2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

# $\mathcal{SP}$Linux*: An Information Flow Secure Linux

Parjanya Vyas
*IIT Bombay*
Mumbai, India
vyas.parjanya@gmail.com

RK Shyamasundar
*IIT Bombay*
Mumbai, India
shyamasundar@gmail.com

Bhagyesh Patil
*IIT Bombay*
Mumbai, India
bhagyesh.patil.94@gmail.com

Snehal Borse
*IIT Bombay*
Mumbai, India
borsesnehal@gmail.com

Satyaki Sen
*IIT Bombay*
Mumbai, India
satyaki.sen2012@gmail.com

***Abstract*—Enforcement of information flow control (IFC) policies for OS to realize a sufficiently secure OS has been a challenging area of research. In this paper, our primary objective has been to present a fully information flow (IF) secure Linux that is usable with a minimal overhead without losing any of the existing functionalities. Towards such a goal, we describe the design, implementation and evaluation of a fully information flow secure Linux operating system called '$\mathcal{SP}Linux$' through complete mediation. Our approach first derives a labeled system (with initial inputs from the user in terms of the given Linux DAC policy) and manages further the labels automatically without users' intervention. It realizes complete mediation by interception of system calls and enforces IFC policy by implementing a recent decentralized security model that supports dynamic labelling and robust declassification. One of the distinct characteristics of the work is that the user has at his disposal all the features of Linux. We describe our experimental evaluation of $\mathcal{SP}Linux$, its assessment of usability and performance evaluation with respect to other secure OS efforts. Results are quite encouraging in terms of performance, expressiveness, and usability.**

***Index Terms*—Information Flow Control, Secure OS**

## I. Introduction

In an untrusted computing environment, assurance of computing systems plays a vital role. According to [1] "Assurance is evidence that a Computer system is secure, i.e., obeys its security specification, referred to as security policy". To assure security of a system, it is important that all of its' TCB components work as mandated/specified. Operating Systems(OSs) are a special class of systems for which security is vital as that is one of the crucial trusted components of the overall system. To quote from a white paper on IoT security [2], "Software security controls need to be introduced at the operating system level, take advantage of the hardware security capabilities now entering the market, and extend up through the device stack to continuously maintain the trusted computing base". An excellent survey on security of OS till 2008 is in [3] and several interesting possibilities of how secure and reliable an OS can be is explored in [4].

Seminal works on BLP [5], Lattice Flow [6], lead to the development of 'Orange Book' with support from the US Department of Defense which in turn lead to *Trusted Computer System Evaluation Criteria (TCSEC)* [7] that covered standards including OSs. One of the first IF secure OS level is designated [7], **B1-Labeled Security Protection** providing MAC that includes IF policies. As of now, we don't have any full-fledged usable OS that falls under this category. In this paper, we shall recast a given Linux into an IF-secure OS (hence, B1-labeled) without much overhead.

There have been several attempts to build secure OSs using IFC like HiStar [8], Flume [9] etc., based on the seminal DLM model [10]. These significant attempts, in particular, HiStar had high hopes as it started from a scratch, built a minimal kernel and isolated untrusted code using a little trusted code using the decentralized label model [10]. However, it did not reach the level of expectation of a complete usable expressive OS, *a la* Linux; as of now, HiStar and Flume prototypes are no longer maintained and available even for experimentation. Another practical effort has been SELinux [11] that has been used in a spectrum of applications and even adapted for Android. Security by augmenting Linux with a policy specification language that can express mandatory policies like BLP. However, it again falls short of B1 category highlighted above. IFC provides a uniform composable basis to build secure systems that can fulfill end-to-end security requirements of confidentiality and integrity. Though building an OS from scratch that provides provable security is a laudable task, securing the widespread existing OSs such as Linux is much more profitable from a practical and deployment perspective. In this paper, we present a fully functional Linux OS that ensures complete mediation through a system call interceptor and uses a verifiable decentralized security model called RWFM [12], [13] for ensuring IF security across the entire system. We encompass all communication channels provided as part of the standard Linux distribution including all major inter process communication (IPC) mechanisms as well as the file system. We present examples of security breach scenarios present in current Linux distributions and how they are prevented in $\mathcal{SP}Linux$. We also evaluate the performance of our system by comparing the overhead with base Linux distribution and SELinux. Results clearly show

*SP stands for Sandesh Palak meaning *information flow security* in *Sanskrit*.

978-0-7381-2646-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00214

that our system provides security guarantees as per the policy and is more expressive than SELinux while maintaining the practical usability by keeping in tact the functionality provided by the base Linux distribution; the overhead is quite bearable, even without pushing the implementation into the kernel. Our performance evaluation with respect to other OS's clearly shows that if we push our implementation into the kernel, the overhead will be insignificant. We further validate this extrapolation, by pushing some of the interceptor tasks to the kernel in section VI-D.

Rest of the paper is organized as follows: Section II provides an overview of RWFM followed by our approach for complete mediation in III. Section IV discusses $\mathcal{SP}Linux$ architecture and validation semantics. Case studies are provided in Section V. Highlights of our implementation/performance analysis are described in Section VI. Section VII provides a comparative evaluation, followed by conclusion in Section VIII.

## II. Background on RWFM [12], [13]

RWFM label model: (i) explicitly captures the readers and writers of information, (ii) makes labels explicit with set of readers and influencers, and (iii) provides an intuition for its position in the lattice flow policy. It realizes confidentiality and integrity supporting dynamic labeling and declassification. It captures several well-known models like Bell-LaPadula model (BLP) [14], Biba [15] and Chinese Wall [16].

Primitive operations that cause information flow (IF) are: (i) when the subject reads an object, (ii) when the subject writes an object, (iii) when the subject downgrades an object, and (iv) when the subject creates a new object or subject. For each of the above mentioned operations, RWFM describes conditions under which it is safe for the various permitted operations.

**Labeling**: Let $S$ and $O$ be the set of subjects and objects in the system respectively. An RWFM label is defined as a triplet $(s, R, W)$, where $s \in S$ denotes the information owner in the class, $R \in 2^S$ denotes the set of permissible readers of the class and $W \in 2^S$ denotes the set of permissible writers of the class or subjects which have influenced the class.

**Read - Write Access Rules**: Let $owner(x)$, $R(x)$ and $W(x)$ be the functions mapping to the owner, readers and writers components of the label respectively. With the above labeling model, access rules of RWFM are specified as follows:

• A subject $s$ is allowed to read an object $o$ if $owner(s) \in R(o)$ and $R(o) \supseteq R(s)$ and $W(o) \subseteq W(s)$.

• A subject $s$ is allowed to write an object $o$ if $owner(s) \in W(o)$ and $R(s) \supseteq R(o)$ and $W(s) \subseteq W(o)$.

Downgrading (declassification) is permitted only by the owner,and additional readers are restricted to the set of stakeholders who have contributed to the information.

**Downgrade Rule**: *Subject $s$ with label $(s_1, R_1, W_1)$ requests to downgrade an object $o$ from its current label $(s_2, R_2, W_2)$ to $(s_3, R_3, W_3)$.*

If $(s \in R_2 \wedge s_1 = s_2 = s_3 \wedge R_1 = R_2 \wedge W_1 = W_2 = W_3 \wedge R_2 \subseteq R_3 \wedge (W_1 = \{s_1\} \vee (R_3 - R_2 \subseteq W_2)))$ then

ALLOW

Else

DENY

**Create Rule**: When *Subject $s$ labeled $(s_1, R_1, W_1)$ requests creation of object $o$*, a new object $o$ is created with label $(s_1, R_1, W_1)$.

The above transition system accurately computes the labels for the newly created information at various stages of the transaction. It satisfies the following properties: (i) subject labels float upwards only, (ii) for a subject $s$, $A(s) = s$, $s \in R(s)$, and $s \in W(s)$, (iii) the set of writers of information is always accurately maintained, (iv) label of newly created objects precisely reflects circumstances under which it is created, (v) downgrade rule is within the boundaries of the flows permissible under a given transaction, and (vi) multiple sessions of the same subject are handled cleanly.

## III. Complete Mediation

Security of an OS is defined by its policy and enforcement mechanisms. As in [3], an OS is said to be secure with respect to its policy, when its enforcement mechanism (i) provides complete mediation, (ii) is tamper proof and (iii) is Verifiable. Thus, complete mediation is a very crucial property for realizing security; tamper resistance and verifiability are ensured by keeping the enforcement mechanism simple, and verifiable.

### A. Complete Mediation: Early Attempts

Complete mediation is an essential property that a system must ensure to guarantee the correct and complete enforcement of the defined security policy. One of the widely used approaches to enhance OS security such as SELinux [11] and AppArmor [17] utilize the LSM framework [18] that provides hooks into the system call code for intercepting sensitive operations. Though it has been very effective for discretionary access control (DAC) policies that are widespread in Linux distributions, it has been show through static analysis of system call control flow graph that placements of hooks are not sufficient to guarantee complete mediation for information flow control policies [19]. Moreover, maintaining and using a dynamic global DB of current system state is also difficult in this framework. LSM hooks are also prone to evasion particularly in the case of concurrent system calls. Laurent Georget et al., present two attacks to evade state-of-the-art LSM based trackers in [20]. Additionally, hooks cannot allow the enforcement mechanism to suppress the error messages returned to the calling process, which may reveal critical information that can be used by covert channels. Security models used in these approaches were based on static labeling that severely placed limitations; note that OS requires creation/destruction of a large number of subjects/objects dynamically.

### B. Complete Mediation: Our Approach

Instead of using predefined hooks, we intercept systematically all system calls before their execution starts and validate the operation according to the policy. This allows us to treat system call code as a black-box and control process's usage of

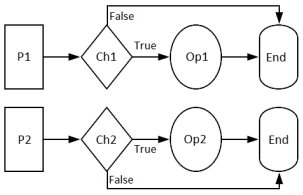


Fig. 1. Concurrent Processes $P_1$ and $P_2$

the black-box as an interceptor. Treatment of system calls as black-boxes is important as it simplifies the proof of complete mediation. Now all system calls can be treated as black-box constructs with varied inputs and outputs. Individual system call codes need not be analyzed as unique entities. Interceptor gets to choose whether to allow a process to use this black-box or not and what value to return as an answer to the calling process. Additionally, all errors and exceptions are returned to the interceptor so that the decision as to what error or exception to be passed can be effectively taken without any leak of information. Below, we show that all policies that can be enforced through LSM hooks can be enforced by the system call interceptor approach and show its' power to realize security adherence even in the context of concurrency.

### C. Policy Enforcement

**Theorem 1.** *Given that system calls are deterministic, all policies that can be enforced through LSM hooks, can also be enforced with a mandatory system call interceptor.*

*Proof.* Let $P$ be a policy that is to be enforced using LSM hooks as well as the system call interceptor (SCI). Let us assume that $S$ is the set of operations that are blocked by LSM but not by SCI. As both LSM and SCI start in the same initial state, sharing the same kernel state (subjects and objects), and the system calls are deterministic, $S$ must be empty. □

While all LSM interceptions w.r.t. policy $P$, is realized by SCI, the converse is not true (will be clear in the sequel).

### D. Security in Concurrent Environment

All modern operating systems allow multiple processes to run concurrently. Security must also hold in such a concurrent environment. The biggest problem with such environments is the 'Time of Check to Time of Use (TOCTOU)' attack [21]. Let there be two processes $P_1$ and $P_2$ with structures as shown in Figure 1. They contain similar structure with a check (Ch1 and Ch2) followed by an operation (Op1 and Op2) if the check returns true.

Concurrency might cause problems in terms of TOCTOU attacks in such scenarios. For example, let us consider the following possibility:

*If Op1 completes successfully then Ch2 fails, preventing operation Op2.*

If $P_1$ is scheduled after $P_2$, i.e., (Ch1→Op1→Ch2→Op2), there is no problem, as check Ch2 executes only after the completion of Op1. Hence, it will fail in the case of successful completion of Ch1. Consider concurrent execution of $P_1$ and $P_2$, with the schedule: Ch1→Ch2→Op1→Op2. In this case, as operation Op1 is yet to be done, but the check Ch2 happens before it. Op1 and Op2 both get executed, violating the security policy. As demonstrated in [20], LSM can fail to monitor some indirect flows when they are concurrent and affect the same containers of information. The root cause of such problems is that the sequence :occurrence of check, updating system state, is not atomic and thus making it prone to *race condition*. The only guarantee needed is the atomicity of checks as the check and update operations must happen atomically without the interleaving of any other process. Such a guarantee can be provided by ensuring mutual exclusion of all the check operations. A side effect of such a mechanism would be a slight increase in the overhead of the average system operations.

## IV. System Architecture and Design

### A. System Architecture

The architecture of $\mathcal{SP}Linux$ is shown in Figure 2. The system has three major components: *system call interceptor, validation rules, and system state database*. It is easily seen that the user application programs can reach the kernel only through the $\mathcal{SP}$-Center, ensuring complete mediation of all the operations requested by user processes. We have also brought the file system that has been explored in [22] into $\mathcal{SP}Linux$ with the following changes: in $\mathcal{SP}Linux$, the corresponding code base has been coded in C language, eliminating an extra indirection level and replacing the IPC done through TCP sockets by *System V* message queues. These changes along with careful bit level optimization tweaks have increased the efficiency of the file system related operations. Different modules of $\mathcal{SP}Linux$ are described below.

*1) System call interceptor:* Currently, instead of intercepting the actual system calls, we have developed a shared library called `preload` to intercept all the function calls analogous to the system calls defined in the dynamically linked shared library called `glibc`. `Preload` intercepts all the sensitive `glibc` function calls related to file open, read and write, and IPC mechanisms such as socket, named and unnamed pipe, message queue, shared memory, semaphores and process signaling. Each operation is then validated using the semantics derived from RWFM model and accordingly allowed to call the underlying actual `glibc` call or fail and report an error. Semantics is refined through validation rules given below.

*2) Validation Rules:* This module implements validation checks for all sensitive `glibc` function calls. Exact semantics of these checks are derived from the rules of RWFM and are discussed later. To ensure safety from TOCTOU attacks, these validation checks that may also include database updation are required to be atomic and mutually exclusive. We ensure these through *System V* semaphores. Based on the outcome of these checks, the module either returns true or false to `preload`, that in turn lets the function call pass or fail.

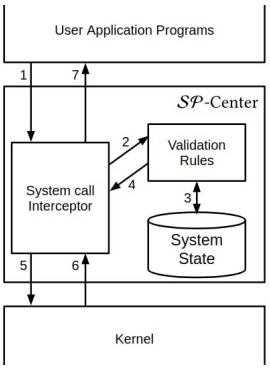


Fig. 2. System Architecture

*3) System State Database:* In contrast to other approaches using access control lists (ACL) or capabilities, we store the labels of system entities in the system state database. By hiding the database from the outside world, we ensure that only the enforcement mechanism accesses or modifies the security labels of system entities. Validation rules refer to the database to generate, access or modify labels for the system entities according to RWFM checks. Database can be accessed only through validation checks. Communication between validation checks and the database is done using separate *System V* message queues. Database access is restricted to validation checks by proper protection of the message queues.

### B. Illustration of the Enforcement Mechanism

Labelled edges in Figure 2, depict steps through which a typical user process operation request. When a process calls a sensitive `glibc` function that leads to IF, `preload` intercepts the call and calls the validation rule to check and validate the operation. Check refers to security labels related to entities involved in the operation from the certification database. It then proceeds to check whether the semantics permits the operation or not, and accordingly reverts to `preload` with a true or false. If the operation is authorized, `preload` will call the actual `glibc` function. Otherwise, it aborts and returns an error to the calling process.

### C. Modeling Validation Checks

Entities present in the system can be active or passive. Processes are active entities referred to as 'subjects' whereas files and intermediaries for IPC mechanisms are passive entities called 'objects'. Objects representing files, semaphore arrays, or message queues are statically labeled as they are persistent, whereas subjects/objects representing sockets, pipes or shared memories are dynamically labeled due to their transient nature. Reasons for the same can be found in [12].

*1) Database Model:* System entities can be classified into processes, files and IPC communication channels like socket, pipe, message queue, semaphore array, shared memory, and process signals. State of each of these entities is defined as a multiple relation with different attributes as shown in Table I.

TABLE I
SYSTEM STATE DATABASE MODEL

| Entity type | Attributes |
|---|---|
| Subject | **user id, process id, host id**, label |
| File | **host id, device id, inode no.**,label |
| Socket | **source-addr dest.-addr** peer-idlist label |
| Pipe | **host id, device id, inode no.**,label |
| Message queue | **host id, message queue id**, label |
| Shared memory | **key id**, label |
| ShM Map | **subject id, key, shmid, shmaddr** |
| Semaphore array | **host id, semaphore id**, label |

Bold-faced attributes together define the primary key for each of the entities.

*2) Validation Check Semantics:* Semantics of operations can be broadly classified: (i) Basic Operations, (ii) File system related operations and (iii) IPC related operations. Both (ii) and (iii) define operations that ultimately result in communication between two or more subjects via some intermediate object. Process signaling is the only operation that causes direct communication between subjects.

### D. Basic Operations

**Subject Creation** When a process creates another process using *fork* system call, we create a new subject and assign it the parent's label to reflect that the child process carries potentially all the information that the creating process has. As per standard semantics of fork, the child process inherits most of the parent process' memory space. If the parent process is out of scope of $\mathcal{SP}Linux$, then the child is given the lowest label in the RWFM lattice.

**Algorithm 1:** Subject Creation

```
1  create new subject for child process;
2  if parent's subject == φ then
3      subject's owner ← current user;
4      subject's readers ← all users present in the system;
5      subject's writers ← current user;
6  else
7      subject's owner ← current user;
8      subject's readers ← parent process' readers;
9      subject's writers ← parent process' writers;
10 end
```

**Object Creation:** When a process $p$ creates a new object (file or directory) using *create, mkdir or open* or open an existing object that was created out of the scope of $\mathcal{SP}Linux$, we create a new object in the database and infer its label from the read-write-execute permissions given to it as shown in Algorithm 2. Note that operations on line 14-15 prioritize permissions given to the owner over group permissions if the owner is also part of the group.

**Downgrade Label:** Robust declassification rule of RWFM model is

**Algorithm 2:** Object Creation

```
create new object;
owner ← creating process' owner;
readers ← φ;
writers ← φ;
get textitDAC permissions for owner, group and others;
if owner has read permission then
    readers ← readers ∪ {owner}
if owner has write permission on object then
    writers ← writers ∪ {owner}
if group has read permission on object then
    readers ← readers ∪ {user|user ∈ group }
if group has write permission on object then
    writers ← writers ∪ {user|user ∈ group }
if owner does not have read permission on object then
    object's readers ← object's readers - owner's readers
if owner does not have write permission on object then
    object's writers ← object's writers - owner's writers
if others has read permission on object then
    object's readers ← object's readers ∪ others' readers
if others has write permission on object then
    object's writers ← object's writers ∪ others' writers
```

implemented through the downgrade label as shown in Algorithm 3.

**Algorithm 3:** Downgrade Label L1 to L2

```
Let (s1, R1, W1) = L1 and (s2, R2, W2) = L2;
R ← R2 − R1;
if s1 = s2 ∧ W1 = W2 ∧ R ⊆ W1 then
    ALLOW;
else
    DENY;
end
```

### E. File Operations

IF can be caused by a subject interacting with a file. Hence, we intercept *read/write* calls Algorithms 4/ 5. Semantically the model is the same as the file system described in [22] but the implementation is realized in C-language with appropriate recasting of our DB model.

**Algorithm 4:** File Read

```
get subject of calling process;
get object representing the file or directory;
if subject ∈ object's readers then
    ALLOW;
    if subject's label ≠ object's label then
        subject's label ← subject's label ⊕ object's label;
else
    DENY;
end
```

**Algorithm 5:** File Write

```
get subject of calling process;
get object of file or directory;
if subject's owner ∈ object's writers ∧ subject's readers ⊇
  object's readers ∧ subject's writers ⊆ object's writers then
    ALLOW;
else
    DENY;
end
```

### F. IPC operations

IPC mechanisms we secure are: sockets, named and unnamed pipes, message queues, shared memory, semaphores & process signaling. There exist two different standards for message queues, shared memory and semaphore, namely *POSIX* and *System V*. Both standards specify and provide similar mechanisms with slight variations. We have chosen *System V* implementations for intercepting and securing operations. The same semantics with slight modifications in implementation would be able to secure their *POSIX* counterparts as well.

**Socket** For securing communication between processes through *sockets*, we intercept *connect, accept, send and recv* system calls. We create a *socket object* as in Section IV-C1, representing a connection via *socket* between two processes, during connect and accept calls that has exactly the same semantics as in Algorithm 6. These calls are allowed to be asynchronous, and the socket object label is guaranteed to be correct after the last of the two calls completes its execution. Semantics to secure the actual IF control are described in Algorithms 7 and 8 that are for send and recv calls.

**Algorithm 6:** Socket Connect and Accept

```
get calling process' subject;
if socket object not present then
    create new socket object;
    Initialize source and destination addresses;
    socket label ← current subject label;
else
    get existing socket object;
    socket label ← socket label ⊕ subject label;
end
update peer id list with current subject id;
```

**Algorithm 7:** Socket Send

```
get calling process' subject;
get socket object;
if socket object's label ≠ subject's label then
    socket label ← socket label ⊕ subject label ;
ALLOW;
```

**Pipe** For securing IPC via *pipes*, we intercept *pipe, open, read, write* system calls. We create a *pipe object* as described

**Algorithm 8:** Socket Recv

```
get calling process' subject;
get socket object;
if subject ∈ socket readers then
    ALLOW;
    if subject label ≠ socket label then
        subject label ← subject label ⊕ socket label;
else
    DENY;
end
```

in Section IV-C1 to represent an IPC connection using named or unnamed pipes. The only difference between named and unnamed pipes is how they are opened. When a subject creates a new pipe using *pipe()* or *mkfifo()* call, we create a new pipe object and assign it the subject's label. We maintain and change the state of *pipe object* for read and write system calls to ensure IF security according to Algorithms 9 and 10.

**Algorithm 9:** Pipe Read

```
get calling process' subject;
get pipe object;
if subject ∈ pipe object's readers then
    pipe label ← pipe label ⊕ subject label;
    ALLOW;
else
    DENY;
end
```

**Algorithm 10:** Pipe Write

```
get calling process' subject;
get pipe object;
pipe label ← pipe label ⊕ subject label;
ALLOW;
```

**Message Queue and Semaphore Array:** For securing communication between processes through *message queues* we intercept *msgget, msgrcv, msgsnd, msgctl*; for securing *semaphore array*, we intercept *semget, semop, semctl* system calls. We create a *message queue object* to represent communication using a message queue and a *semaphore object* to represent communication using a semaphore array. *System V* message queues and semaphore arrays are secured exactly like regular files using DAC permissions. They too are persistent objects. Subjects interact with these objects similar to regular files as far as IF are concerned. A message queue is read and written by a subject using msgrcv and msgsnd system calls respectively. Similarly, a semaphore array is read and written by subjects using semop system call with different parameters. We derive labels of these objects similar to that of files using Algorithm 2. The read and write checks are also similar to file read and file write checks as in Algorithms 4 and 5 with the difference that we use message queue objects and semaphore objects instead of file object when consulting object labels during checks for the respective IPC mechanisms.

**Shared Memory** Providing IF security for shared memory has been one of the biggest challenges. The primary reason is the lack of 'read' and 'write' like system calls. After attaching a shared memory segment to its memory space a process does not need any other system call to read or write in the memory segment. Instead, it uses direct memory addresses by means of pointers to access the memory. Such pointer accesses cannot be intercepted by any higher level interceptor such as LSM hooks or a system call interceptor. As described in [20], once a shared memory segment is attached to a process' memory space, the enforcement mechanism needs to make a conservative assumption that the process is sharing all its information with the shared memory, and hence, with all processes attached to this shared memory segment. However, a process can attach a large number of shared memory segments to itself, complicating the already complex problem.

We solve this problem by representing shared memory segments and subjects as nodes of a bipartite graph. A shared memory object is as defined in Section IV-C1. The state of system for shared memories at an instance can be modeled as a *bipartite graph* $G$ with $A \cup B$ as nodes, where A = *subjects*, B = *shared memory objects*, and an undirected edge between $A$ and $B$ denotes the *attached* relation. Due to the conservative assumption given above, the semantics of shared memory related system calls need to ensure that for each connected component in the graph, all nodes belonging to the same connected component can safely read and write each other. To prove that this condition is satisfied at all times, we prove Theorem 2 and 3 given below. The semantics for shmget and shmat are described in Algorithms 11 and 12. Whenever a subject who has attached a shared memory segment to itself, tries to upgrade its label from $(s, R_1, W_1)$ to $(s, R_2, W_2)$, then the label change operation is allowed only if Algorithm 13 validates it. This result is mandatory to prove Theorem 3.

**Theorem 2.** *Subjects $S_1$ and $S_2$ can both read from each other if and only if $R(S_1) = R(S_2)$ and $W(S_1) = W(S_2)$ where $R$ and $W$ denote functions that map subjects with their reader and writer sets respectively.*

*Proof.* According to read rule from Section II $S_1$ can read $S_2 \iff R(S_1) \subseteq R(S_2) \wedge W(S_1) \supseteq W(S_2)$. Similarly, $S_2$ can read $S_1 \iff R(S_2) \subseteq R(S_1) \wedge W(S_2) \supseteq W(S_1)$
Therefore, $R(S_1) = R(S_2)$ and $W(S_1) = W(S_2)$ □

**Theorem 3.** *Given that shmat system call is always guarded by Algorithm 12 and a label upgrade of a subject who has at least one shared memory segment attached to itself is always guarded by Algorithm 13, if the shared memory graph is denoted by $G$, and connected components are denoted by $C$, then $\forall C \in G, \forall S_1, S_2 \in C \Rightarrow R(S_1) = R(S_2) \wedge W(S_1) = W(S_2)$*

*Proof.* It is clear from Algorithm 12, that the shared memory is allowed to be attached to subject if and only if all reachable nodes from subject can read from all reachable nodes of shared memory object and vice versa. This along with Theorem 2

implies $\forall S_1, S_2 \in C \Rightarrow R(S_1) = R(S_2) \wedge W(S_1) = W(S_2)$ is true after successful execution of Algorithm 12.

This changes only when a subject label gets upgraded. But Algorithm 13 ensures that label upgradation is allowed only if labels of all reachable nodes from the subject can also be upgraded to the same lattice point. This in turn ensures the necessary condition of having exactly the same set of readers and writers of all the connected nodes. □

**Algorithm 11:** Shared memory shmget

```
get calling process' subject;
if key ∉ set of key objects then
    create new shared memory object with key;
    shared memory label ← subject label;
    create new ShM Map shm_map;
    shm_map.subject id ← calling process's subject id;
    shm_map.shmid ← shmid returned by shmget call;
    shm_map.shmaddr ← NULL;
ALLOW;
```

**Process signaling:** is the only IPC mechanism that provides direct communication between subjects without any intermediate object. For securing it, we intercept *kill, sigqueue, killpg* system calls. If process *P1* tries to signal process *P2*, a simple read check is performed to check whether *P2* is allowed to receive from *P1* as per Algorithm 4 using label of *P2* instead of object label. When a process signals a group of processes, the same check is performed for each receiving process, and the operation is allowed only if all checks pass as shown in Algorithm 14.

**Algorithm 12:** Shared memory shmat

```
get calling process subject;
get key object;
set U ← reachable(subject) ∪ reachable(key);
U_sub = U ∩ A;
U_key = U ∩ B ;
U_own = {owner(s)|s ∈ U_sub};
if U_own ⊆ readers(subject) ∩ readers(key) then
    foreach u ∈ U do
        label(u) ← label(u) ⊕ label(subject) ⊕ label(key);
    end
    add edge(subject,key) and edge(key,subject);
    ALLOW;
else
    DENY;
end
```

## V. Case Studies

We illustrate security provided by $\mathcal{SP}Linux$ through the classic spyware example used in Linux and show how attacks are prevented in $\mathcal{SP}Linux$ and demonstrate power of robust declassification mechanism through a scenario, where other IFC models wrongly deny a legitimate operation that $\mathcal{SP}Linux$ successfully recognizes to be valid.

**Algorithm 13:** Subject label upgrade from $L1$ to $L2$

```
Let (s, R1, W1) = L1 and (s, R2, W2) = L2;
get calling process subject;
set U ← reachable(subject);
U_sub = U ∩ A;
U_key = U ∩ B ;
U_own = {owner(s)|s ∈ U_sub};
if U_own ⊆ R2 then
    foreach u ∈ U do
        label(u) ← label(u) ⊕ L2;
    end
    ALLOW;
else
    DENY;
end
```

**Algorithm 14:** Process signal killmany

```
get subject;
foreach peer process do
    get peer process' subject;
    if peer subject's owner ∉ subject's readers then
        return Deny;
    foreach peer process do
        peer subject label ← peer subject label ⊕ subject
        label;
    end
    Allow;
end
```

### A. Spyware Example

Figure 3 depicts a spyware that leaks information from a secret file to an unclassified process. 'Secret File' is a file created by user 'u1' with read-write permissions given only to u1. 'Classified Process' is a process executed by u1 whereas 'Public Process' is another process executed by 'u2'. As shown in Figure 3a, in the base Linux, if an 'innocent looking' spyware can disguise itself as the classified process, then it can read the information contained in Secure File and pass it to an unclassified 'Public Process' that is owned by another user, resulting in an information leak.

$\mathcal{SP}Linux$ handles it as follows. As per the DAC permissions of the Secret File, using the object label inference algorithm (cf. Section IV-C2), the RWFM label of the file would be $(u1, \{u1\}, \{u1\})$. The initial label of Classified Process is $(u1, \{All\}, \{u1\})$ and Public Process is $(u2, \{All\}, \{u2\})$. When the spyware disguised as Classified Process reads the Secret File, its' label changes to $(u1, \{u1\}, \{u1\})$. When it tries to pass this information to Public Process using an IPC mechanism, the check $\{All\} \subseteq \{u1\} \wedge \{u2\} \supseteq \{u1\}$ would fail and IF will be blocked as shown in Figure 3b.


(a) Running in base Linux Distri.


(b) Secured with $\mathcal{SP}Linux$

Fig. 3. Spyware Example

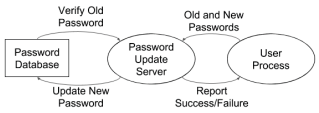


Fig. 4. A Legitimate Password Update Scenario

### B. A Password Update Example

Figure 4 shows a legitimate scenario where user 'u' tries to update his password by communicating with the password update server. Password Database is a sensitive file owned, read and written only by administrator 'a'. Password Update Server is a classified process owned by 'a' and the user Process is a normal process owned by user u. The sequence of events shown in Figure 4 are: (a) the user passes old and new passwords to the password update server, (b) password server queries the password database and matches the old password, (c) If passwords match, password server updates the new password by writing it into the database, and (d) Password server reverts back to the user process with success/failure. As per the information flow policy, the initial label of the Password Database, Password Update Server and User Process would be $(a, \{a\}.\{a\})$, $(a, \{All\}.\{a\})$, $(u, \{All\}.\{u\})$ respectively. According to normal IF rules, after the first three operations are done, the label of password server is updated to $(a, \{a\}.\{a, u\})$. The normal IF rules deny the last operation as it involves an invalid read operation and the user process would never receive a success or failure message. Due to declassification rule (Section IV-C2), the label of password server can be declassified to $(a, \{a, u\}.\{a, u\})$ and the final operation succeeds.

**Remarks:** Applying declassification rule for the Spyware example, the final write will fail – preventing information leak.

## VI. Implementation and Evaluation

We evaluate performance of $\mathcal{SP}Linux$ relative to SELinux. Overhead of $\mathcal{SP}Linux$ and SELinux is computed wrt performance of operations in the base Linux Source code of the user space/ kernel module implementations, performance benchmarking etc. are available at: https://github.com/anonymous-for-conferences/secureOS_Anonymized.

### A. Implementation Highlights

A schematic diagram of the implementation is shown in Figure 5. $\mathcal{SP}Linux$ uses Ubuntu 16.04.6 running with 4.4.0-145-generic Linux kernel (adaptable for others easily). All components are written in C.

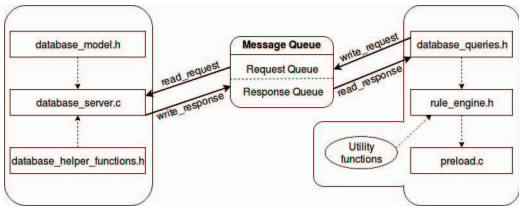


Fig. 5. Schematic Layout

`glibc` call interceptor is a shared library called `preload.c`, that wraps the actual libc shared library and consists of around 450 LOC (Lines of Code). `rule_engine.h` is a header file included in `preload` library that implements all the validation semantics defined in Section IV and consists of around 550 LOC.

DB model is defined in the header file `database_model.h` and implemented as a separate C file called `database_server.c` consisting of about 800 LOC. Information contained in DB is stored in the memory space of DB server process. A `database_helper_functions.h` file defines insert, delete and update operations for objects in DB and consists of about 500 LOC. DB server can be queried using special message queues called `REQUEST_MQ` and `RESPONSE_MQ`. DB server reads requests from `REQUEST_MQ`, performs queried operations by calling corresponding functions defined in `database_helper_functions.h`, and places the result in `RESPONSE_MQ`.

Although processes can directly perform DB operations by writing in `REQUEST_MQ` and reading from `RESPONSE_MQ`, we facilitate these operations by writing yet another header called `database_queries.h` that defines functions for requesting operations and reading responses from DB server.

The code that needs to be **trusted** is the **RWFM rule engine** which is a pure sequential small code and our TCB is minimal. Formal verification of such a small TCB is easy and hence, easy to realize tamper free property of the system (cf. Section IV). The entire project consists of about 3715 LOC in C language including comments and utility functions.

### B. Expressive Power of Policy and Usability

Labels for $\mathcal{SP}Linux$ are derived using DAC policies defined as **read-write-execute** permissions in base Linux. According to DAC, a subject can access an object only if that subject and/or the group to which the subject belongs has permission to access that object. Use of DAC permissions and building the IF policy through DAC, provides a succinct user-friendliness that has not been the case in earlier efforts. $\mathcal{SP}Linux$ can model any lattice flow policy as in RWFM [13] and all consistent SELinux policies can be transformed into RWFM [23].

### C. Performance Evaluation

For performance evaluation, we have scaled up benchmarks like ***lmbench*** and ***UnixBench*** to define a new set of performance benchmarks. File system benchmark measures total time taken

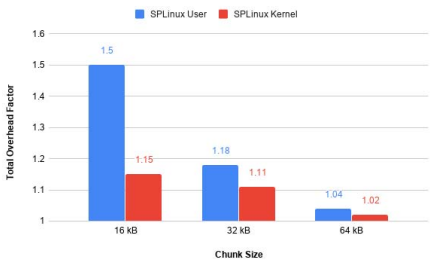


Fig. 6. Overhead in kernel & user space in $\mathcal{SP}Linux$

for various file related operations - `create a file, write to a file, read from a file, and read from and then write to a file`, using different chunk sizes. Pipe, FIFO, socket and message queue benchmarks measure total time taken to read/write and transfer data using unnamed pipe, named pipe, socket and message queue respectively. For every benchmark that we run, we use total data size as 10MB and chunk sizes of 16KB, 32KB, and 64KB. All time measurements are in *milliseconds*. Performance of fork() is shown in Table IV. We measure total time taken by different kernel configurations to create 10, 100 and 1000 processes.

We create a VM using Oracle VM VirtualBox on 3.50GHz 8 Core i7-4770K CPU with 16GB RAM & 1 TB hard disk. VM is installed with Ubuntu 16.04.6 with 4.4.0-145-generic Linux kernel with 16GB RAM. *base* kernel configuration is the unmodified Ubuntu 16.04.6, *SELinux* kernel uses SELinux version 31 in permissive mode and $\mathcal{SP}Linux$ configuration is as in Section VI-A.

From Tables II - III, we can see an increase in the total time taken by $\mathcal{SP}Linux$ as compared to Base. This is so, as the implementation of $\mathcal{SP}Linux$ is in user space that needs to maintain auxiliary data for all operations. From Figure 7, one can see that the overhead decreases as the chunk size increases. As the number of operations decreases as chunk size increases, we can say that the overhead with our system is constant. Overhead comparison of $\mathcal{SP}Linux$ with SELinux is shown in Figure 7; it can seen that the overhead is negligible. Major overhead arises in process creation as replication of auxiliary data of parent process to child process is done.

### D. $\mathcal{SP}Linux$ - User Space to Kernel Space

Auxiliary data maintained in user space and communication with DB server using message queues (cf. Figure 5) are the major contributors for overhead. (cf. section VI-C). To demonstrate expected improvement if we push the same into kernel, we implemented the file system related operations as a kernel module. The experimental kernel module modifies `sys_call_table` residing inside the kernel and replaces the default open, read, write, and clone system calls with the custom implementations that add rwfm rule checks and logs access decisions before invoking original system calls. We insert the kernel module using `insmod` and perform file system related experiments as in Section VI-C. As seen from Figure 6, the overhead decreases significantly when

TABLE II
TIME TAKEN FOR OPERATIONS WITH 16KB & 32KB CHUNK SIZE

| IPC | Base | $\mathcal{SP}L$ | SEL | Base | $\mathcal{SP}L$ | SEL |
|---|---|---|---|---|---|---|
| File | 551.67 | 829 | 583.67 | 809.67 | 953.67 | 796.67 |
| Pipe | 276.78 | 428.49 | 271.21 | 402.78 | 477.87 | 398.70 |
| FIFO | 290.69 | 442.45 | 319.38 | 400.73 | 445.96 | 412.79 |
| Socket | 286.89 | 393.45 | 271.29 | 318.65 | 379.13 | 318.01 |
| Mess.Q | 231.67 | 347.21 | 251.11 | 318.65 | 379.13 | 318.01 |

TABLE III
TIME TAKEN FOR OPERATIONS WITH 64KB CHUNK SIZE

| IPC Mechanism | Base | $\mathcal{SP}Linux$ | SELinux |
|---|---|---|---|
| File system | 1662.67 | 1723 | 1644.67 |
| Pipe | 828.46 | 879.37 | 826.98 |
| FIFO | 845.87 | 853.82 | 839.96 |
| Socket | 811.03 | 877.89 | 822.36 |
| Message Queue | 648.70 | 697.81 | 645.60 |

$\mathcal{SP}Linux$ is pushed inside the kernel space. For chunk size 16KB, overhead decreases from 1.5 to 1.15, whereas for 32KB it gets reduced and for 64 KB, the overhead is almost nil.

## VII. COMPARATIVE EVALUATION OF $\mathcal{SP}Linux$

SELinux [24] is a Linux kernel security module providing mechanisms for enforcing security policies. For type enforcement policy, every access is denied and audited by default, that can be overridden with access vector rules *allow, neverallow, auditallow* and *dontaudit*; however consistency of the policy is not guaranteed [23]. In contrast, $\mathcal{SP}Linux$ always recognizes inconsistencies wrt IF at run time and never allows contradictory policies and has distinct advantages like (i) almost no additional policy specification, (ii) all lattice flow policies are possible that is not the case with SELinux, and (iii) a SELinux policy is compiled only if it is IF consistent.

HiStar [8] provides strict IFC allowing users to specify precise data security policies with no notion of superuser. It tracks IF dynamically through tainting, labeling files with private user data as it gets tainted. It has a highly restricted functionality, even creating multi-users is quite a task and declassification is not robust. For instance, modelling of password update of Section V-B in HiStar, would require `wrap` to declassify the taint of password server **correctly** Security thus, depends solely on the trust on `wrap`. In contrast, $\mathcal{SP}Linux$ defines robust declassification rule (cf. Section IV-C2), its' advantages over HiStar are (i) full functionality of Linux, (ii) agents cannot misuse declassification privilege, and (iii) levels need not be fixed a priori statically.

Flume [9] uses DIFC model that runs as a user-level reference monitor on Linux. A process monitored by Flume cannot perform most system calls directly, an interception layer replaces system calls with IPC to the reference monitor, that enforces data flow policies and performs safe operations on behalf of the process. Flume uses tags/labels to track data flows and it relies on the trust of sensitive subjects

TABLE IV
TIME TAKEN FOR PROCESS CREATION

| No. of Proc. | Base | $\mathcal{SP}Linux$ | SELinux |
|---|---|---|---|
| 10 | 2 | 5.67 | 0 |
| 100 | 14.67 | 62.67 | 26 |
| 1000 | 115.33 | 546.33 | 291 |

TABLE V
OVERHEAD FACTORS

| Chunk Size | 16KB | | 32KB | | 64KB | |
|---|---|---|---|---|---|---|
| Operation | Kernel | User | Kernel | User | Kernel | User |
| Create | 1.96 | 0.94 | 1.02 | 1.15 | 1.02 | 1.14 |
| Write | 1.04 | 1.27 | 1.08 | 1.07 | 1 | 1.02 |
| Read Write | 1.15 | 1.39 | 1.1 | 1.13 | 1.02 | 1.03 |
| Read | 1.57 | 58.8 | 1.38 | 22 | 1.08 | 14 |
| Total | 1.15 | 1.5 | 1.11 | 1.18 | 1.02 | 1.04 |

for declassification. The issue of assurance of integrity of capability tags does not arise in $\mathcal{SP}Linux$. Advantages of $\mathcal{SP}Linux$ over Flume as are the same as given for HiStar[1].

## VIII. CONCLUSIONS AND FUTURE WORK

We have described the design and implementation of $\mathcal{SP}Linux$ that derives a fully IF secure Linux OS, without losing any functionality; IF policy is derived automatically from the underlying DAC, relieving the user from the burden of providing a complex IFC policy. Such an automation not only increases user-friendliness but also productivity of the system. We have provided case studies to demonstrate features of our system and evaluated it's performance using state-of-the-art benchmarks. Results show that overhead is barely visible to end users. Comparing our results with SELinux, we can conclude that if the interceptor is pushed inside the kernel, our system would have a similar overhead as SELinux

[1]While source code of Flume is not available, source code of HiStar is available at https://github.com/zeldovich/histar; as the systems are not maintained, we could not benchmark $\mathcal{SP}Linux$ with [8] or [9].

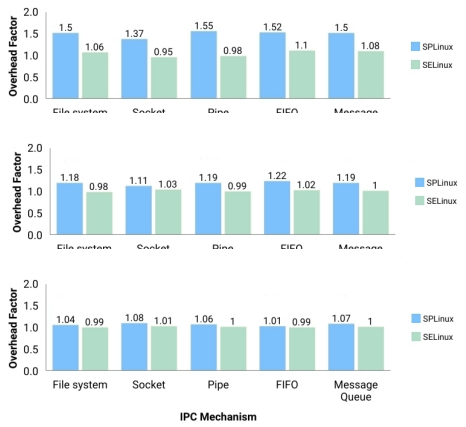


Fig. 7. Overhead SELinux vs $\mathcal{SP}Linux$: 16KB, 32KB,64KB

(validated in section VI-D) while providing a much better IF security in a user-friendly way. We are currently working on creating an LSM module that enforces the IFC checks described in this work. We intend to compare the performance and flexibility of this LSM module with the approach taken here. To sum up, $\mathcal{SP}Linux$ is in a sense the first B1-labelled OS with full Linux functionality.

## REFERENCES


[1] B. Lampson, "Making untrusted code useful: Technical perspective," *Commun. ACM*, vol. 54, no. 11, pp. 92–92, Nov. 2011.

[2] I. Wind River Systems, "Security in the iots, lessons from the past for the connected future," 2019. [Online]. Available: https://www.windriver.com/whitepapers/security/security-in-the-internet-of-things/

[3] T. Jaeger, *Operating System Security*, ser. G - Reference, Information and Interdisciplinary Subjects Series. Morgan & Claypool Publishers, 2008.

[4] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make os reliable and secure?" *Computer*, vol. 39, no. 5, pp. 44–51, May 2006.

[5] D. Bell and L. LaPadula, "Secure computer systems: mathematical foundations and model. mitre corp," *TR M74-244, Bedford, MA*, 1975.

[6] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[7] D. C. Latham, "Department of defense trusted computer system evaluation criteria," *Department of Defense*, 1986.

[8] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *7th OSDI*, 2006, pp. 263–278.

[9] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *21st ACM SIGOPS SOSP*, 2007, pp. 321–334.

[10] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. SEM*, vol. 9, no. 4, pp. 410–442, 2000.

[11] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system." in *USENIX Annual Technical Conference, FREENIX Track*, 2001, pp. 29–42.

[12] N. V. N. Kumar and R. K. Shyamasundar, "Realizing purpose-based privacy policies succinctly via information-flow labels," in *IEEE 4th Int. Conf. on Big Data and Cloud Computing*, 2014, pp. 753–760.

[13] ——, "A complete generative label model for lattice-based access control models," in *SEFM*. LNCS 10469, Springer, 2017, pp. 35–53.

[14] D. E. Bell and L. J. La Padula, "Secure computer system: Unified exposition and multics interpretation," Mitre Corp., Tech. Rep., 1976.

[15] K. J. Biba, "Integrity considerations for secure computer systems," MITRE CORP BEDFORD MA, Tech. Rep., 1977.

[16] D. F. Brewer and M. J. Nash, "The chinese wall security policy," in *Proc. 1989 IEEE Symp. on Security and Privacy*, pp. 206–214.

[17] C. Cowan and et al., "Subdomain: Parsimonious server security." in *LISA*, 2000, pp. 355–368.

[18] J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *USENIX Security Symp.* ACM Berkeley, CA, 2002, pp. 17–31.

[19] L. Georget, M. Jaume, F. Tronel, G. Piolle, and V. V. T. Tong, "Verifying the reliability of operating system-level information flow control systems in linux," in *IEEE/ACM 5th FormaliSE*, 2017, pp. 10–16.

[20] L. Georget, M. Jaume, G. Piolle, F. Tronel, and V. V. T. Tong, "Information flow tracking for linux handling concurrent system calls and shared memory," in *LNCS, 10469*. Springer, 2017, pp. 1–16.

[21] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan, "Concurrency attacks," in *4th USENIX W/S on Hot Topics in Parallelism*, 2012.

[22] R. K. Shyamasundar, N. V. N. Kumar, A. Taware, and P. Vyas, "An experimental flow secure file system," in *17th IEEE Int. Conf. On Trust, Security And Privacy In Computing And Comm. (TrustCom)*, 2018, pp. 790–799.

[23] B. S. Radhika, N. V. N. Kumar, and R. K. Shyamasundar, "Flowconseal: Automatic flow consistency analysis of seandroid and selinux policies," in *Data and Applications Security and Privacy XXXII*. Springer, 2018, pp. 219–231.

[24] J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *USENIX Security Symposium*, 2002, pp. 17–31.