emulation code approximates this mechanism by unmapping machine pages containing tinted values. Upon such a page fault, control is vectored to the emulator (since this is where the tint data structures are stored) and the target address is checked to see if is tinted. If so, the address is used as part of a memory load then the destination register's tint value is replaced with that of the target. If the address is part of a store then the destination addresses' tint value is set to that of the source register (typically zero if this is the first such fault). Regardless, subsequent execution proceeds within the CPU emulator (system calls and synchronous faults are vectored back through the VMM to the guest virtual machine) obeying the same tint propagation rules. When memory addresses are untinted (and associated data structures reclaimed) they are overwritten with a literal or untinted source register.

In addition, we have modified QEMU to propagate tint across instructions that maintain both source and target data registers. Thus, tints may be combined as a side effect of arithmetic operations or address arithmetic (such as indexed addressing). We do not currently implement idiomatic optimizations such as removing tint as a result of `xor reg, reg` or `sub reg, reg` optimizations [4]. To exit from emulated mode we use the heuristic of Ho et al. and do so after 50 memory references have not accessed tinted data *and* all live registers are free of tint. This heuristic is admittedly untuned and can introduce significant overhead when sparsely tinted data is located on pages with high reference locality. In the worst case, accessing a page containing tinted data can incur long emulation overheads even if the tinted data is never accessed itself. Similar to the *false sharing* problem in multi-processor memory coherency protocols, this arises from the mismatch between the granularity of the faulting mechanism and the granularity of access. Finer-grained memory fault mechanisms, such as Qin et al's ECC-based trapping would reduce this mismatch and hence reduce unnecessary emulation overhead [24].

4.3 Propagating Tint Remotely

To ensure that data tints propagate remotely, we invoke the QEMU process for each tinted outbound packet buffer and insert the associated tint into the packet header. For convenience, we reuse the 8-bit "Type of Service" (ToS) field in the IP packet header to specify tint values. The tint encoded in the TOS field applies to a packet's entire contents. Since packet reception and forwarding are atomic events, this coarse granularity makes little difference for the implementation of enforcement actions. However, if the recipient of the data uses it to further propagate tint this may lead to spurious tinting. A finer-grained packet tinting representation is possible, but could require variable length packet headers and the associated software complexity.

Inbound packets are handled in a similar fashion. Packet buffers are vectored to the QEMU process where their header fields are inspected for tint labels. If so, this tint propagates to the buffer containing the packet, which is mapped into the address space of the guest OS and propagates normally thereafter.

4.4 Creating Tint

Our system provides no native means to introduce tint on a client workstation. We envision tint as being managed centrally over the set of files needing specific protections. Thus, all tint in our system originates, *deus ex machina*, from a modified NFS file server.

Administrators of the file server can set per-file tint explicitly through an interface that overloads the file's GID field. Moreover, files that are written from clients store their tint values in a similar fashion. Again motivated by convenience in prototyping, this approach allows easy modification of tint using the existing `chgrp` program.

We choose not to modify the NFS server code itself but instead implement a network-level filter for packets arriving to or leaving from the server. In particular, we use netfilter to queue all incoming and outgoing NFS packets to a user-space application which is able to interpret packet-level tint representations [29]. This application, called *nfswall*, performs two tasks: tint marking and tint propagation. It can monitor, mangle, and in some cases drop selected NFS packets. For simplicity, nfswall only supports NFS over UDP because the boundaries between RPC calls are clearly identified.

4.4.1 Tint Marking

Nfswall does tint marking by monitoring NFS traffic and setting per-packet ToS bits appropriately for read reply messages on tinted files. The procedure type of an NFS reply message is not directly stored in the packets of the reply; therefore, nfswall also monitors the RPC call stream. We uniquely identify each RPC call by its RPC XID, source IP address, and UDP source port stored in an *xid_entry* structure. We use a *call_entry* structure to store the procedure number of the call and the file handle the operation is called upon. All call_entries are stored in a hash table keyed by their corresponding xid_entries. Together, these allow the procedure of each NFS reply to be identified – and tint is propagated solely via read reply messages. Since NFS version 3 includes file attributes in the reply of all successful operations [2], we extract the GID from the read reply packet to check its tint. The 8-bit ToS field is not large enough to hold all possible GIDs; hence, we map a small range of GIDs (GID 1001-1255) as tinted (ToS 1-255) and all other GIDs as not tinted (ToS 0).

NFS replies are often larger than the link layer MTU forcing the server to fragment them. We track read reply fragments to appropriately mark their tint. We distinctly identify a set of fragments using their IP source address and IP identification fields and store them in a *frag_entry* data structure. When nfswall encounters a read reply with the "more fragments" (MF) flag set, it stores the tint in a hash table keyed by its frag_ entry. Since iptables cannot filter IP fragments based on the target port number, nfswall receives all IP fragments leaving the system rather than just those on the standard NFS port 2049. Nfswall forwards any fragment that is not matched in the hash table because it assumes it to be untinted NFS or another protocol.

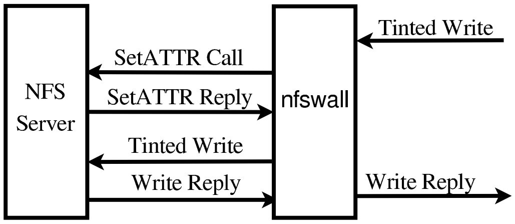


Figure 2: NFS Tint Propagation

4.4.2 NFS Tint Propagation

When a remote Neon client issues a write on a file or creates a new file on the NFS server, the tint of the file may be different than the GID on the server because of mixing with other tinted data. To remain transparent, nfswall keeps its own table of file attributes keyed by the file's NFS file handle and interacts with the file system only through NFS calls. It uses a zero padded opaque definition of a file handle from the NFS RFC to allow compatibility with any RFC compliant NFS server. Nfswall tracks `getattr` replies to populate the file table with the current attributes for each file. The attributes in the file table may be out of date due to modifications to the file locally executed on the NFS server. On an NFS write, nfswall knows what the most up to date GID *should* be; therefore, it only uses this table to reduce unnecessary updates to the GID.

Upon receiving an NFS write call, nfswall checks if the write call is tinted. If it is not tinted, nfswall never explicitly clears the tint from the GID on the file server. Due to the difference in granularity between Neon hosts (byte-level tint) and the file server (file-level tint), an untinted write does not imply that the entire file is untinted. If the write call is tinted, nfswall looks up the file handle in the file table and checks it against the GID stored in the table. When there is a mismatch between the GID of the write and the GID stored in the table, it sets the GID on the file server to the new value. The GID is propagated to the server *before* the tinted write to prevent another host from accessing improperly marked tinted data as shown in Figure 2.

To update the GID of a file, nfswall generates an NFS `setattr` call which appears to originate from the client issuing the write. The user performing the write must have permission to change the GID because nfswall reuses the file handle and RPC credentials from the write call. We use a raw IP socket to deliver the spoofed `setattr` packet directly to the local nfs daemon. Nfswall randomly generates an RPC xid for the spoofed packet which should not collide with another simultaneous request from the same host with very high probability. To make the spoofed `setattr` transparent to the client, nfswall drops the outgoing reply message for the spoofed `setattr`. The GID update appears to the client to have been a server modification to the attributes which it will retrieve on the access of on the file.

4.5 Tint Policy Enforcement

In general, Neon enforces tint policies at the network layer, either in Domain 0 on the client host, or in a physically separate network firewall element.

In either case, we utilize iptables firewall rules to enforce confinement policies. We use the u32 iptables module to match arbitrary ToS values (including use of nonstandard fields and combinations). We can then drop, reroute, or log the tinted packets as specified by policy. Since all outgoing packets are processed and sent by the Domain 0 kernel, we can disallow tinted data from leaving the machine at all as well as block it at the firewall. The firewall is an iptables router which uses the same ToS matching mechanism to enforce policy at the network perimeter.

Neon does automatic I/O encryption/decryption in Domain 0. There are two modes in which we can encrypt: Neon peer communications and NFS traffic. For all TCP and UDP traffic to other Neon systems, we encrypt the payload of all packets which contain tinted data using a stream cipher (*e.g.*, RC4, Xor substitution) with a fixed shared key. In future work, we envision the use of more powerful key management and cryptographic primitives.

For all traffic destined to and received from the NFS server on port 2049, we automatically encrypt/decrypt the payload of the write requests and read replies. Because we do not store information on which byte extents of a file are actually tinted, we must assume that the tinted data is already encrypted. Neon will further ensure that the data stays encrypted, either from edits to the original or copying (or otherwise deriving) the file. We also use only the Xor substitution cipher to avoid problems with stream alignment on subsequent writes to the same region.

5 Tint applications

We evaluate our proof-of-concept Neon implementation by measuring both the overhead of various operations as well as its performance in a variety of real-world scenarios. Our test setup consists of an NFS server, Neon hosts, and a firewall. For single-VM tests, we used a local RAM disk. For tests involving a remote storage server, we ran a tint-enabled NFS Server (kernel 2.6.17) on VMware Server 1.02 with 512 megabytes of RAM. Neon hosts are guest virtual machines in a development release of Xen 3.0 with Qemu 0.7.2. Each guest has 256 megabytes of RAM and runs a Linux 2.6.12-xenU kernel. The firewall uses iptables to enforce confinement policy and runs a 2.6.12-xen0 kernel. All the test systems and Xen domains are hosted on Dell PowerEdge SC1450s with two 2.8-GHz Pentium 4 Xeon processors and 2 GB of RAM.

5.1 Overheads

We performed a series of micro-benchmarks to measure Qemu overhead, the effect of tinting and untinting operations on network throughput, and the timing overhead of the tint tracking table.

5.1.1 Emulation

To begin, we measured the overhead of transitioning between Xen and Qemu on our hardware. On average, it takes 51.3K cycles to transition from virtualized to emulated execution (V2E), and 42.9K cycles to transition from emulated to virtualized execution (E2V). These results are similar to previously published performance measurements of Qemu in Xen [15], and show that our implementation does not introduce a significant transition penalty to the existing Xen-Qemu implementation.

5.1.2 Network processing

In our prototype implementation of Neon, Qemu inspects both incoming and outgoing packets to check for tint. Because Qemu executes as a user process, there is significant context switching overhead before a packet completely traverses the dom0-qemu-domU path. Without Qemu's networking thread present, pinging a machine on the local Ethernet segment from a guest VM has an average round trip time of 0.19 ms in our configuration. With the networking thread running, the average round trip time increases to 5.6 ms. Pinging a host a few hops away yields round trip times averaging 4.15 ms without the network inspection thread and 9.54 ms with the networking thread, suggesting a constant additive latency of approximately 5 ms in the current configuration. Because packet processing is dependent on a user process, an increased load in dom0 will increase the latency of the guest.

While sub-optimal, a minor increase in latency is unlikely to impact most applications. More worrisome, however, would be a decrease in network throughput. Using iperf, we measured the throughput of the loop-back Ethernet device; the results are summarized in Table 1. Latency has little impact on throughput when no tint is present. As a control, we transfered 'tinted' data without Qemu, meaning we set the ToS field manually to measure any possible impact due to special ToS handling. Our results indicate a negligible difference in throughput. When the Qemu networking thread is present, however, we see a 4.5% decrease in throughput when handling tinted data.

5.1.3 Tint tracking

We also measured the time required to manipulate entries to the tint tracking table when tinting memory, checking for tint, and untinting memory. Specifically, we repeatedly tinted, checked, and then untinted X bytes of memory at the same address for a large number of iterations.

	No-Qemu	Qemu
Untinted	94.2 Mbps	94.1 Mbps
Tinted	93.3 Mbps	89.1 Mbps

Table 1: Network throughput measurements with and without the tint thread.

	Tint (μs)	Check (μs)	Untint (μs)
Machine word	0.056	0.057	0.043
Ethernet frame	5.84	6.38	1.67

Table 2: Time overhead of the tint table for tinting, checking tint, and untinting a particular region of memory.

We initially set $X = 4$, since a primary method of propagating tint through the CPU is the loading and storing of 4-byte words. We also considered $X = 1514$ as the network device is another critical pathway for importing and exporting tint for the system.

Table 2 shows the results. For tinting 4-bytes at a time, the table operations incur an overhead of 40-60 ns; MTU-sized Ethernet packets incur roughly a 6-μs overhead. These numbers represent worst-case costs, however, because in this configuration, the tint operation will allocate the maximum number of tables for the given memory range, the check operation will traverse the table to the deepest level, and the untint operation will deallocate every sub-table that was allocated during tinting.

5.2 Application performance

While the overhead associated with tint tracking are non-trivial, we expect the vast majority of data to be untinted. In this section, we provide performance measurements of a number of real-world scenarios using tinted data to determine the practical impact of Neon. Each experiment is carried out on the test bed described previously. For each test, we provide a use-case and quantify the overhead due to Neon, both in terms of execution time and memory consumption due to tint tracking.

5.2.1 Derived data

The test illustrates the concept of system-wide tint propagation. Consider the following scenario (Figure 3): A file A contains some amount of data tinted with the tint X. A user retrieves A from the file server onto host I, and copies the contents of file A into file B. Now, when file B is transmitted to other hosts (including the NFS server), its contents are tinted appropriately. Additionally, we can enforce confinement policy on the data at the firewall if file B is ever transmitted across the network. Assuming there are no other hosts that modify the file B, the packets containing file B should have the same tint as file A, namely X.

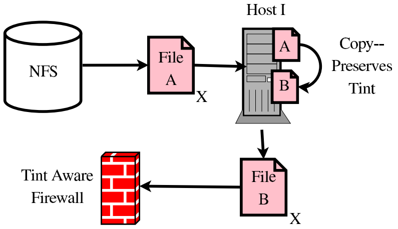


Figure 3: Direct derivation application

We implemented the various stages of this type of experiment and give the performance results in Table 3. The Local Copy test is the time taken to copy a 4-MB file already retrieved from NFS to another file local to the system. Consistent with our expected use case, the file is sparsely tinted—one out of every 64 bytes is tinted. The 'Remote Copy' test takes a previously retrieved file and copies it back to the NFS server. We see that the performance of both operations is slightly degraded by a factor of 2–10. There is considerable variance in the performance, however, due to the fact that Qemu processing is performed at user level as described earlier. In both cases, Neon consumes slightly less than one MB of memory tracking the tinted bytes. Of course, the sparse tinting represents a worst case: much more compact extent-based data structures can be constructed for large contiguous regions of tinted data.

5.2.2 Combining tints

In our next scenario, we consider combining data from N different files each with unique tint values into one file and sending it over the network. Given files A, B and C, with tint values X, Y and Z, we append files B and C to file A and send the resulting file over the network as shown in Figure 4. The tint value observed in the packets containing the modified A are XYZ. This test shows that our system not only tracks tint values, but it performs tint aggregation, and it uniquely identifies multiple sources of tinted data.

We repeated the copy tests for varying number of source files (with unique tints). We concatenated a variable number of files with distinct tints together to form a mixed tint output file. For Remote Copy, we combine *and* copy the file back to NFS. Both the time overhead size of the tracking data structures increase with added tint complexity.

5.2.3 Automatic I/O encryption

Neon can also automatically encrypt tinted data before it gets written to the network, or on disk. In our system we treat the tint enabled NFS server as the the disk. When Neon needs to send a packet with tinted data to another Neon host, it automatically encrypts the TCP or

	Non-Qemu (s)	Qemu (s)	Tinted (s)	Space (MB)
Local Copy	0.010	0.011	0.098	0.86
Local Copy (2 tints)	0.010	—	0.055	0.73
Local Copy (4 tints)	0.010	—	0.081	1.14
Local Copy (8 tints)	0.010	—	0.100	1.62
Remote Copy	1.279	1.136	2.002	0.90
Remote Copy (2 tints)	1.243	—	4.535	0.84
Remote Copy (4 tints)	1.450	—	5.056	1.30
Remote Copy (8 tints)	1.237	—	4.882	1.88
Local Compress	0.087	0.082	8.346	0.85

Table 3: Neon overhead for each application processing a sparsely tinted 4-MB file.

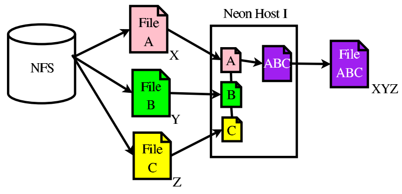


Figure 4: Combining Tints

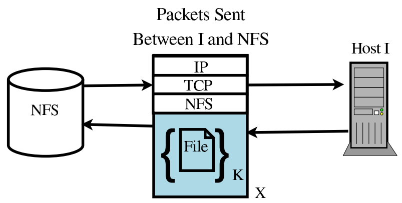


Figure 5: Combining Tints

UDP payload with a shared key. Similarly, when the host issues a write call with tinted data, Neon automatically encrypts the payload of the NFS write leaving the RPC and NFS headers clear text (Shown in Figure 5). Neon automatically decrypts tinted data from peers or the NFS server making the guest OS unaware of the process.

Because of the different in granularity of the NFS server (file-level tint) and Neon (byte-level tint), automatic disk encryption only works when a file is already encrypted. Neon will ensure that any additional data is encrypted or new files derived from encrypted content (written in their entirety) are also encrypted. Hence, we do not include the results in Table 3, but encrypting fully tinted data decreases performance by a factor of 10–15 in our configuration.

5.2.4 Compression

Our system also handles tracking tint in compressed data. Compression presents nontrivial complications to standard data taint tracking approaches because many compression algorithms do not directly reuse any data from the original file. The algorithms build the resulting compressed file using a table of substitution values (*e.g.*, gzip, bzip). Even though the tables introduce a level of indirection in constructing the derived data, Neon is able to handle this scenario by propagating tint values in operations that uses tinted values to index memory.

We performed the 'Local Compress' test by gzipping a tinted file previously retrieved from the NFS server. As shown in the table, the execution time slows by approximate a factor of 10, while requiring less than a megabyte of tracking structures to compress a 4 MB file.

5.2.5 Compilation

As another common example of tint combination, we a compile an executable using tinted headers. Consider the that fileA.h is tinted X and fileB.h is tinted Y. The resulting executable which includes fileA.h and fileB.h will be tinted XY. Additionally object files which only include one header or the other will have tint X or Y. As a motivating example, we compile the freely available gzip utility from source. The results of our complication tests are presented in Table 4. The time we give is to both compile and link gzip. By writing the resulting executable back to NFS, we can see that it retains the proper mixed tint. We also note that intermediary object files from source which includes the tinted headers are also tinted appropriately.

The data structures used to track tint are dynamically allocated and destroyed as bytes are tinted. The space figures quoted in Table 4 are the maximal values during the entire test. Figure 6 shows a representative time-series of the growth of the tint structure as well as the number of tainted bytes in the system over time. We divided this experiment into four phases to investigate impact of the residual tint. In the first and third parts (labeled "Un" in the graph), we compiled untinted gzip source. In the second and fourth parts (labeled "T") in the graph, we compiled gzip with two tinted source files (same as in the "Compile with 2 tinted files" experiment in Table 4). We observe a few trends.

First, tinted compiles results in an increase in both tinted bytes and tint structure overhead. Second, we see the first untinted compile results in no overhead, as ex-

	Non-Qemu (s)	Qemu (s)	Tinted (s)	Space (MB)
Compile (1 tinted file)	13.711	15.009	18.466	0.41
Compile (2 tinted files)	13.700	15.065	86.268	0.81

Table 4: Compilation of gzip with tinted headers.

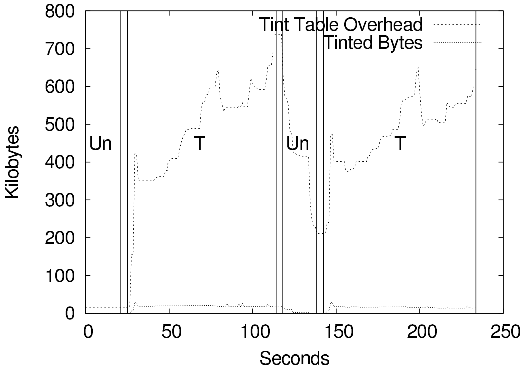


Figure 6: Memory consumed by the tint-tracking data structures while compiling gzip with a tinted header file.

pected — aside from the fixed cost of the top-level tint table. The other interesting trend is that the tint overhead and number of tint bytes decreases *during* the second untinted compile phase. This reflects that data will not be untinted until another application claims and overwrites the same memory.

5.2.6 *Copy & paste*

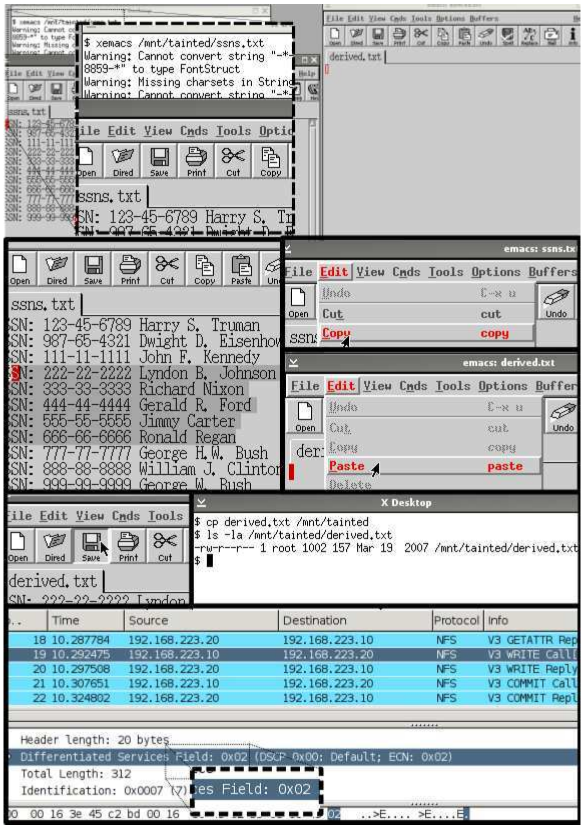
In this test, we exercise the propagation of tint through common application usage. We copy and paste (Figure 7) a portion of a tinted file into an alternative file. To verify that the tint has been preserved, we copy the derived file back to the NFS server. We can see that the tint value is preserved on the NFS server, furthermore the packets containing the tint data are appropriately marked.

The figure illustrates how we executed our exercise. We start with two instances of Xemacs, We open and copy a portion of a tinted file containing social security numbers from the first instance of XEmacs into the second. We proceed by copying the derived file to the NFS server. We affirm that the packets with tinted data are marked correctly by examining them with Wireshark, more over we ensure the tint values are propagated to the NFS server.

5.2.7 *Multi-hop tests*

In the previous tests, we propagated tint only to the NFS server or other local VMs and not to other Neon peers. Here, we verify the ability of Neon to attribute data from multiple hops while mixing tints.

Consider the following general scenario: host I retrieves a file A with tint value X and sends it to host J.



```
$ xemacs /mnt/tainted/ssns.txt
Warning: Cannot convert string "-*-
8859-*" to type FontStruct
Warning: Missing charsets in String
Warning: Cannot convert string "-*-
```

```
ssns. txt
SN: 123-45-6789 Harry S. Truman
SN: 987-65-4321 Dwight D. Eisenhow
SN: 111-11-1111 John F. Kennedy
SN: 222-22-2222 Lyndon B. Johnson
SN: 333-33-3333 Richard Nixon
SN: 444-44-4444 Gerald R. Ford
SN: 555-55-5555 Jimmy Carter
SN: 666-66-6666 Ronald Regan
SN: 777-77-7777 George H.W. Bush
SN: 888-88-8888 William J. Clintor
SN: 999-99-9999 George W. Bush
```

```
derived. txt
SN: 222-22-2222 Lyndon
```

```
$ cp derived.txt /mnt/tainted
$ ls -la /mnt/tainted/derived.txt
-rw-r--r-- 1 root 1002 157 Mar 19  2007 /mnt/tainted/derived.txt
$
```

..	Time	Source	Destination	Protocol	Info
18	10.287784	192.168.223.20	192.168.223.10	NFS	V3 GETATTR Rep
19	10.292475	192.168.223.10	192.168.223.20	NFS	V3 WRITE Call
20	10.297508	192.168.223.20	192.168.223.10	NFS	V3 WRITE Reply
21	10.307651	192.168.223.10	192.168.223.20	NFS	V3 COMMIT Call
22	10.324802	192.168.223.20	192.168.223.10	NFS	V3 COMMIT Repl

```
Header length: 20 bytes
Differentiated Services Field: 0x02 (DSCP 0x00: Default; ECN: 0x02)
Total Length: 312
Identification: 0x0007 (7)
es Field: 0x02
00  00 16 3e 45 c2 bd 00 16                02   ..>E.... >E....E.
```

Figure 7: Copy & paste between two XEmacs sessions.

Host J already has some tinted data with tint X. Host J then combines file A with another file B with tint Y. This date when propagated to a peer or to NFS will be tinted XY. This test demonstrates that tint values are propagated on along multiple hops through multiple machines within a network.

We performed an example of the above test using `scp` to demonstrate multi-hop encrypted tint tracking. We retrieved a 4-MB file with a tint value of 0x2 and used `scp` to copy A to another Neon host. We examine the tint value of A on the destination host by copying it back to the NFS server. The tint value of X is properly propagated on back to the gid of the remote file.

A more advanced example of this type of scenario is for port forwarding. We ran the following test: We used `ssh` to port forward the same scp test described above through a third machine running Neon. Though the extra host is just an intermediary and does not store the data, the source tint is still propagated to the final destination.

6 Conclusions

Mandating restrictions over information use is a fool's errand unless one has a mechanism for enforcing their use both on individual hosts and between them. Today few such mechanisms exist that are both effective and compatible with existing operating systems and applications. In this paper we have motivated the need for transparent system-wide information flow tracking for enforcing policies on derived data. We have demonstrated the viability of this approach using in a prototype system, Neon, that propagates and combines orthogonal per-byte policy labels. Finally, we have used these mechanisms to implement a variety of information management policies including mandatory encryption of sensitive data, network-based confinement and tracking copyright license compliance.

References

[1] F. Bellard. QEMU: a fast and protable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, Apr. 2005.

[2] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813 (Informational), June 1995.

[3] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications*, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society.

[4] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proc. 13th USENIX Security Symposium*, August 2004.

[5] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 133–147, New York, NY, USA, 2005. ACM Press.

[6] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[7] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing Hardware Architectures for Security. In *Workshop on Duplicating, Deconstructing and Debugging*, Boston, MA, June 2006.

[8] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[9] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[10] A. Dolya. Global data leakage survey 2006. http://www.infowatch.com/, Feb. 2007.

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

[12] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[13] F. S. Foundation. Negotiating compliance. http://www.fsf.org/licensing/dealt.html, 2007.

[14] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. CSI/FBI Computer Crime and Security Survey. http://i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2006.pdf, 2006.

[15] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proceedings of the ACm Eurosys Conference*, 2006.

[16] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[17] C. Laird. Taking a hard-line approach to encryption. *IEEE Computer*, 40(3), 2007.

[18] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[19] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 129–142, New York, NY, USA, 1997. ACM Press.

[20] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[21] U. G. D. of Veterans Affairs. Latest information on veterans affairs data security. http://www.usa.gov/veteransinfo/, Mar. 2007.

[22] S. O'Hanlon. Spy laptop safety no longer mission impossible. Reuters, Aug. 2001.

[23] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fringerprinting zero-day attacks. In *Proceedings of the ACm Eurosys Conference*, 2006.

[24] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.

[25] D. Reilly. Hotels.com credit-card data lost in stolen laptop computer. Wall Street Journal, June 2006.

[26] U. S. Seargant at Arms. Testimony of the computer report. REPORT ON THE INVESTIGATION INTO IMPROPER ACCESS TO THE SENATE JUDICIARY

COMMITTEE'S COMPUTER SYSTEM, 2004.

[27] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[28] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[29] H. Welte. Netfilter libnetfilter_queue. http://www.netfilter.org/.

[30] J. Xu and N. Nakka. Defeating memory corruption attacks via pointer taintedness detection. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.

[31] N. Zeldoch, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementatoin*, Seattle, WA, November 2006.