# Studying Naïve Users and the Insider Threat with SimpleFlow

Ryan Johnson
ryan.v.johnson.mil@mail.mil

Jessie Lass
jessie@lassimus.com

W. Michael Petullo
mike@flyn.org

Department of Electrical Engineering and Computer Science
United States Military Academy
West Point, New York USA

## ABSTRACT

Most access control systems prohibit illicit actions at the moment they seem to violate a security policy. While effective, such early action often clouds insight into the intentions behind negligent or willful security policy violations. Furthermore, existing control mechanisms are often very low-level; this hinders understanding because controls must be spread throughout a system. We propose SimpleFlow, a simple, information-flow-based access control system which allows illicit actions to occur up until sensitive information would have left the local network. SimpleFlow marks such illicit traffic before transmission, and this allows network devices to filter such traffic in a number of ways. SimpleFlow can also spoof intended recipients to trick malware into revealing application-layer communication messages even while blocking them. We have written SimpleFlow as a modification to the Linux kernel, and we have released our work as open source.

## 1 Introduction

Most modern access control systems immediately deny all unauthorized activity on a system. This holds for the discretionary filesystem access controls found in Unix and also for Security-Enhanced Linux (SELinux); both stop any action that is unauthorized as soon as a violation occurs. Network firewalls also behave this way as they block suspected malicious traffic from ever interacting with the protected network. Access controls following this pattern reduce how much information we can gather about an attack because the security models require that the attack fail early. With current systems we must decide whether we wish to enforce strong security or maintain the ability to gain further information about an attack; it is difficult to provide both.

Consider a user Mallory who wishes to exfiltrate sensitive data contained in file $f$ from her corporate network. Many contemporary systems provide a choice: either (1) forbid Mallory from reading $f$, which might prevent users such as Mallory from performing benign work; or (2) allow Mallory to read $f$, which might allow Mallory to inappropriately transmit the contents of $f$. A third possibility, an air-gap network, is inconvenient and incurs a high monetary cost. A secondary disadvantage of (1) is that if Mallory were to attempt to access $f$ and fail, then it is likely that Mallory could claim her access was accidental, as there would be little evidence of malicious intent. We are interested in allowing Mallory to read $f$ for legitimate work while collecting clear evidence of Mallory's intentions and preventing Mallory from exposing $f$ outside of the local network. Our goals are the same for accidentally-executed malware which acts under the principals of other users.

In this paper, we introduce SimpleFlow, an information-flow-based access control system which spans an operating system kernel modification and a network filter. We implemented the kernel aspect of SimpleFlow as a Linux Security Module (LSM). SimpleFlow:

- modifies the Linux kernel and thus can constrain all processes on the host it protects,
- demonstrates an information-flow-based access-control system based on the LSM interface,
- delays access controls until a process tries to write confidential data outside of the local network,
- spoofs blocked recipient hosts to reveal application-layer communication messages and thus discover the intentions of malware,
- requires few changes outside of the kernel, and
- incurs a small runtime overhead.

The design of SimpleFlow enables the study of naïve and malicious users as they use the system by allowing them to act freely, yet it denies their ability to exfiltrate confidential data. SimpleFlow composes well with other access control systems, so existing mechanisms can protect information in other ways as required.

In the following sections, we describe related work (§2); summarize our threat model (§3); describe the design of the SimpleFlow LSM and network (§4); and present performance results along with examples of the practical use of SimpleFlow (§5).

## 2 Related work

Saltzer et al. established five basic principles of access control with Multics, and these provide the foundation for security in today's computing systems [28]. The principles are: (1) check permissions on each access, (2) protect based on permission rather than exclusion, (3) secure by design rather than obscurity, (4) grant each process only the least privilege necessary, and (5) provide simple, easy-to-use interfaces. In modern Unix-like

Permanent ID of this document: 1d99a38f432d84dc52936f334acc15d5.
Date: September 15, 2016.
*MIST'16, October 28 2016, Vienna, Austria*
ISBN 978-1-4503-4571-2/16/10.
DOI: http://dx.doi.org/10.1145/2995959.2995960

systems, these principles primarily manifest as memory protections and controls on filesystem access.

**Unix**: The Unix kernel isolates the memory of each process and ensures that processes cannot directly interact with the kernel's memory or most hardware. Processes must make use of kernel-mediated system calls to access files and other system objects. The owner of each system object determines the access controls which apply to the object. A special user, root, is not subject to Unix's discretionary access controls.

It is advisable to avoid running network-facing dæmons as root or to only run dæmons that drop privileges immediately after performing the privileged operations necessary to set up a connection. Unix-like systems grant least privilege by making use of pseudouser accounts. Such accounts only exist to run a particular dæmon, and they only provide access to the resources the dæmon needs to satisfy its purpose.

Bernstein designed the qmail mail-transfer agent with least privilege in mind [6], and it exemplifies how many network services implement the principle of least privilege on modern Unix systems. Qmail runs a core dæmon process which executes separate delivery processes under the user ID of each email's recipient. While the core process must run as root so that it may start processes under other users' IDs, the other components of qmail are subject to the access controls already present in Unix. This minimizes the amount of code in qmail and hence reduces the chance for bugs. Bernstein justifies this design because the core process is very small, highly-audited, and well-controlled. Other programs follow similar designs [26].

Other programs, including many applications built on Apache, implement their own access control system. This removes the need to maintain root privileges, but it has a significant consequence: a vulnerability anywhere in Apache might provide access to anything that Apache controls, whereas a compromised qmail component would likely only provide the attacker with access to a small subset of qmail's resources. Furthermore, applications built on Apache often duplicate the access controls found in Unix and these additional lines of code present more possibilities for bugs.

**Limitation of discretionary access controls**: An important limitation of the Multics model was acknowledged by its authors [28, final ¶ on p. 392]. Namely, users decide the access controls for the system objects they own. This confounds preventing insider attacks within a protected subsystem; administrators cannot restrict data sharing without extending the standard access control list mechanism. The traditional access controls of the Unix filesystem exhibit a similar limitation.

**Linux Security Module and netfilter**: Linux supports alternate security models with its LSM framework [33]. Developers of an LSM can add arbitrary security context to key kernel data structures, and the kernel will invoke their LSM callback functions to decide whether to allow or deny a system call [29]. LSM also provides a `/proc`-based interface with which an administrator can interact with the running LSM.

Similarly, the Linux kernel's NetFilter subsystem [2] allows programmers to write custom network-filtering modules. The NetFilter interface is similar to LSM in that it exists as a set of hooks, here throughout the Linux kernel's network stack. A NetFilter module registers callback functions against these hooks in order to affect the kernel's processing of packets.

**Mandatory access controls**: Mandatory access controls set a security policy which cannot be subverted by normal users. SELinux builds on top of LSM to provide a type-enforcement security model. SELinux enforces a policy to constrain programs more precisely than traditional Unix controls, and this policy is not subject to root or system-object owners. Yet SELinux's sample policy is very large [22], and its complexity leads many users and system administrators to simply disable SELinux entirely [13]. It seems that SELinux is complex because its access control model follows very closely the low-level semantics of Unix's system calls.

**Information-flow-based access controls**: If programmers have difficulty wisely using Unix's system calls and security facilities, then it is likely that policy engineers would likewise have trouble appropriately constraining programs through low-level policy. Instead of reasoning about low-level operations and objects, systems such as IX [18] and HiStar [35] aim to provide a simpler security model based on information flow. IX, an early implementation of multi-level security for Unix, adjusts the labels on outputs based on system calls which involve information flow. IX is more dynamic than the prescription of the so-called Orange Book [9], but it still denies access to high levels of information before an unauthorized user might reveal his intentions. We note the portions of SimpleFlow which follow IX throughout this paper.

Other researchers have applied similar approaches elsewhere in the software stack. Some programming languages—such as Perl [3], Jif [21], and Jif's subsequent work—provide information-flow-based protections. Similar protections also exist within applications; one example is the cross-site-scripting defense in the Chromium browser [30].

**Provenance systems**: Systems which track the origin and evolution of the data they process are said to exhibit whole-system provenance. (Here there is often overlap between monitoring and access control.) Recent work in this field includes Panorama [34], Hi-Fi [24], and Linux Provenance Modules (LPM) [4].

Panorama discovers information flow by running a system within a processor emulator and considering the information-flow effect of each processor instruction. This means that Panorama can work with closed-source operating systems which cannot be modified, but Panorama decreases runtime performance by a factor of 20; thus Panorama targets off-line analysis.

Both Hi-Fi and LPM modify the Linux kernel. Like SimpleFlow, Hi-Fi makes use of LSM. LPM provides a LSM-like interface which allows writing provenance modules. The initial LPM release provided two reference implementations: Provmon, which extends Hi-Fi with file versioning and improved network context, and SPADE. LPM provides a data-loss prevention tool which makes use of special file-transfer utility to permit or deny transfers based on LPM's information-flow analysis.

SimpleFlow resembles Hi-Fi and LPM. Most notably, SimpleFlow shares with Hi-Fi the use of LSM; we note other similarities throughout this paper. Yet SimpleFlow favors simplicity by dictating a security model and discerns more information through the use of network spoofing. Because SimpleFlow combines authorization and provenance over confidential files, it can implement a fixed label like IX [18, §2.4]; we describe this in §4.1.5. SimpleFlow also mediates all IP traffic instead of relying on a specialized file-transfer utility like LPM; SimpleFlow could make use of a specialized utility to sanitize confidential messages, but public network traffic flows unimpeded. SimpleFlow also avoids LPM's provenance-graph-size bottleneck [4, §5.3]. Table 1 summarizes the differences between SimpleFlow and the access-control and provenance systems described above.

**The evil bit and security labeling**: Bellovin describes the placement and use of a security flag within IPv4 headers in RFC 3514, released on April 1st (April Fool's Day), 2003. The

| System | Kernel agnostic | Low overhead | Provenance* | Access controls | Centralizes granting of permissions | Delays access controls until net. write | Labels IP traffic without transfer utility | Block connections yet reveal app.-layer messages |
|---|---|---|---|---|---|---|---|---|
| Multics | | ✓ | | ✓ | | | | |
| Traditional UNIX | | ✓ | | ✓ | | | | |
| SELinux | | ✓ | | ✓ | ✓ | | | |
| IX | | ✓ | | ✓ | ✓ | | | |
| Panorama | ✓ | | ✓ | | | | | |
| Hi-Fi | | ✓ | ✓ | | | | | |
| LPM | | ✓ | ✓ | | | | | |
| SIMPLEFLOW | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*The provenance systems could recreate some of SIMPLEFLOW's features using external tools, but they might have difficulty scaling as the provenance graph grows.

**Table 1:** Comparison of provenance and access-control systems

RFC declares the previously-unused high-order bit of the IP fragment offset field—originally marked as reserved in RFC 791 [25]—as the evil bit [5]. The RFC requires that malicious users set this bit to indicate that firewalls should drop their packets.

Due to its satirical nature, there is little practical use of RFC 3514. Matthew Dodd submitted a patch to FreeBSD that implemented support for the evil bit on the day the RFC was released [10], but the FreeBSD team removed the patch the next day [11]. The patch caused FreeBSD to discard all transmitted and received evil packets. Ben Cox implemented a small patch to set the evil bit on all outgoing traffic in the Linux kernel. In doing so, he was able to find a small list of domains which refused evil connections. He also implemented some basic rules in `iptables` to allow only connections which satisfied various evil-bit and port requirements [8].

RFC 1108 [15] and RFC 5570 [14] describe a scheme for labeling the security classification of IPv4 and IPv6 packets, respectively. This is a more serious feature, yet it resembles the simpler-if-less-expressive evil bit. Commercial products exist which have the ability to filter on these fields [7, see IP Security Options].

## 3 Threat model

SIMPLEFLOW monitors and restricts non-root users as they go about their work in a SIMPLEFLOW-protected subnet. SIMPLEFLOW users might be naïve insiders—who are at risk of unintentionally performing some action to reveal sensitive data—or malicious insiders—trusted users who intend to violate the confidentiality of sensitive data. Users can operate a SIMPLEFLOW host—either graphically, through the network, or through a local shell account—to execute programs, access sensitive or confidential data subject to traditional UNIX access controls, and send non-sensitive data over the network. The goal of SIMPLEFLOW is to (1) forbid the exfiltration of data an administrator has marked as confidential from the SIMPLEFLOW-protected subnet and (2) allow the attacker freedom of action so that administrators can safely assess the attacker's intentions and techniques.

SIMPLEFLOW cannot itself prevent a local user who observes and remembers sensitive data from manually entering it into a host outside of the SIMPLEFLOW subnet. This type of determined insider can be thwarted by controlling the use of pen-and-paper notebooks and other non-technical measures. However, a user who logs into a SIMPLEFLOW host from outside the SIMPLEFLOW subnet will find himself unable to display confidential data over his network connection. This holds even while his other programs—those that avoid transmitting the data—are able to locally process the data. SIMPLEFLOW's protections do not extend to removable media, but we describe two strategies for addressing removable media without further modifications to SIMPLEFLOW itself in §4.1.5. In any case, SIMPLEFLOW provides a clear record of all confidential data access.

We do not claim to protect against malicious-but-trusted administrators because they can: (1) detect the presence of SIMPLEFLOW and thus might avoid malicious actions, (2) untaint a process using SIMPLEFLOW's `/proc` interface, (3) manipulate the confidentiality labels on system objects, and (4) mark programs as trusted and thus not subject to SIMPLEFLOW. While UNIX-based systems are sometimes susceptible to attacks which escalate a normal user to root, administrators can reduce the likelihood and hazard of these threats by applying the least-privilege principle. This includes installing with care setuid-bit programs and dæmons which run with special privileges.

SIMPLEFLOW is dependent on the integrity of the Linux kernel and its ability to isolate processes and mediate system calls using the running LSM. SIMPLEFLOW leaves the protection of the kernel, a hypervisor, and hardware to techniques such as software verification and measured and verified booting.

SIMPLEFLOW's design allows for the system administrator to designate crucial data that cannot leave the network. An administrator with root access manages the confidentiality of data, and the SIMPLEFLOW kernel enforces the administrator's settings. We do not claim to remove all covert channels with SIMPLEFLOW; however, we do address a number of high-bandwidth channels in §4.1.3.

## 4 Design and implementation

SIMPLEFLOW aims to protect confidential information on a system regardless of the design of applications attempting to access this information. We accomplish this by tracking information flow as it propagates through the system as a result of system calls, albeit in a manner much less sophisticated than that of IX or HiStar. As we are interested in understanding the mistakes or intentions of naïve and malicious users, we set out to allow dangerous behavior to continue up until it tries to communicate outside of an isolated subnet.

Under SIMPLEFLOW, the system administrator designates some filesystem objects as *confidential* and some programs as *trusted*. (Similar to SELinux, SIMPLEFLOW stores both using extended attributes in the filesystem.) Any process not loaded from a trusted program will become *tainted* upon reading from a confidential object. The kernel transfers this taint status from process to process as a result of inter-process communication (e.g., an untainted process reads from a tainted process over UNIX socket). If a tainted process writes to the network, then the kernel sets the packet's RFC 3514 evil bit; this allows for a variety of filtering or spoofing strategies which might help determine the intention of the principal who read the confidential data in the first place.

SIMPLEFLOW considers as confidential filesystem objects which bear a `security.simple-flow.confidential` extended attribute set to `true`. A privileged administrator can use standard tools such as `getfattr` and `setfattr` to view or set this value. An administrator can also mark an executable file as trusted by setting its `security.simple-flow.trusted` extended attribute to `true`. Hi-Fi [24, §5.5] and LPM [4, 3.4.4] similarly use extended attributes.

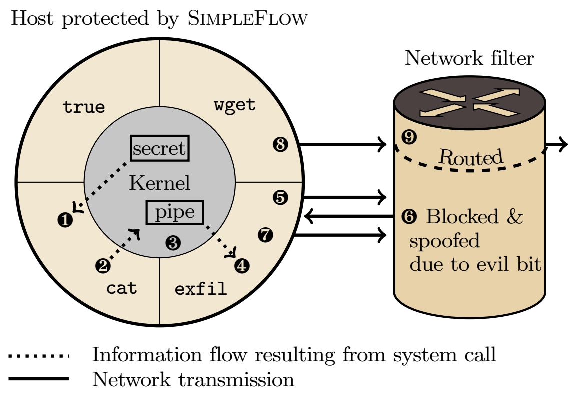
Host protected by SIMPLEFLOW
Network filter
true
wget
secret
Kernel
pipe
cat
exfil
❶
❷
❸
❹
❺
❼
❽
❾
Routed
❻ Blocked & spoofed due to evil bit
Information flow resulting from system call
Network transmission

**Figure 1:** SIMPLEFLOW in practice: mediating `cat secret | exfil`; on the left is a host running the SIMPLEFLOW kernel, and on the right is the SIMPLEFLOW network filter

Figure 1 depicts a practical use of SIMPLEFLOW. Here a user ran the command `cat secret | exfil` in an attempt to exfiltrate the contents of `secret`. Let us assume `exfil` tries to transmit the secret using an HTTP POST request.

❶ `Cat` invokes the `open` system call to open the file `secret` for reading. Since this file is confidential, the kernel taints the process running `cat` when it reads from the file.

❷ The `cat` process forks a child, executes `exfil`, and writes the contents of the secret file over a pipe to `exfil`.

❸ SIMPLEFLOW marks the pipe as confidential because `cat` wrote to it while tainted.

❹ `Exfil` reads the contents of the secret file from the pipe and becomes tainted.

❺ `Exfil` attempts to send the contents of the secret file within an IP packet toward the attacker's command-and-control server. Since `exfil` is tainted, SIMPLEFLOW sets the evil bit on any such IP packet `exfil` produces.

The second aspect of SIMPLEFLOW is its network filter. The purposes of the filter are (1) to ensure that packets which contain sensitive data do not leave the SIMPLEFLOW-protected subnet and (2) to extract information about the nature of the attacker's exfiltration attempt. Recall that the attacker might be using HTTP or another protocol on top of TCP as an exfiltration channel. The filter will block evil DNS request datagrams and TCP SYN segments, and so we are at risk of `exfil` giving up before sending the application-layer request that should follow the initial connection setup. This missing information would be valuable to the defenders, as it reveals the attacker's attempted exfiltration channel. Returning to Figure 1:

❻ The network filter spoofs the DNS server and intended recipient of an evil-bit exfiltration packet, thus responding to DNS requests and completing the three-way handshake.

❼ `Exfil` begins sending its application-layer request. This message is blocked but recorded by the network filter.

SIMPLEFLOW does not impede normal network traffic:

❽ An unrelated process running `wget` is not tainted, and it sends an IP packet.

❾ The network filter routes this benign packet to the Internet.

Thus SIMPLEFLOW forbids the transmission of many exfiltration packets, but leaves other packets unimpeded.

## 4.1 SimpleFlow kernel module

We implemented SIMPLEFLOW's access controls as an LSM. As with previous work, this ensures that user applications cannot bypass SIMPLEFLOW's controls.

### *4.1.1 Security context*

SIMPLEFLOW adds a number of fields to the kernel data structures capable of bearing arbitrary security data. First, SIMPLEFLOW adds a Boolean `owner_tainted` field to the inode, socket, message queue, and shared-memory structures to represent whether the communication channel is tainted and hence whether the kernel should taint untrusted processes which read from the channel. SIMPLEFLOW also adds `tainted` and `trust_mask` fields to the task structure to manage the state of processes. Finally, SIMPLEFLOW adds to the socket structure a pointer back to the task which created the socket.

### *4.1.2 Mediation points*

SIMPLEFLOW registers a number of LSM hooks to track information flow across each of the system calls which permit I/O. Here we describe some of the unique aspects of each such mediation point.

**File and pipe I/O**: SIMPLEFLOW mediates file and pipe I/O using the `file_open` and `file_permission` hooks. The `file_open` implementation logs the interaction with confidential files for diagnostic purposes. The `file_permission` hook serves two purposes: (1) on reads, the function checks whether the inode in question has its `owner_tainted` field set and whether the reader is not trusted; if both are true, the kernel taints the reader. (2) on writes, the function checks to see if the writer is tainted; if so, the kernel sets the inode's `owner_tainted` field to true. In the case of a write, the kernel also tries to set the target object's `security.simple-flow.confidential` extended attribute.

Pipes provide an additional requirement. It is possible that a reader blocks while waiting for data, and while the reader is blocked, a writer taints the pipe with a write. Since the existing Linux kernel invokes `file_permission` only before blocking, we added a new LSM call to the kernel in `pipe_read`.

**Sockets**: SIMPLEFLOW mediates sockets with the `socket_post_create`, `socket_sock_rcv_skb`, `file_permission`, `socket_recvmsg`, and `socket_sendmsg` hooks along with a netfilter module which provides `NF_INET_LOCAL_OUT` functions for IPv4 and IPv6. This affects UNIX-domain, IPv4, and IPv6 sockets.

SIMPLEFLOW's `socket_post_create` implementation sets the socket's `task` and `owner_tainted` fields, based on the process which creates the socket. The `task` field is important because the process which is in execution when the kernel later transmits the packet might not be the process associated with the packet.

For network sockets, incoming data causes the kernel to invoke `socket_sock_rcv_skb` before `file_permission`. This allows the kernel to inspect the incoming packet's evil-bit field. If it is set, the kernel sets the receiving socket's `owner_tainted` field to true. Thus SIMPLEFLOW's taint status will pass from a process on one SIMPLEFLOW host to a process on another across a network. (This is not true if both hosts are not running SIMPLEFLOW, but we assume non-SIMPLEFLOW hosts are beyond the network filter.)

`File_permission` behaves as it did while mediating file I/O. Here it sets or gets the socket's `owner_tainted` field instead of the inode's. The exception is with a UNIX socket, where the inode still bears the taint status; this serves as an analog to the use of `socket_sock_rcv_skb` with IP sockets.

SimpleFlow invokes `file_permission` in response to having its `socket_recvmsg` or `socket_sendmsg` implementations called. This ensures the mediation of UDP-style I/O.

Finally, SimpleFlow uses a netfilter module to set the evil bits of tainted packets. For both IPv4 and IPv6, SimpleFlow registers a function that inspects the subject socket's `owner_tainted` field and possibly sets the packet's evil bit. Hi-Fi [24, §5.6] and LPM [4, §3.4.2] similarly use a netfilter module; each of these set a unique identifier on TCP connections or UDP datagrams. SimpleFlow needs only a single bit because its analysis precedes packet transmission.

**Pseudo terminals**: Linux provides pseudo-terminal devices to aid programmers who wish to write software-based terminals such as remote shells and X11 terminals. A process which opens `/dev/ptmx` will receive a file descriptor which the process can use to establish a peer device, `/dev/pts/`*n*. A child process can then read and write `/dev/pts/`*n* to communicate with its parent reading and writing on `/dev/ptmx` as if it were communicating with a hardware terminal. The parent can then serve as a bridge between its child and X11 or a network server.

Code paths in `simple_flow_file_permission` handle pseudo terminals along with other files. In the case of terminals, a tainted process that writes to a pseudo terminal causes SimpleFlow to mark the device's peer pseudo terminal.

**System V message queues**: SimpleFlow mediates System V message queues by implementing the `msg_queue_msgsnd` and `msg_queue_msgrcv` hooks. These functions set the message queue structure's `owner_tainted` field or the calling process's `tainted` field under the appropriate conditions.

**Shared memory and `mmap`**: Shared memory is troublesome because the kernel does not mediate access after it initially attaches a process. Instead, processes can directly manipulate shared memory as they can any other memory in their address space. To maintain control, SimpleFlow implements `shm_shmat` and `mmap_file`, and it also performs other checks during the lifetime of a shared memory segment. We describe this in detail in §4.1.5.

### *4.1.3 Mediation with covert-channel considerations*

SimpleFlow removes or monitors a number of covert channels. We leave a full covert-channel analysis of Linux and SimpleFlow to future work, but discuss the high-bandwidth channels SimpleFlow considers here. We refer the reader to the Orange Book [9] for an introduction to covert channels and the tradeoffs involved in covert-channel mitigation.

**Seek pointers**: A Linux process inherits file offset pointers for existing file descriptors from its parent, and thus the `lseek` system call can be abused to create a covert channel [18, §7.2]. We added a new LSM hook, `security_file_lseek` to the Linux kernel. If a tainted process seeks on a file which is also opened by another process, then our implementation of `security_file_lseek` marks the file as confidential. Our `security_file_lseek` similarly taints a process which seeks on a confidential file. Modern Unix conflates `seek` and `tell`. IX returned to these separate calls for more precise control, but we found this too intrusive because it would require modifying applications.

**File names**: Filesystem names provide a channel for communication which is not subject to checks on file writes—a process might try to write confidential data to the name of a file. Thus SimpleFlow implements `inode_create`, `inode_mkdir`, `inode_mknod`, `inode_link`, `inode_symlink`, and `inode_rename` to mark as confidential any directory whose names are modified by a tainted process. SimpleFlow taints processes which call `getdents` (i.e., with `ls`) on confidential directories, as `getdents` invokes the same LSM hook as `read`.

Another channel involves a process $p_1$ creating a series of ordered filenames such as $1$–$n$, reading confidential data (and becoming tainted), and then removing the files whose names correspond with the zeros in the message's binary form. Process $p_2$ could then check the existence of each file to read the message. Since this involves $p_2$ using `open` instead of `getdents`, SimpleFlow implements `inode_permission` to taint a process which reads or executes a marked directory inode. The kernel invokes this LSM hook repeatedly as it walks a path before servicing an `open`, so this prevents $p_2$ from reading from this channel without becoming tainted.

**Getting standard file attributes**: A Unix process can modulate information as a series of variably-sized files or file modification times; another process could demodulate this information by using `stat` instead of `read`. Daniel Ford demonstrated a practical use of this covert channel during the experiments we will describe in §5. For this reason, SimpleFlow's implementation of `inode_getattr` now taints a process which calls a `stat`-like system call on a confidential file.

**Getting and setting extended file attributes**: SimpleFlow's `inode_getxattr` and `inode_setxattr` implementations forbid processes from passing confidential data through extended attributes. A tainted process which sets an extended attribute on a file causes the kernel to mark the file as confidential. A process which reads an extended attribute from a confidential file causes the kernel to taint the process.

SimpleFlow also entirely forbids unprivileged users from getting or setting the SimpleFlow-related extended attributes such as `security.simple-flow.confidential`. For privileged users who set these fields, `inode_setxattr` resets the `owner_tainted` field in the inode structure associated with the target object.

**Program execution**: Parent processes share a number of data with their children in the form of the user area [18, §7.2]. When a process forks a child, SimpleFlow ensures the taint status of the child matches the parent.

SimpleFlow also implements `bprm_set_creds` to check whether the program being loaded by an `exec` is trusted or marked as confidential. If it is trusted, then the kernel sets the process's `trusted` field to true, and if necessary the kernel untaints the process. If the program is confidential, then the kernel taints the process.

**Proc filesystem**: A program can overwrite the values pointed to by its argument pointers (i.e., second argument to `main` in C), and thus affect `/proc/<pid>/cmdline`. A tainted process could modify these values, and another process might read them. We addressed this by implementing `d_instantiate` to check the process associated with a `/proc/<pid>` path and mark the path confidential if the process is tainted.

**Process exit codes**: Process exit codes provide a communication channel at a rate of one byte per `fork`, `exit`, and `wait`. We found tainting all processes which wait on other tainted processes prohibitive in practice as it resulted in too many tainted processes; the process tree would taint back towards the `init` process. Yet we observed the use of this channel during the practical experiments we will describe in §5. Thus SimpleFlow now taints a process which `wait`s on a tainted child only if the parent's program does not bear a `security.simple-`

`flow.trusted` label set to `wait` or `true`. A program whose label is set to `wait` avoids only being tainted due to a `wait`; the label `true` prevents tainting entirely, as described earlier.

An administrator might prevent user programs from tainting system programs by labeling certain user-transitioning programs such as X11's display manager, `login`, or `sshd`. Or, he might label all root-owned programs except for interpreters. Most system programs would not demodulate the contents of a file from exit codes without a control-flow compromise.

**`EADDRINUSE`-type channels**: One process can signal another by either exclusively reserving or not reserving a system object. One proposal we encountered during our testing follows: A tainted process $p_1$ could `bind` to a transport-layer network port in a 255-port range, and another process $p_2$ could try to `bind` to each port in the same range. Process $p_2$ will receive an `EADDRINUSE` error as a result of binding to some port in this range, and that port value is the byte communicated from $p_1$ to $p_2$.

SimpleFlow aims to reduce the bandwidth of this type of channel. In the case of the `bind` channel, SimpleFlow implements `socket_bind` to taint a process which attempts to `bind` to a port already held by a tainted process. This reduces a success/fail channel to a success/taint channel, which is much less convenient to the attacker.

### *4.1.4 Administrative interfaces*

**The SimpleFlow `/proc` interface**: SimpleFlow uses `getprocattr` and `setprocattr` to allow privileged administrators to get and set the taint values of individual processes. Reading from `/proc/<pid>/attr/current` will provide either a `"1\n"` or `"0\n"`, depending if the process in question is tainted or not. (The command `ps auxZ` presents this information in a nice format.) Writing the same values will cause the kernel to taint or untaint the process.

We wanted our `/proc` interface functions to return `-EPERM` to unprivileged users who try to get or set the values it exports. However, we found that some existing utilities were not programmed to properly handle `-EPERM` [27]. Thus SimpleFlow presently returns `-ENODATA` in the case of inadequate privileges.

**The `syslog` interface**: SimpleFlow implements the `syslog` hook to ensure only root can access the kernel's message ring buffer using `dmesg`. This prevents unprivileged users from reading SimpleFlow's log messages. SimpleFlow's implementation of `syslog` returns `-EPERM` when invoked by an unprivileged process to read the contents or size of the ring buffer.

Privileged users can use `dmesg` to access all of SimpleFlow's messages. SimpleFlow logs when trusted programs execute, when untrusted processes read a confidential file, the propagation of the taint status, the presence of evil bits in packets, and a number of other things which are useful to track the activities of users on the system.

**The `ptrace` interface**: Oliver DiNallo demonstrated a practical use of `ptrace` to extract confidential data during the experiments we will describe in §5. SimpleFlow now implements `ptrace_access_check` to ensure that a tainted process does not leak confidential information through Linux's debugging interface.

### *4.1.5 Tradeoffs and considerations*

**Trusted programs**: SimpleFlow's concept of trusted programs provides an administrator with a way to exempt certain programs from mediation. SimpleFlow logs the activities of such programs, but it does not taint them. This allows some programs to transfer confidential data over the network, and the notion of trust also serves as a stop-gap for troublesome programs such as X11. Clearly, trusted programs present a hazard, so administrators should designate them with care.

**Persistence of taint status**: Another consideration is whether to preserve on disk a confidential label which propagated to a file through process interactions. If a tainted process creates or writes to a non-confidential file, then should the operating system mark that file as confidential in a persistent way?

Consider the case where a malicious actor compromises Firefox so he can read and attempt to exfiltrate a confidential file. Further imagine that Firefox, during subsequent benign operations, modifies one of its user settings. If the kernel marked the settings file as confidential when the tainted Firefox process wrote to it, then Firefox could not communicate over the network until a privileged administrator cleared the confidentiality of the configuration file. This is because each time Firefox restarted, it would reread its configuration file and become tainted again. Files like `.bash_history` pose a similar challenge. This problem is well known: LPM names the problem *overtainting* [4, §3.4.5], Jif addresses it with declassification [21], and it resembles the high-water mark described by Weissman [32].

We follow IX and support *fixed labeling* [18, §2.4]. Administrators can set files such as `.bash_history` as fixed by setting the extended attribute `security.simple-flow.confidential` to `never`. A tainted process cannot write to an object bearing this label; such a write would fail with the `-EPERM` error code.

We recognize that this decision might make applications occasionally fail. However, never persistently marking any files would provide many opportunities for an attacker to bypass SimpleFlow's security, and always marking them would cause the problems described above. We leave deciding which files to set as fixed to the system administrator. Privileged administrators can of course avoid the use of fixed labeling and instead declassify persistent objects using `setfattr`.

**Kernel threads vs. user-space processes**: SimpleFlow does not assign a security context to kernel threads, nor do we taint kernel threads. We identify kernel threads as tasks whose `mm` field is set to `NULL`. This is consistent with our threat model which assumes attackers cannot corrupt the operating system kernel.

**Shared memory and `mmap`**: In order to effectively mediate shared memory, SimpleFlow accepts a minor performance impact. The SimpleFlow kernel maintains a graph whose nodes represent processes and whose undirected edges represent connections between processes through shared memory segments. A process might attach to many shared-memory segments and a memory segment might be shared among many processes. Starting from process $p_n$, any other process that is reachable by traversing this graph has the ability to communicate with $p_n$ without invoking a system call.

SimpleFlow's implementation of `shm_shmat` checks to see if the shared memory segment being attached is tainted; if it is, then the kernel taints the calling process. `Shm_shmat` also checks if the calling process $p_m$ is tainted. If it is, then the kernel traverses the graph to find any process directly or transitively connected to $p_m$, and the kernel taints these processes too. The kernel also performs this search each time a process is tainted. The kernel does not taint any trusted processes that it finds in these searches.

The kernel also implements `mmap_file` to mediate the `mmap` system call. If an untrusted process $p_1$ maps a file for reading, then the kernel will taint $p_1$ if the file bears a taint. If an untrusted and tainted process $p_2$ maps without `MAP_PRIVATE` a file $f$ for writing, then the kernel will mark $f$ as confidential. If the process specified `MAP_SHARED`, then the kernel will also search for other processes which have mapped $f$, and it will taint those processes. Furthermore, each time a process taints, the kernel finds each file the process has mapped using `PROT_WRITE` and `MAP_SHARED` and transitively taints the other processes which have mapped the same file.

Aside from these checks, shared memory accesses continue unimpeded by SimpleFlow. In most cases, the only overhead takes place when the kernel attaches a process to a shared-memory segment. Shared-memory already incurs a setup overhead, but this is amortized over the much faster memory accesses which follow. The overhead due to tainting should be even more rare.

**Combining SELinux and SimpleFlow**: An advantage of SimpleFlow's simple model is that it can be combined with other security models while remaining manageable. We created a research prototype which enforced both the SELinux and SimpleFlow models on a Linux system. We did this by modifying the SELinux LSM function implementations to in turn invoke SimpleFlow's implementations and by adding to SELinux's security structures the fields required by SimpleFlow.

We unified the `/proc` interface with a few tricks. SimpleFlow's `setprocattr` adds a `";1"` or `";0"` to the end of the string produced by SELinux's `setprocattr` to represent a tainted and untainted process, respectively. The SimpleFlow implementation of `getprocattr` strips these suffixes before providing the result to SELinux's `getprocattr` *unless* the calling process is `ps`. This way the human reading the output from `ps` can view taint statuses, but existing utilities are not exposed to the new suffixes. We hope to re-implement this using stackable LSMs after porting SimpleFlow to a newer Linux kernel.

**X11**: Practical use of SimpleFlow requires trusting X11; otherwise, tainting any X11 client would result in SimpleFlow tainting every X11 client. The monolithic design of X11 makes it difficult for an operating system to mediate access to the windowing system. For example, if the operating system permits an application to connect to the X11 server, then the operating system is left unable to prevent the application from using copy-and-paste to communicate with other X11 applications or from performing a screen capture. This problem is well known in the literature surrounding X11. Kilpatrick et al. [16] proposed adding calls in X11 out to SELinux to unify X11 and SELinux security policies. Most recently, Garrot used X11 to bypass the isolation provided by Ubuntu snap [12].

We propose adding checks to the X11 server which could propagate the taint status of applications that use X11 to communicate with other applications. Another possible solution would be to follow the work of Wang et al. [31] to sanitize the X11 protocol. Other programs, such as dbus, share similar issues, and researchers have proposed solutions for them too [20, §10.2]. We have left these considerations to future work.

**Removable media**: Neither the SimpleFlow kernel nor SimpleFlow's network filter claim to address removable media. Such media is troublesome because once it bears confidential information it can be separated from the administrative controls of the host computer and network. While introducing a new device to the Unix filesystem using the `mount` system call is a privileged operation, mechanisms exist to allow user-mounted filesystems. For example, the `mount` command often bears the setuid bit and allows users to mount filesystems marked with the user option in `/etc/fstab`. Utilities such as `polkit` and `udisks2` also allow users to mount disks, subject to system policies. We propose addressing removable media by either (1) dynamically marking the folders the media bears as `never` or (2) requiring removable media to be encrypted in such a way that only a controlled computer can decrypt it (i.e., the system but not the user has access to the decryption key). Either of these could be implemented without further modification of SimpleFlow itself.

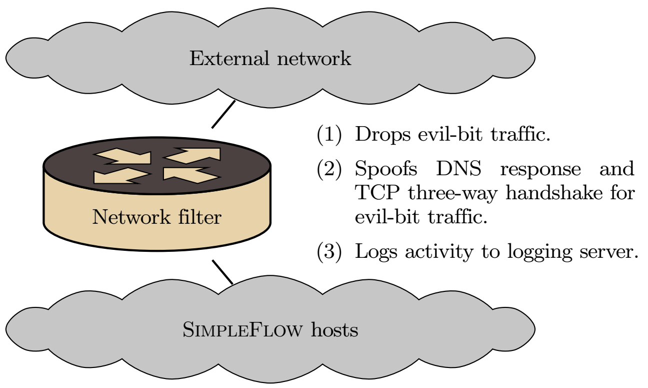
External network
Network filter
(1) Drops evil-bit traffic.
(2) Spoofs DNS response and TCP three-way handshake for evil-bit traffic.
(3) Logs activity to logging server.
SimpleFlow hosts

**Figure 2:** The placement and function of the SimpleFlow network filter

## 4.2 SimpleFlow network

The SimpleFlow LSM exists in the larger context of a computer network. Here we describe how the design of this network compliments SimpleFlow. SimpleFlow works best when paired with a network that (1) routes only non-evil packets out of the SimpleFlow subnet, (2) spoofs the response to any DNS request that is marked with an evil bit, (3) spoofs to complete any TCP connection attempt that is marked with an evil bit, and (4) logs packets that are marked with an evil bit for further analysis. SimpleFlow labels packets in the IP header, thus allowing it mediate any protocol which builds on IP, such as ICMP, TCP and UDP.

A computer running Linux serves as the gateway for our SimpleFlow network. This design ensures that we can monitor, filter, or spoof any packets between our SimpleFlow hosts and the external network. Thus the gateway, which we call SimpleFlow's *network filter*, satisfies the four requirements above.

The network filter has two network interfaces, one on the SimpleFlow subnet and one on the external network. We configure the network filter to forward packets between these interfaces while restricting its own outgoing connections to the SimpleFlow network's central log server. The network filter itself does not accept any inbound connections to reduce its own attack surface.

The network filter works with the SimpleFlow LSM to deny attackers from exfiltrating confidential data in the following way: First, it checks whether each packet bears the evil bit. The network filter logs evil packets and alerts the logging server of their presence. Simultaneously, the network filter determines if the evil packet contains either a DNS request or a TCP SYN segment. If either of these conditions hold, the filter responds with either a spoofed DNS response or a spoofed TCP SYN ACK. Lastly, the network filter drops evil packets from further processing. If the packet is not marked with the evil bit, then the network filter merely forwards the packet to the external network. Evil packets passing through the network filter from the external network to the SimpleFlow network are blocked and logged, but not spoofed.

### 4.2.1 *Network filtering*

The network filter drops outgoing evil packets, as the process which generated them had access to confidential data. While researchers have found few evil packets on the Internet, an attacker could purposefully send them to a SimpleFlow host to cause a denial of service: the receiving SimpleFlow kernel would taint a process which read them. Thus the network filter also drops incoming evil packets. We do allow evil packets among the SimpleFlow hosts on the local network to increase the amount of information we collect. This seems a reasonable tradeoff in practice because: (1) we assume the attacker does not have a root account on our hosts, and thus he cannot arbitrarily create evil-bit packets and (2) local network activity will have both sides logged, so the actions of an attacker who creates a denial of service are evident.

We set the default policy of the `INPUT` and `OUTPUT` chains to `DENY` to drop all traffic in and out of the host itself with the exception of whitelists, and we set the default policy of the `FORWARD` chain to `ACCEPT`. Next, we configured the network filter to drop all evil-bit packets [8]:

```
iptables -I FORWARD 1 -m u32 --u32 "3&0x80>>7=1" \
         -j DROP.
```

This rule uses the match option (`-m`) to filter based on an expression which detects evil bits. If the `u32` expression (`3&0x80>>7=1`) produces true, then the evil bit was set on the packet and it is dropped. If it produces false, then the packet moves to the next rule in the iptables chain.

The `u32` expression is of the form:

```
<start> & <mask> = <range>.
```

This allows iptables to inspect a slice of the packet beginning at byte offset `start` (3), apply the `mask` (`0x80>>7`) to the value there, and determine if the result matches the `range` (1) of values specified. Thus we inspect bytes three through six which contain the IP flags where the evil bit exists. The mask `0x80` zeros out every bit except for the evil bit. The statement then shifts the binary string to the right seven bits and compares it to the value one. If the evil bit is set, then the string will equal one and the `u32` match will produce true. Otherwise, the `u32` match produces false.

With evil-bit filtering in place, we next configured the network filter to forward unfiltered packets to the external network. We configured the kernel to forward packets by adding `net.ipv4.ip_forward = 1` to `/etc/sysctl.conf`.

Our final task was to allow the network filter to create connections to the logging server. Heretofore the network filter would drop all traffic other than that which is being forwarded between the SimpleFlow network and the external network.

In order to allow outgoing connections from the network filter to the logging server, we added a rule to the `OUTPUT` chain:

```
iptables -I OUTPUT -d <log server IP> -m conntrack \
         --ctstate NEW,ESTABLISHED -j ACCEPT.
```

Another rule permits packets from the logging server on established connections. This rule resembles the one above, except it exists in the `INPUT` chain, matches source IP addresses (`-s`), and accepts the connection states `RELATED,ESTABLISHED`.

In order to support IPv6 filtering, we configured `ip6tables` and IPv6 forwarding. We configured the kernel to forward IPv6 packets by adding `net.ipv6.conf.all.forwarding = 1` to `/etc/sysctl.conf`. We then denied any IPv6 connections to the network filter by setting the default policy of the `INPUT` and `OUTPUT` chains to `DENY`. Next we added a rule that drops evil labeled IPv6 packets:

```
ip6tables -I FORWARD 1 -m u32 --u32 \
      "0&0xFFFFF=0xbad1e" -j DROP.
```

This rule, using the same matching methods explained earlier, checks to see if the flow label of the packet equals the evil label, and it drops matching packets. IPv6 forwarding also requires allowing ICMPv6 and IPv6 multicast address connections, as IPv6 uses neighbor discovery instead of ARP. We added a rule in each chain which allows the ICMPv6 protocol as well as a rule in the `INPUT` and `OUTPUT` chains which allow IPv6 multicast addresses. These rules follow the same format as the logging connection rules, but they do not use the connection state matching.

### 4.2.2 *Spoofing*

SimpleFlow's network filter uses a spoofer to complete TCP connections in order to gather information about the intentions of tainted processes. Without this, the network filter would block evil SYN packets, and tainted processes would never send any application-layer requests. We are interested in recording the insightful requests which the spoofer brings forth, such as HTTP POSTs. The spoofer also responds to evil DNS queries, which the network filter also blocks. The spoofer's DNS responses include an IP address outside of the SimpleFlow subnet which ensures the tainted process will direct follow-on traffic through the network filter.

We build our spoofer using Python and Scapy. Our implementation makes use of Scapy's `sniff` function, to which it passes our filtering function and spoofing function as the `lfilter` and `prn` parameters, respectively.

The `lfilter` parameter accepts a Boolean function which it uses to check and filter packets. We pass in a function that filters out all packets except for IPv4 or IPv6 packets that are tainted with the evil bit or label, respectively. The function determines if the packet bears the evil bit by inspecting it in a manner similar to the firewall rules we described earlier. If the packet gets through the filter it is passed to our spoofing function.

Our spoofing function completes TCP connections, provides DNS responses, and logs evil packets. The function first determines if the packet is a TCP SYN packet, a DNS request, or just a regular evil packet. In the case of a TCP SYN packet, the spoofer creates a new IP header (IPv4 or IPv6) by swapping the source and destination addresses from the original packet. Next, it creates a spoofed TCP SYN ACK header by swapping the source and destination ports, setting the acknowledgment number to the incoming sequence number plus one, and adding a new random sequence number. Lastly, the function sets the ACK flag of the segment and transmits the packet.

If the packet is a DNS request, then the function creates the spoofed IP header in the same manner, and it spoofs the UDP header by swapping the source and destination ports. The function then creates a DNS response, setting the ID value to the ID from the request and copying the question from the request. The function sets the QR field to one to indicate a response. Next, the function adds a DNS resource record containing the requested name and an IP address that is outside of the SimpleFlow subnet. Lastly, the function transmits the packet.

### 4.2.3 *Logging*

The network filter logs information about every response it spoofs, and it stores evil-bit packets on disk for later inspection. We also ran an instance of Snort which included a rule which checks for the evil bit in IPv4 packets and our flow-label field identifier in IPv6 packets. The network filter sends all of these logs to a centralized log server.

## 5 Practical use and performance

We put SimpleFlow to practical use during the 2016 Cyber-Defense Exercise (CDX) and during our 2016 Cadet Competitive Cyber Team (C3T) tryouts. We also measured the performance of SimpleFlow using a series of microbenchmarks.

**Cyber-Defense Exercise**: The CDX is an annual competition sponsored by the US National Security Agency (NSA). The exercise challenges a number of undergraduate institutions to design, implement, and defend a computer network against attack. The NSA builds the backbone exercise network, provides the scoring infrastructure, and acts as the competition referee. The NSA also fields a red cell which it tasks with trying to compromise the confidentiality, integrity, and availability of the competitors' networks. Our network included 27 virtual machines and four network devices.

The CDX requires each competing team to attach a number of known-compromised computers to their network. The NSA gives each school the task of sanitizing these computers before activating them. These computers represent the users present on each school's network, and the red cell uses them to gain initial access to the network during the competition. (One of the rules which increases the likelihood of successful red-cell attacks restricts teams from applying vendor-provided security updates to these user workstations; this is why our SimpleFlow prototype targeted Linux 3.10.) One aim of the red cell is to obtain and exfiltrate token files which exist on each competitor's computers; one aim of each competitor is to protect these tokens. Another publication further describes the CDX and our team [23].

We installed SimpleFlow on the CentOS user workstation, and we integrated SimpleFlow into the network's centralized logging and monitoring system. Thus the SimpleFlow host forwarded its system logs over a Transport Layer Security (TLS)-protected channel to a server running Graylog [1]. SimpleFlow's network filter also forwarded its logs to Graylog.

Due to the care with which our students sanitized the CentOS user workstation before the competition, the host running SimpleFlow was not compromised; thus we did not observe any live evil-bit traffic during the exercise. Despite this, we put SimpleFlow's network filter to practical use during the CDX. We identified in our DNS server logs a number of DNS domains to which the red cell would exfiltrate data from Windows hosts not protected by SimpleFlow. We configured SimpleFlow's network filter to always spoof these domains, and thus we gathered much more information about the exfiltration attempts than an outright blacklist of these DNS lookups would have allowed. With SimpleFlow's network filter, we were able to observe application-layer requests related to exfiltration attempts even while blocking those attempts.

We also used SimpleFlow extensively during the testing which preceded the exercise. We found that undergraduate students not previously familiar with SimpleFlow were able to recreate the activity taking place on the SimpleFlow host. For example, we ran commands such as:

```
wget --no-check-certificate -i confidential \
         --quiet -O /dev/null,
```

where the file `confidential` bears a confidential mark and contains a URL to load. By watching only the stream of logs aggregated in Graylog, the students could determine that the kernel had tainted `wget` due to its interaction with the file `confidential`, the user running `wget`, `wget`'s process ID, and that `wget` had produced evil packets. By correlating these events with packet captures, the students could recognize the DNS lookup and HTTP GET request produced by `wget` even though SimpleFlow's network filter forbid the evil packets from leaving the SimpleFlow subnet. Indeed, our team wrote a number of Snort and Graylog alerts which would notify us if the monitoring system detected any evil packets.

Similar results followed from more sophisticated command sequences which involved some amount of Inter-Process Communication (IPC) before eventually producing a network packet. This provided much better network awareness in our students when compared to the previous year, when our team relied primarily on packet captures and traditional logs to recreate post-compromise activity. Figure 3a shows the SimpleFlow log messages resulting from a Unix pipeline.

**Cadet Competitive Cyber Team tryouts**: Each year the United States Military Academy hosts tryouts for its student-led C3T. Aside from an interview and an administrative review, the tryouts take the form of a Jeopardy-style capture-the-flag competition which spans seven days. We wrote a challenge based on SimpleFlow for this portion of the tryouts.

The SimpleFlow challenge consisted of a per-candidate Unix account which was available using `ssh` and whose home directory existed in a `chroot` jail. Inside each user's environment were the utilities `bash`, `ls`, `cat`, and `scp` as well as a confidential file named `flag`. The object of the challenge was to read the contents of `flag`; the presence of `scp` allowed candidates to copy in programs which they wrote and compiled elsewhere. The ordinary means of reading `flag` (e.g., `cat flag`) would cause `sshd` to become tainted and thus unresponsive until the user initiated another `ssh` session.

We expected candidates to solve this challenge by writing a program to repeatedly fork a child process which reads `flag` and signals an eight-bit portion of `flag` to is parent using an exit code. Figure 4 depicts this technique in pseudocode, and Figure 3b shows the logs SimpleFlow produces when `exfil` runs while labeled as wait-trusted. Success indicates a basic understanding of covert channels and the inner workings of SimpleFlow.

Twenty C3T candidates, members, and alumni attempted the SimpleFlow challenge. Two undergraduate C3T members and three graduate alumni solved our challenge using the technique we expected. As a result of this, we revised our handling of the `wait`/`exit` channel as described in §4.1.3. Some of our graduate alumni went on to demonstrate other channels, which we also described in §4.1.3.

**Performance**: We used `lmbench` 2 [19] to measure the relative performance of our SimpleFlow kernel and a vanilla Linux kernel of the same version. Our test computer was a 3.4 GHz Intel Core i7-4770 Pro with four cores and 32 GB of memory. We ran `lmbench` five times on each kernel, produced mean values for each kernel's benchmark runs, and then calculated the percent overhead cost of SimpleFlow. We also compared SimpleFlow's overhead to the that of LPM/Provmon. Since the authors of LPM measured using `lmbench` on a different type of computer, we compared only the overhead percentage. While an imperfect comparison, the dual Xeon quad-core computer used in the LPM paper is near enough to our i7 that our results illustrate the trends involved. Table 2 summarizes the key measurements, and the full results from `lmbench` appear in the Appendix.

All of the overheads we measured were less than or equal

```
tainting process running cat (pid: 2572, euid: 1000, uid: 1000) due to getattr interaction with /home/test/confidential
cat tainted; mark FIFO at pipe:[17902]
tainting process running wget (pid: 2575, euid: 1000, uid: 1000) due to read interaction with pipe:[17902]
wget tainted; mark socket at socket:[17906] [repeated a number of times]
setting evil bit on packet generated by wget (pid: 2575, euid: 1000, uid: 1000) [repeated a number of times]
```

(a) SimpleFlow logs resulting from running `cat confidential | test_evil_bit wget -no-check-certificate -i - --quiet -O /dev/null`

```
tainting process running exfil (pid: 28353, euid: 1003, uid: 1003) due to read interaction with /home/test/confidential
untainted exfil waiting on tainted exfil (could transfer one-byte exit code)
tainting process running exfil (pid: 28354, euid: 1003, uid: 1003) due to read interaction with /home/test/confidential
untainted exfil waiting on tainted exfil (could transfer one-byte exit code)
tainting process running exfil (pid: 28355, euid: 1003, uid: 1003) due to read interaction with /home/test/confidential
untainted exfil waiting on tainted exfil (could transfer one-byte exit code)
tainting process running exfil (pid: 28356, euid: 1003, uid: 1003) due to read interaction with /home/test/confidential
untainted exfil waiting on tainted exfil (could transfer one-byte exit code)
```

(b) SimpleFlow logs resulting from a wait-trusted command receiving four bytes of confidential data through a `wait/exit` channel; each byte incurs a cost of one `fork`, one `wait`, and one `exit`; an administrator could remove this channel and ensure SimpleFlow taints the parent `exfil` process by labeling the `exfil` program as untrusted.

**Figure 3:** Examples of SimpleFlow's logging; the file at `/home/test/confidential` bears the confidential label

```
for (int i = 0; i < 8; i++)
  pid = fork()
  if (pid != 0)
    waitpid(pid, &exitcode)
    print(exitcode)
  else
    f = open("confidential")
    seek(f, i)
    c = getc(f)
    exit(c)
```

**Figure 4:** Pseudocode for `exfil`, a program which communicates eight bytes of confidential data between two processes using `wait` and `exit`

to those of LPM/Provmon. We posit that this is due to SimpleFlow's more focused provenance and the lower overhead of maintaining a taint status on processes. Additionally, nearly all of the overheads we measured are near to or better than those measured in an early SELinux paper [17, Table 10]. The exception is `AF_UNIX`, where the overhead comes from locking the Unix socket in order to identify the taint status of its peer. We note that we have not yet attempted any optimization of the SimpleFlow code.

## 6 Conclusion

We set out to create a new access control system which would support the study of naïve and malicious users while providing a simple model for undergraduate experimentation. SimpleFlow permits most operations requested of the kernel, yet it taints processes which read data marked confidential, and it labels packets generated by tainted processes with the RFC 3514 evil bit. This allows user activity to continue unimpeded until a tainted process tries to communicate using the network. The presence of the evil bit makes it easy to block and record such traffic. In our practical tests, undergraduate students were able to describe in detail what was taking place during simulated malicious activity.

In addition to blocking evil-bit traffic, the SimpleFlow network filter responds to tainted DNS requests and completes three-way handshakes on behalf of what would otherwise be the destination host. This reveals information about the intentions of a user who reads a confidential file and later has his traffic blocked. During the CDX we found this useful even when dealing with Windows hosts not protected by SimpleFlow, because SimpleFlow's spoofing allowed us to extract information about exfiltration attempts to blacklisted DNS domains.

| Benchmark | Base | SimpleFlow | overhead (%) | Δ overhead vs. LPM [4, Table 1] |
|---|---|---|---|---|
| `null call` | 0.38 | 0.38 | 0.00 | 0.00 |
| `null I/O` | 0.44 | 0.60 | 33.75 | −16.25 |
| `stat` | 1.02 | 1.04 | 1.96 | −76.04 |
| `open/close` | 2.12 | 2.16 | 1.51 | −40.49 |
| `mmap` | 3,245.60 | 3,250.20 | 0.14 | −4.86 |
| `pipe` | 6.08 | 6.83 | 12.39 | n/a |
| `create 0KB` | 12.50 | 12.76 | 2.08 | −34.92 |
| `delete 0KB` | 10.90 | 11.12 | 2.02 | −36.98 |
| `create 10KB` | 26.10 | 27.10 | 3.83 | −19.17 |
| `delete 10KB` | 14.14 | 14.54 | 2.83 | −15.17 |
| `sig. inst.` | 0.44 | 0.44 | 0.00 | 0.00 |
| `sig. handle` | 1.03 | 1.04 | 0.58 | −0.42 |
| `fork` | 60.68 | 61.74 | 1.75 | −4.25 |
| `exec` | 260.40 | 262.80 | 0.92 | −3.08 |
| `sh` | 1,145.00 | 1,182.00 | 3.23 | −0.77 |
| `AF_UNIX` | 6.18 | 8.88 | 43.67 | n/a |
| `UDP` | 9.63 | 10.10 | 4.85 | n/a |
| `TCP` | 11.30 | 12.00 | 6.19 | n/a |
| `TCP connect` | 17.38 | 17.56 | 1.04 | n/a |
| `select TCP` | 2.80 | 2.65 | −2.19 | −2.19 |
| `prot. fault` | 0.46 | 0.47 | 1.59 | −6.41 |
| `page fault` | 1.00 | 1.00 | 0.00 | 0.00 |

**Table 2:** Key `lmbench` 2 benchmarks, including average vanilla and SimpleFlow kernel runtimes, SimpleFlow percent overhead, and difference between LPM/Provmon and SimpleFlow overhead

SimpleFlow demonstrates it possible to build an information-flow-based access control system on top of the LSM interface. The kernel component of SimpleFlow is roughly 2,100 lines of C code, and it exists almost entirely as a LSM. Aside from some evil-bit-related constant definitions, we needed only add a LSM call to `pipe_read` and write one new LSM hook (`security_file_lseek`). SimpleFlow incurs a small performance penalty which is similar to early versions of SELinux and smaller than LPM/Provmon. SimpleFlow is available at https://www.flyn.org/projects/SimpleFlow/.

Future work on SimpleFlow will include performance optimizations, a deliberate code review, and a port to a newer

Linux kernel which supports LSM stacking. We also expect to gather additional data about the practical use of SimpleFlow in cyber-defense competitions. Finally, we hope to address X11 and similar programs as described in §4.1.5.

## Acknowledgments

This material is based upon work supported by the US National Science Foundation under grant CNS-1464121. We are grateful for our teammates from the 2016 US Military Academy CDX team, as they helped us put SimpleFlow to practical use. We also thank the organizers of the CDX, the red cell, and the competing teams. Finally, we thank the C3T candidates, members, and alumni who tested SimpleFlow during their 2016 tryouts, especially those who first solved the SimpleFlow challenge and provided practical demonstrations of unexpected or covert channels: Matthew Shockley, Frederick Ulrich, Oliver DiNallo, Daniel Ford, and Chris Maixner.

## References

[1] Graylog log management system. https://www.graylog.org/ [Accessed Jan 20, 2016].

[2] The netfilter.org project. http://www.netfilter.org/ [Accessed May 27, 2016].

[3] perlsec. http://perldoc.perl.org/perlsec.html [Accessed Jul 4, 2016].

[4] Bates, A., Tian, D. J., Butler, K. R., and Moyer, T. Trustworthy whole-system provenance for the Linux kernel. In *Proc. of the USENIX Security Symposium* (Washington, D.C., Aug. 2015), USENIX Association, pp. 319–334.

[5] Bellovin, S. RFC 3514: The security flag in the IPv4 header. https://www.ietf.org/rfc/rfc3514.txt [Accessed Jan 20, 2016], Apr. 2003. Status: INFORMATIONAL.

[6] Bernstein, D. J. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the 2007 ACM workshop on Computer security architecture* (New York, NY, USA, 2007), CSAW '07, ACM, pp. 1–10.

[7] Cisco Systems, Inc. Cisco IOS security command reference. https://www.cisco.com/c/en/us/td/docs/ios/12_2/security/command/reference/fsecur_r.html [Accessed Jul 26, 2016], Dec. 2013.

[8] Cox, B. I may be the only evil (bit) user on the Internet, Nov. 2015. https://blog.benjojo.co.uk/post/evil-bit-RFC3514-real-world-usage [Accessed Apr 25, 2016].

[9] Department of Defense. Trusted computer system evaluation criteria. Tech. Rep. DOD 5200.28–STD, U. S. Department of Defense, 1985.

[10] Dodd, M. N. CVS commit: `src/sbin/ping ping.8 ping.c src/share/man/man4 inet.4 ip.4 src/sys/netinet in.h in_pcb.h ip.h ip_input.c ip_output.c ip_var.h src/usr.bin/netstat inet.c`. FreeBSD CVS commit message, Apr. 2003. https://lists.freebsd.org/pipermail/cvs-all/2003-April/001098.html [Accessed Apr 25, 2016].

[11] Dodd, M. N. CVS commit: `src/sbin/ping ping.8 ping.c src/share/man/man4 inet.4 ip.4 src/sys/netinet in.h in_pcb.h ip.h ip_input.c ip_output.c ip_var.h src/usr.bin/netstat inet.c`. FreeBSD CVS commit message, Apr. 2003. https://lists.freebsd.org/pipermail/cvs-all/2003-April/001295.html [Accessed Apr 25, 2016].

[12] Garrett, M. Circumventing Ubuntu snap confinement, 2016. http://mjg59.dreamwidth.org/42320.html [Accessed Apr 28, 2016].

[13] Hayden, M. Stop disabling SELinux, 2013. http://stopdisablingselinux.com/ [Accessed Apr 25, 2016].

[14] Johns, M. S., Atkinson, R., and Thomas, G. RFC 5570: Common architecture label IPv6 security option (CALIPSO). https://tools.ietf.org/html/rfc5570 [Accessed Jul 26, 2016], July 2009. Status: INFORMATIONAL.

[15] Kent, S. RFC 1108: U.S. Department of Defense security options for the Internet Protocol. https://tools.ietf.org/html/rfc1108 [Accessed Jul 26, 2016], Nov. 1991. Status: INFORMATIONAL.

[16] Kilpatrick, D., Salamon, W., and Vance, C. Securing the X Window System with SELinux, 2003.

[17] Loscocco, P., and Smalley, S. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, June 2001), The USENIX Association, pp. 29–42.

[18] McIlroy, M. D., and Reeds, J. A. Multilevel security in the UNIX tradition. *Software–Practice and Experience 22* (1992), 673–694.

[19] McVoy, L., and Staelin, C. `lmbench`: Portable tools for performance analysis. In *Proc. of the USENIX Annual Technical Conference* (pub-USENIX:adr, 1996), USENIX, Ed., USENIX Conference Proceedings 1996, USENIX, pp. 279–294.

[20] Morris, J. Have you driven an SELinux lately? In *Proceedings of the Linux Symposium* (July 2008), vol. 2 of *OLS '08*, pp. 101–114.

[21] Myers, A. C., and Liskov, B. Protecting privacy using the decentralized label model. *Software Engineering and Methodology 9*, 4 (2000), 410–442.

[22] Nakamura, Y., Sameshima, Y., and Tabata, T. SEEdit: SELinux security policy configuration system with higher level language. In *Proc. of the Conference on Large Installation System Administration (LISA)* (Berkeley, CA, USA, 2009), LISA'09, USENIX Association, pp. 8–8.

[23] Petullo, W. M., Moses, K., Klimkowski, B., Hand, R., and Olson, K. The use of cyber-defense exercises in undergraduate computing education. In *Proceedings of the 2016 USENIX Workshop on Advances in Security Education* (Washington, DC, USA, Aug. 2016), ASE '16, USENIX Association.

[24] Pohly, D. J., McLaughlin, S., McDaniel, P., and Butler, K. Hi-Fi: Collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 259–268.

[25] Postel, J. RFC 791: Internet Protocol, Sept. 1981. Obsoletes RFC076. See also STD0005. Status: STANDARD.

[26] Provos, N., Friedl, M., and Honeyman, P. Preventing privilege escalation. In *Proc. of the USENIX Security Symposium* (Berkeley, CA, USA, Aug. 2003), USENIX Association, pp. 231–242.

[27] Red Hat Bugzilla. Patch (and other utilities) act oddly with respect to custom Linux LSM (i.e., replacing SELinux), 2016. https://bugzilla.redhat.com/show_bug.cgi?id=1312575 [Accessed Apr 26, 2016].

[28] Saltzer, J. H. Protection and the control of information sharing in Multics. *Communications of the ACM (CACM) 17*, 7 (July 1974), 388–402.

[29] Smalley, S., Vance, C., and Salamon, W. Implementing SELinux as a Linux security module. Report #01-043, NAI Labs, Dec. 2001. Revised April 2002.

[30] Stock, B., Lekies, S., Mueller, T., Spiegel, P., and Johns, M. Precise client-side protection against DOM-based cross-site scripting. In *Proc. of the USENIX Security Symposium* (San Diego, CA, Aug. 2014), USENIX Association, pp. 655–670.

[31] Wang, L. *JailX: Protecting users from X applications.* PhD thesis, University of Illinois at Chicago, 2006.

[32] Weissman, C. Security controls in the ADEPT-50 time-sharing system. *Proc. FJCC, AFIPS 35* (1969).

[33] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. Linux Security Modules: General security support for the Linux Kernel. In *Proc. of the USENIX Security Symposium* (San Francisco, Ca., 2002).

[34] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 116–127.

[35] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making information flow explicit in HiStar. In *Symposium on Operating System Design and Implementation (OSDI)* (Seattle, Washington, Nov. 2006).

## APPENDIX: `lmbench` results with vanilla and SimpleFlow kernel

```
Processor, Processes - times in microseconds - smaller is better
------------------------------------------------------------------------------
Host                 OS  Mhz null null      open selct sig  sig  fork exec sh
                             call  I/O stat clos TCP   inst hndl proc proc proc
--------- ------------- ---- ---- ---- ---- ---- ----- ---- ---- ---- ---- ----
Vanilla   Linux 3.10.0V 3400 0.38 0.45 1.02 2.11 2.622 0.44 1.06 60.7 259. 1111
Vanilla   Linux 3.10.0V 3400 0.38 0.44 1.02 2.12 2.818 0.44 1.03 62.4 260. 1150
Vanilla   Linux 3.10.0V 3400 0.38 0.44 1.02 2.14 2.720 0.44 1.03 60.1 262. 1145
Vanilla   Linux 3.10.0V 3400 0.38 0.44 1.03 2.12 2.670 0.44 1.03 60.0 260. 1163
Vanilla   Linux 3.10.0V 3400 0.37 0.44 1.02 2.13 2.715 0.45 1.02 60.2 261. 1156

Context switching - times in microseconds - smaller is better
-------------------------------------------------------------------------
Host                 OS 2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
                        ctxsw  ctxsw  ctxsw ctxsw  ctxsw   ctxsw   ctxsw
--------- ------------- ----- ------ ------ ------ ------ ------- -------
Vanilla   Linux 3.10.0V 1.900 2.0000 3.1800 2.4200 2.7000 2.68000 3.10000
Vanilla   Linux 3.10.0V 1.920 2.0100 2.1700 2.3400 2.8400 2.52000 3.10000
Vanilla   Linux 3.10.0V 1.920 2.0100 3.1600 2.4300 2.8900 2.65000 3.22000
Vanilla   Linux 3.10.0V 1.990 1.9800 2.1800 2.3300 2.7000 2.59000 3.11000
Vanilla   Linux 3.10.0V 1.960 2.0400 3.0900 2.5000 2.8700 2.73000 3.11000

*Local* Communication latencies in microseconds - smaller is better
---------------------------------------------------------------------
Host                 OS 2p/0K  Pipe AF     UDP  RPC/   TCP  RPC/ TCP
                        ctxsw       UNIX         UDP         TCP conn
--------- ------------- ----- ----- ---- ----- ----- ----- ----- ----
Vanilla   Linux 3.10.0V 1.900 6.121 6.16 9.545        11.3       17.6
Vanilla   Linux 3.10.0V 1.920 6.133 5.95 9.587        11.3       17.3
Vanilla   Linux 3.10.0V 1.920 5.987 6.32 9.743        11.3       17.2
Vanilla   Linux 3.10.0V 1.990 6.086 6.15 9.605        11.3       17.3
Vanilla   Linux 3.10.0V 1.960 6.061 6.31 9.684        11.3       17.5

File & VM system latencies in microseconds - smaller is better
-------------------------------------------------------------------------------
Host                 OS   0K File      10K File     Mmap    Prot   Page
                        Create Delete Create Delete Latency Fault  Fault
--------- ------------- ------ ------ ------ ------ ------- -----  -----
Vanilla   Linux 3.10.0V   14.5   10.9   29.0   14.2  3205.0 0.465   1.000
Vanilla   Linux 3.10.0V   12.0   10.9   25.0   14.1  3256.0 0.461   1.000
Vanilla   Linux 3.10.0V   12.1   10.9   25.1   14.1  3266.0 0.463   1.000
Vanilla   Linux 3.10.0V   11.9   10.9   26.3   14.1  3253.0 0.465   1.000
Vanilla   Linux 3.10.0V   12.0   10.9   25.1   14.2  3248.0 0.466   1.000

*Local* Communication bandwidths in MB/s - bigger is better
-----------------------------------------------------------------------------
Host                OS  Pipe AF    TCP  File   Mmap  Bcopy  Bcopy  Mem   Mem
                             UNIX      reread reread (libc) (hand) read write
--------- ------------- ---- ---- ---- ------ ------ ------ ------ ---- -----
Vanilla   Linux 3.10.0V 3285 11.K 6859 6347.7  13.8K 9159.1 6671.2 13.K 9746.
Vanilla   Linux 3.10.0V 3533 10.K 6598 6336.6  13.8K 9131.7 6667.9 13.K 9810.
Vanilla   Linux 3.10.0V 3528 11.K 10.K 6350.4  13.8K 9101.7 6663.7 13.K 9836.
Vanilla   Linux 3.10.0V 3320 10.K 6666 6354.6  13.8K 9152.9 6678.8 13.K 9918.
Vanilla   Linux 3.10.0V 3615 10.K 10.K 6341.9  13.7K 9161.9 6667.6 13.K 9888.
```

```
Processor, Processes - times in microseconds - smaller is better
------------------------------------------------------------------------------
Host                 OS  Mhz null null      open selct sig  sig  fork exec sh
                             call  I/O stat clos TCP   inst hndl proc proc proc
--------- ------------- ---- ---- ---- ---- ---- ----- ---- ---- ---- ---- ----
SimpleFlo Linux 3.10.0S 3400 0.37 0.60 1.04 2.14 2.622 0.45 1.04 60.5 270. 1165
SimpleFlo Linux 3.10.0S 3400 0.38 0.60 1.04 2.18 2.622 0.44 1.04 62.4 269. 1167
SimpleFlo Linux 3.10.0S 3400 0.38 0.60 1.04 2.15 2.713 0.44 1.04 62.5 251. 1206
SimpleFlo Linux 3.10.0S 3400 0.38 0.60 1.05 2.17 2.668 0.45 1.04 62.3 252. 1193
SimpleFlo Linux 3.10.0S 3400 0.38 0.60 1.04 2.14 2.623 0.44 1.04 61.0 272. 1179

Context switching - times in microseconds - smaller is better
-------------------------------------------------------------------------
Host                 OS 2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
                        ctxsw  ctxsw  ctxsw ctxsw  ctxsw   ctxsw   ctxsw
--------- ------------- ----- ------ ------ ------ ------ ------- -------
SimpleFlo Linux 3.10.0S 1.900 1.9900 2.2200 2.2600 2.6600 2.46000 3.07000
SimpleFlo Linux 3.10.0S 1.820 2.9600 3.3000 2.4800 2.7500 2.65000 3.20000
SimpleFlo Linux 3.10.0S 1.820 1.9300 2.2400 2.2700 3.0100 2.66000 3.15000
SimpleFlo Linux 3.10.0S 2.900 1.9700 2.2000 2.4600 2.7100 2.50000 3.05000
SimpleFlo Linux 3.10.0S 1.830 1.9300 2.2000 2.4700 2.7000 2.70000 3.21000

*Local* Communication latencies in microseconds - smaller is better
---------------------------------------------------------------------
Host                 OS 2p/0K  Pipe AF     UDP  RPC/   TCP  RPC/ TCP
                        ctxsw       UNIX         UDP         TCP conn
--------- ------------- ----- ----- ---- ----- ----- ----- ----- ----
SimpleFlo Linux 3.10.0S 1.900 6.854 9.04  10.1  13.2  12.0  16.3 17.5
SimpleFlo Linux 3.10.0S 1.820 6.724 9.08  10.1  13.3  12.1  16.4 17.8
SimpleFlo Linux 3.10.0S 1.820 6.869 8.72  10.1  13.2  12.0  16.3 17.7
SimpleFlo Linux 3.10.0S 2.900 6.964 8.65  10.0  13.3  12.0  16.3 17.4
SimpleFlo Linux 3.10.0S 1.830 6.741 8.89  10.2  13.3  11.9  16.3 17.4

File & VM system latencies in microseconds - smaller is better
-------------------------------------------------------------------------------
Host                 OS   0K File      10K File     Mmap    Prot   Page
                        Create Delete Create Delete Latency Fault  Fault
--------- ------------- ------ ------ ------ ------ ------- -----  -----
SimpleFlo Linux 3.10.0S   13.7   11.1   28.1   14.6  3240.0 0.472   1.000
SimpleFlo Linux 3.10.0S   12.3   11.1   26.4   14.5  3240.0 0.471   1.000
SimpleFlo Linux 3.10.0S   12.8   11.2   27.9   14.6  3256.0 0.472   1.000
SimpleFlo Linux 3.10.0S   12.6   11.1   26.4   14.5  3271.0 0.473   1.000
SimpleFlo Linux 3.10.0S   12.4   11.1   26.7   14.5  3244.0 0.469   1.000

*Local* Communication bandwidths in MB/s - bigger is better
-----------------------------------------------------------------------------
Host                OS  Pipe AF    TCP  File   Mmap  Bcopy  Bcopy  Mem   Mem
                             UNIX      reread reread (libc) (hand) read write
--------- ------------- ---- ---- ---- ------ ------ ------ ------ ---- -----
SimpleFlo Linux 3.10.0S 3585 9757 6275 6250.8  13.8K 9177.3 6658.0 13.K 9881.
SimpleFlo Linux 3.10.0S 2962 9687 6281 6250.1  13.8K 9208.5 6641.1 13.K 9721.
SimpleFlo Linux 3.10.0S 3045 9769 6276 6243.6  13.8K 9118.4 6651.2 13.K 9862.
SimpleFlo Linux 3.10.0S 3058 9647 6272 6250.8  13.8K 9153.2 6648.1 13.K 9876.
SimpleFlo Linux 3.10.0S 3073 9880 6219 6242.0  13.8K 9137.3 6644.9 13.K 9836.
```