Software fault isolation with API integrity and multi-principal modules

Yandong Mao, Haogang Chen, Dong Zhou†, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek
MIT CSAIL, †Tsinghua University IIIS

ABSTRACT

The security of many applications relies on the kernel being secure, but history suggests that kernel vulnerabilities are routinely discovered and exploited. In particular, exploitable vulnerabilities in kernel modules are common. This paper proposes LXFI, a system which isolates kernel modules from the core kernel so that vulnerabilities in kernel modules cannot lead to a privilege escalation attack. To safely give kernel modules access to complex kernel APIs, LXFI introduces the notion of *API integrity*, which captures the set of contracts assumed by an interface. To partition the privileges within a shared module, LXFI introduces *module principals*. Programmers specify principals and API integrity rules through capabilities and annotations. Using a compiler plugin, LXFI instruments the generated code to grant, check, and transfer capabilities between modules, according to the programmer's annotations. An evaluation with Linux shows that the annotations required on kernel functions to support a new module are moderate, and that LXFI is able to prevent three known privilege-escalation vulnerabilities. Stress tests of a network driver module also show that isolating this module using LXFI does not hurt TCP throughput but reduces UDP throughput by 35%, and increases CPU utilization by 2.2–3.7×.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection.

General Terms: Security.

1 INTRODUCTION

Kernel exploits are not as common as Web exploits, but they do happen [2]. For example, for the Linux kernel, a kernel exploit is reported about once per month, and often these exploits attack kernel modules instead of the core kernel [5]. These kernel exploits are devastating because they typically allow the adversary to obtain "root" privilege. For instance, CVE-2010-3904 reports on a vulnerability in Linux's Reliable Datagram Socket (RDS) module that allowed an adversary to write an arbitrary value to an arbitrary kernel address because the RDS page copy function missed a check on a user-supplied pointer. This vulnerability can be exploited to overwrite function pointers and invoke arbitrary kernel or user code. The contribution of this paper is LXFI, a new software fault isolation system to isolate kernel modules. LXFI allows a module developer to partition the privileges held by a single shared module into multiple *principals*, and provides annotations to enforce *API integrity* for complex, irregular kernel interfaces such as the ones found in the Linux kernel and exploited by attackers.

Previous systems such as XFI [9] have used software isolation [26] to isolate kernel modules from the core kernel, thereby protecting against a class of attacks on kernel modules. The challenge is that modules need to use support functions in the core kernel to operate correctly; for example, they need to be able acquire locks, copy data, etc., which require invoking functions in the kernel core for these abstractions. Since the kernel does not provide type safety for pointers, a compromised module can exploit some seemingly "harmless" kernel API to gain privilege. For instance, the `spin_lock_init` function in the kernel writes the value zero to a spinlock that is identified by a pointer argument. A module that can invoke `spin_lock_init` could pass the address of the user ID value in the current process structure as the spinlock pointer, thereby tricking `spin_lock_init` into setting the user ID of the current process to zero (i.e., root in Linux), and gaining root privileges.

Two recent software fault isolation systems, XFI and BGI [4], have two significant shortcomings. First, neither can deal with complex, irregular interfaces; as noted by the authors of XFI, attacks by modules that abuse an over-permissive kernel routine that a module is allowed to invoke remain an open problem [9, §6.1]. BGI tackles this problem in the context of Windows kernel drivers, which have a well-defined regular structure amenable to manual interposition on all kernel/module interactions. The Linux kernel, on the other hand, has a more complex interface that makes manual interposition difficult. For example, Linux kernel interfaces often store function pointers to both kernel and module functions in data structures that are updated by modules, and invoked by the kernel in many locations.

The second shortcoming of XFI and BGI is that they cannot isolate different instances of the same module. For example, a single kernel module might be used to implement many instances of the same abstraction (e.g., many block devices or many sockets). If one of these instances is compromised by an adversary, the adversary also gains access to the privileges of all other instances as well.

This paper's goal is to solve both of these problems for Linux kernel modules. To partition the privileges held by a shared module, LXFI extends software fault isolation to allow modules to have multiple *principals*. Principals correspond to distinct instances of abstractions provided by a kernel module, such as a single socket or a block device provided by a module that can instantiate many of them. Programmers annotate kernel interfaces to specify what principal should be used when the module is invoked, and each principal's privileges are kept separate by LXFI. Thus, if an adversary compromises one instance of the module, the adversary can only misuse that principal's privileges (e.g., being able to modify data on a single socket, or being able to write to a single block device).

To handle complex kernel interfaces, LXFI introduces *API integrity*, which captures the contract assumed by kernel developers for a particular interface. To capture API integrity, LXFI uses capabilities to track the privileges held by each principal, and introduces *light-weight annotations* that programmers use to express the API

SOSP '11, October 23–26, 2011, Cascais, Portugal.

integrity of an interface in terms of capabilities and principals. LXFI enforces API integrity at runtime through software fault isolation techniques.

To test out these ideas, we implemented LXFI for Linux kernel modules. The implementation provides the same basic security properties as XFI and BGI, using similar techniques, but also enforces API integrity for multiple principals. To use LXFI, a programmer must first specify the security policy for an API, using source-level annotations. LXFI enforces the specified security policy with the help of two components. The first is a compile-time rewriter, which inserts checks into kernel code that, when invoked at runtime, verify that security policies are upheld. The second is a runtime component, which maintains the privileges of each module and checks whether a module has the necessary privileges for any given operation at runtime. To enforce API integrity efficiently, LXFI uses a number of optimizations, such as *writer-set tracking*. To isolate a module at runtime, LXFI sets up the initial capabilities, manages the capabilities as they are added and revoked, and checks them on all calls between the module and the core kernel according to the programmer's annotations.

An evaluation for 10 kernel modules shows that supporting a new module requires 8–133 annotations, of which many are shared between modules. Furthermore, the evaluation shows that LXFI can prevent exploits for three CVE-documented vulnerabilities in kernel modules (including the `RDS` module). Stress tests with a network driver module show that isolating this module using LXFI does not hurt TCP throughput, but reduces UDP throughput by 35%, and increases CPU utilization by 2.2–3.7×.

The contributions of the paper are as follows. First, this paper extends the typical module isolation model to support multiple principals per code module. Second, this paper introduces the notion of API integrity, and provides a light-weight annotation language that helps describe the security properties of kernel and module interfaces in terms of capabilities. Finally, this paper demonstrates that LXFI is practical in terms of performance, security, and annotation effort by evaluating it on the Linux kernel.

The rest of the paper is organized as follows. The next section defines the goal of this paper, and the threat model assumed by LXFI. §3 gives the design of LXFI and its annotations. We describe LXFI's compile-time and runtime components in §4 and §5, and discuss how we expect kernel developers to use LXFI in practice in §6. §7 describes the implementation details. We evaluate LXFI in §8, discuss related work in §9, and conclude in §10.

2 GOAL AND PROBLEM

LXFI's goal is to prevent an adversary from exploiting vulnerabilities in kernel modules in a way that leads to a privilege escalation attack. Many exploitable kernel vulnerabilities are found in kernel modules. For example, Chen et al. find that two thirds of kernel vulnerabilities reported by CVE between Jan 2010 and March 2011 are in kernel modules [1, 5].

When adversaries exploit bugs in the kernel, they trick the kernel code into performing operations that the code would not normally do. For example, an adversary can trick the kernel into writing to arbitrary memory locations, or invoking arbitrary kernel code. Adversaries can leverage this to gain additional privileges (e.g., by running their own code in the kernel, or overwriting the user ID of the current process), or to disclose data from the system. The focus of LXFI is on preventing integrity attacks (e.g., privilege escalation), and not on data disclosure.

In LXFI, we assume that we will not be able to fix all possible underlying software bugs, but instead we focus on reducing the possible operations the adversary can trick the kernel into performing to the set of operations that code (e.g., a kernel module) would ordinarily be able to do. For example, if a module does not ordinarily modify the user ID field in the process structure, LXFI should prevent the module from doing so even if it is compromised by an adversary. Similarly, if a module does not ordinarily invoke kernel functions to write blocks to a disk, LXFI should prevent a module from doing so, even if it is compromised.

LXFI's approach to prevent privilege escalation is to isolate the modules from each other and from the core of the kernel, as described above. Of course, a module may legitimately need to raise the privileges of the current process, such as through `setuid` bits in a file system, so this approach will not prevent all possible privilege escalation exploits. However, most of the exploits found in practice take advantage of the fact that every buggy module is fully privileged, and making modules less privileged will reduce the number of possible exploits.

Another challenge in making module isolation work lies in knowing what policy rules to enforce at module boundaries. Since the Linux kernel was written without module isolation in mind, all such rules are implicit, and can only be determined by manual code inspection. One possible solution would be to re-design the Linux kernel to be more amenable to privilege separation, and to have simpler interfaces where all privileges are explicit; however, doing this would involve a significant amount of work. LXFI takes a different approach that tries to make as few modifications to the Linux kernel as possible. To this end, LXFI, like previous module isolation systems [4, 9, 26], relies on developers to specify this policy.

In the rest of this section, we will discuss two specific challenges that have not been addressed in prior work that LXFI solves, followed by the assumptions made by LXFI. Briefly, the challenges have to do with a shared module that has many privileges on behalf of its many clients, and with concisely specifying module policies for complex kernel interfaces like the ones in the Linux kernel.

2.1 Privileges in shared modules

The first challenge is that a single kernel module may have many privileges if that kernel module is being used in many contexts. For example, a system may use the `dm-crypt` module to manage encrypted block devices, including both the system's main disk and any USB flash drives that may be connected by the user. The entire `dm-crypt` module must have privileges to write to all of these devices. However, if the user accidentally plugs in a malicious USB flash drive that exploits a bug in `dm-crypt`, the compromised `dm-crypt` module will be able to corrupt all of the block devices it manages. Similarly, a network protocol module, such as `econet`, must have privileges to write to all of the sockets managed by that module. As a result, if an adversary exploits a vulnerability in the context of his or her `econet` connection, the adversary will be able to modify the data sent over any other `econet` socket as well.

2.2 Lack of API integrity

The second challenge is that kernel modules use complex kernel interfaces. These kernel interfaces could be mis-used by a compromised module to gain additional privileges (e.g., by corrupting memory). One approach is to re-design kernel interfaces to make it easy to enforce safety properties, such as in Windows, as illustrated by BGI [4]. However, LXFI's goal is to isolate existing Linux kernel modules, where many of the existing interfaces are complex.

To prevent these kinds of attacks in Linux, we define *API integrity* as the contract that developers intend for any module to follow when using some interface, such as the memory allocator, the PCI subsystem, or the network stack. The set of rules that make up the contract between a kernel module and the core kernel are different

```
1  struct pci_driver {
2    int (*probe) (struct pci_dev *pcidev);
3  };
4
5  struct net_device {
6    struct net_device_ops *dev_ops;
7  };
8
9  struct net_device_ops {
10   netdev_tx_t (*ndo_start_xmit)
11     (struct sk_buff *skb,
12      struct net_device *dev);
13 };
14
15 /* Exported kernel functions */
16 void pci_enable_device(struct pci_dev *pcidev);
17 void netif_rx(struct sk_buff *skb);
18
19 /* In core kernel code */
20 module_driver->probe(pcidev);
21
22 void
23 netif_napi_add(struct net_device *dev,
24       struct napi_struct *napi,
25       int (*poll) (struct napi_struct *, int))
26 {
27   dev->dev_ops->ndo_start_xmit(skb, ndev);
28   (*poll) (napi, 5);
29 }
30
31 /* In network device driver's module */
32 int
33 module_pci_probe(struct pci_dev *pcidev) {
34   ndev = alloc_etherdev(...);
35   pci_enable_device(pcidev);
36   ndev->dev_ops->ndo_start_xmit = myxmit;
37   netif_napi_add(ndev, napi, my_poll_cb);
38   return 0;
39 }
40
41 /* In network device driver's code */
42 netif_rx(skb);
```

Figure 1: Parts of a PCI network device driver in Linux.

for each kernel API, and the resulting operations that the kernel module can perform are also API-specific. However, by enforcing API integrity—i.e., ensuring that each kernel module follows the intended contract for core kernel interfaces that it uses—LXFI will ensure that a compromised kernel module cannot take advantage of the core kernel's interfaces to perform more operations than the API was intended to allow.

To understand the kinds of contracts that an interface may require, consider a PCI network device driver for the Linux kernel, shown in Figure 1. In the rest of this section, we will present several examples of contracts that make up API integrity for this interface, and how a kernel module may violate those contracts.

Memory safety and control flow integrity. Two basic safety properties that all software fault isolation systems enforce is *memory safety*, which guarantees that a module can only access memory that it owns or has legitimate access to, and *control flow integrity*, which guarantees that a module can only execute its own isolated code and external functions that it has legitimate access to. However, memory safety and control flow integrity are insufficient to provide API integrity, and the rest of this section describes other safety properties enforced by LXFI.

Function call integrity. The first aspect of API integrity deals with *how* a kernel module may invoke the core kernel's functions. These contracts are typically concerned with the arguments that the module can provide, and the specific functions that can be invoked, as we will now illustrate.

Many function call contracts involve the notion of object ownership. For example, when the network device driver module in Figure 1 calls `pci_enable_device` to enable the PCI device on line 35, it is expected that the module will provide a pointer to its own `pci_dev` structure as the argument (i.e., the one it received as an argument to `module_pci_probe`). If the module passes in some other `pci_dev` structure to `pci_enable_device`, it may be able to interfere with other devices, and potentially cause problems for other modules. Furthermore, if the module is able to construct its own `pci_dev` structure, and pass it as an argument, it may be able to trick the kernel into performing arbitrary device I/O or memory writes.

A common type of object ownership is write access to memory. Many core kernel functions write to a memory address supplied by the caller, such as `spin_lock_init` from the example in §1. In these cases, a kernel module should only be able to pass addresses of kernel memory it has write access to (for a sufficient number of bytes) to such functions; otherwise, a kernel module may trick the kernel into writing to arbitrary kernel memory. On the other hand, a kernel module can also have ownership of an object without being able to write to its memory: in the case of the network device, modules should not directly modify the memory contents of their `pci_dev` struct, since it would allow the module to trick the kernel into controlling a different device, or dereferencing arbitrary pointers.

Another type of function call contract relates to callback functions. Several kernel interfaces involve passing around callback functions, such as the `netif_napi_add` interface on line 23. In this case, the kernel invokes the `poll` function pointer at a later time, and expects that this points to a legitimate function. If the module is able to provide arbitrary function pointers, such as `my_poll_cb` on line 37, the module may be able to trick the kernel into running arbitrary code when it invokes the callback on line 28. Moreover, the module should be able to provide only pointers to functions that the module itself can invoke; otherwise, it can trick the kernel into running a function that it is not allowed to call directly.

Function callbacks are also used in the other direction: for modules to call back into the core kernel. Once the core kernel has provided a callback function a kernel module, the module is expected to invoke the callback, probably with a prescribed callback argument. The module should not invoke the callback function before the callback is provided, or with a different callback argument.

Data structure integrity. In addition to memory safety, many kernel interfaces assume that the actual *data* stored in a particular memory location obeys certain invariants. For example, an `sk_buff` structure, representing a network packet in the Linux kernel, contains a pointer to packet data. When the module passes an `sk_buff` structure to the core kernel on line 42, it is expected to provide a legitimate data pointer inside of the `sk_buff`, and that pointer should point to memory that the kernel module has write access to (in cases when the `sk_buff`'s payload is going to be modified). If this invariant is violated, the kernel code can be tricked into writing to arbitrary memory.

Another kind of data structure integrity deals with function pointers that are stored in shared memory. The Linux kernel often stores callback function pointers in data structures. For example, the core kernel invokes a function pointer from `dev->dev_ops` on line 27. The implicit assumption the kernel is making is that the function pointer points to legitimate code that should be executed. However,

if the kernel module was able to write arbitrary values to the function pointer field, it could trick the core kernel into executing arbitrary code. Thus, in LXFI, even though the module can write a legitimate pointer on line 36, it should not be able to corrupt it later. To address this problem, LXFI checks whether the function pointer value that is about to be invoked was a legitimate function address that the pointer's writer was allowed to invoke too.

API integrity in Linux. In the general case, it is difficult to find or enumerate all of the contracts necessary for API integrity. However, in our experience, kernel module interfaces in Linux tend to be reasonably well-structured, and it is possible to capture the contracts of many interfaces in a succinct manner. Even though these interfaces are not used as security boundaries in the Linux kernel, they are carefully designed by kernel developers to support a range of kernel modules, and contain many sanity checks to catch buggy behavior by modules (e.g., calls to `BUG()`).

LXFI relies on developers to provide annotations capturing the API integrity of each interface. LXFI provides a safe default, in that a kernel function with no annotations (e.g., one that the developer forgot to annotate) cannot be accessed by a kernel module. However, LXFI trusts any annotations that the developer provides; if there is any mistake or omission in an annotation, LXFI will enforce the policy specified in the annotation, and not the intended policy. Finally, in cases when it is difficult to enforce API integrity using LXFI, re-designing the interface to fit LXFI's annotations may be necessary (however, we have not encountered any such cases for the modules we have annotated).

2.3 Threat model

LXFI makes two assumptions to isolate kernel modules. First, LXFI assumes that the core kernel, the annotations on the kernel's interfaces, and the LXFI verifier itself are correct.

Second, LXFI infers the initial privileges that a module should be granted based on the functions that module's code imports. Thus, we trust that the programmer of each kernel module only invokes functions needed by that module. We believe this is an appropriate assumption because kernel developers are largely well-meaning, and do not try to access unnecessary interfaces on purpose. Thus, by capturing the intended privileges of a module, and by looking at the interfaces required in the source code, we can prevent an adversary from accessing any additional interfaces at runtime.

3 ANNOTATIONS

At a high level, LXFI's workflow consists of four steps. First, kernel developers annotate core kernel interfaces to enforce API integrity between the core kernel and modules. Second, module developers annotate certain parts of their module where they need to switch privileges between different module principals. Third, LXFI's compile-time rewriter instruments the generated code to perform API integrity checks at runtime. Finally, LXFI's runtime is invoked at these instrumented points, and performs the necessary checks to uphold API integrity. If the checks fail, the kernel panics. The rest of this section describes LXFI's principals, privileges, and annotations.

3.1 Principals

Many modules provide an abstraction that can be instantiated many times. For example, the `econet` protocol module provides an `econet` socket abstraction that can be instantiated to create a specific socket. Similarly, device mapper modules such as `dm-crypt` and `dmraid` provide a layered block device abstraction that can be instantiated for a particular block device.

To minimize the privileges that an adversary can take advantage of when they exploit a vulnerability in a module, LXFI logically breaks up a module into multiple *principals* corresponding to each instance of the module's abstraction. For example, each `econet` socket corresponds to a separate module principal, and each block device provided by `dm-crypt` also corresponds to a separate module principal. Each module principal will have access to only the privileges needed by that instance of the module's abstraction, and not to the global privileges of the entire module.

To support this plan, LXFI provides three mechanisms. First, LXFI allows programmers to define principals in a module. To avoid requiring existing code to keep track of LXFI-specific principals, LXFI names module principals based on existing data structures used to represent an instance of the module's abstraction. For example, in `econet`, LXFI uses the address of the socket structure as the principal name. Similarly, in device mapper modules, LXFI uses the address of the block device.

Second, LXFI allows programmers to define what principal should be used when invoking a module, by providing annotations on function types (which we discuss more concretely in §3.3). For example, when the kernel invokes the `econet` module to send a message over a socket, LXFI should execute the module's code in the context of that socket's principal. To achieve this behavior, the programmer annotates the message send function to specify that the socket pointer argument specifies the principal name. At runtime, LXFI uses this annotation to switch the current principal to the one specified by the function's arguments when the function is invoked. These principal identifiers are stored on a shadow stack, so that if an interrupt comes in while a module is executing, the module's privileges are saved before handling the interrupt, and restored on interrupt exit.

Third, a module may share some state between multiple instances. For example, the `econet` module maintains a linked list of all sockets managed by that module. Since each linked list pointer is stored in a different socket object, no single instance principal is able to add or remove elements from this list. Performing these cross-instance operations requires global privileges of the entire module. In these cases, LXFI allows programmers to switch the current principal to the module's *global* principal, which implicitly has access to the capabilities of all other principals in that module. For example, in `econet`, the programmer would modify the function used to add or remove sockets from this linked list to switch to running as the global principal. Conversely, a *shared* principal is used to represent privileges accessible to all principals in a module, such as the privileges to invoke the initial kernel functions required by that module. All principals in a module implicitly have access to all of the privileges of the shared principal.

To ensure that a function that switches to the global principal cannot be tricked into misusing its global privileges, programmers must insert appropriate checks before every such privilege change. LXFI's control flow integrity then ensures that these checks cannot be bypassed by an adversary at runtime. A similar requirement arises for other privileged LXFI functions, such as manipulating principals. We give an example of such checks in §3.4.

3.2 Capabilities

Modules do not explicitly define the privileges they require at runtime—such as what memory they may write, or what functions they may call—and even for functions that a module may legitimately need, the function itself may be expecting the module to invoke it in certain ways, as described in §2.2 and Figure 1.

To keep track of module privileges, LXFI maintains a set of capabilities, similar to BGI, that track the privileges of each module principal at runtime. LXFI supports three types of capabilities, as follows:

$$
\begin{aligned}
\textit{annotation} &::= \textbf{pre}(\textit{action}) \mid \textbf{post}(\textit{action}) \mid \textbf{principal}(\textit{c-expr}) \\
\textit{action} &::= \textbf{copy}(\textit{caplist}) \\
&\mid \textbf{transfer}(\textit{caplist}) \\
&\mid \textbf{check}(\textit{caplist}) \\
&\mid \textbf{if}\ (\textit{c-expr})\ \textit{action} \\
\textit{caplist} &::= (c, \textit{ptr}, [\textit{size}]) \\
&\mid \textit{iterator-func}(\textit{c-expr})
\end{aligned}
$$

Figure 2: Grammar for LXFI annotations. A *c-expr* corresponds to a C expression that can reference the annotated function's arguments and its return value. An *iterator-func* is a name of a programmer-supplied C function that takes a *c-expr* argument, and iterates over a set of capabilities. c specifies the type of the capability (either WRITE, CALL, or REF, as described in §3.2), and *ptr* is the address or argument for the capability. The *size* parameter is optional, and defaults to `sizeof`(`*`*ptr*).

WRITE $(ptr, size)$**.** This capability means that a module can write any values to memory region $[ptr, ptr + size)$ in the kernel address space. It can also pass addresses inside the region to kernel routines that require writable memory. For example, the network device module in Figure 1 would have a WRITE capability for its `sk_buff` packets and their payloads, which allows it to modify the packet.

REF (t, a)**.** This capability allows the module to pass a as an argument to kernel functions that require a capability of REF type t, capturing the object ownership idea from §2. Type t is often the C type of the argument, although it need not be the case, and we describe situations in which this happens in §6. Unlike the WRITE capability, REF (t, a) does not grant write access to memory at address a. For instance, in our network module, the module should receive a REF (`pci_dev`, `pcidev`) capability when the core kernel invokes `module_driver->probe` on line 20, if that code was annotated to support LXFI capabilities. This capability would then allow the module to call `pci_enable_device` on line 35.

CALL (a)**.** The module can call or jump to a target memory address a. In our network module example, the module has a CALL capability for `netif_rx`, `pci_enable_device`, and others; this particular example has no instances of dynamic call capabilities provided to the module by the core kernel at runtime.

The basic operations on capabilities are granting a capability, revoking all copies of a capability, and checking whether a caller has a capability. To set up the basic execution environment for a module, LXFI grants a module initial capabilities when the module is loaded, which include: (1) a WRITE capability to its writable data section; (2) a WRITE capability to the current kernel stack (does not include the shadow stack, which we describe later); and (3) CALL capabilities to all kernel routines that are imported in the module's symbol table.

A module can gain or lose additional capabilities when it calls support functions in the core kernel. For example, after a module calls `kmalloc`, it gains a WRITE capability to the newly allocated memory. Similarly, after calling `kfree`, LXFI's runtime revokes the corresponding WRITE capability from that module.

3.3 Interface annotations

Although the principal and capability mechanisms allow LXFI to reason about the privileges held by each module principal, it is cumbersome for the programmer to manually insert calls to switch principals, transfer capabilities, and verify whether a module has a certain capability, for each kernel/module API function (as in BGI [4]). To simplify the programmer's job, LXFI allows programmers to annotate interfaces (i.e., prototype definitions in C) with principals and capability actions. LXFI leverages the `clang` support for attributes to specify the annotations.

LXFI annotations are consulted when invoking a function, and can be associated (in the source code) with function declarations, function definitions, or function pointer types. A single kernel function (or function pointer type) can have multiple LXFI annotations; each one describes what action the LXFI runtime should take, and specifies whether that action should be taken before the function is called, or after the call finishes, as indicated by **pre** and **post** keywords. Figure 2 summarizes the grammar for LXFI's annotations.

There are three types of annotations supported by LXFI: **pre**, **post**, and **principal**. The first two perform a specified action either before invoking the function or after it returns. The **principal** annotation specifies the name of the module principal that should be used to execute the called function, which we discuss shortly.

There are four actions that can be performed by either **pre** or **post** annotations. A **copy** action grants a capability from the caller to the callee for **pre** annotations (and vice-versa for **post**). A **transfer** action moves ownership of a capability from the caller to the callee for **pre** annotations (and vice-versa for **post**). Both **copy** and **transfer** ensure that the capability is owned in the first place before granting it. A **check** action verifies that the caller owns a particular capability; all **check** annotations are **pre**. To support conditional annotations, LXFI supports **if** actions, which conditionally perform some action (such as a **copy** or a **transfer**) based on an expression that can involve either the function's arguments, or, for **post** annotations, the function's return value. For example, this allows transferring capabilities for a memory buffer only if the return value does not indicate an error.

Transfer actions revoke the transferred capability from *all* principals in the system, rather than just from the immediate source of the transfer. (As described above, transfers happen in different directions depending on whether the action happens in a **pre** or **post** context.) Revoking a capability from all principals ensures that no copies of the capability remain, and allows the object referred to by the capability to be re-used safely. For example, the memory allocator's `kfree` function uses **transfer** to ensure no outstanding capabilities exist for free memory. Similarly, when a network driver hands a packet to the core kernel, a **transfer** action ensures the driver—and any other module the driver could have given capabilities to—cannot modify the packet any more.

The **copy**, **transfer**, and **check** actions take as argument the list of capabilities to which the action should be applied. In the simple case, the capability can be specified inline, but the programmer can also implement their own function that returns a list of capabilities, and use that function in an action to iterate over all of the returned capabilities. Figure 3 provides several example LXFI annotations and their semantics.

To specify the principal with whose privilege the function should be invoked, LXFI provides a **principal** annotation. LXFI's principals are named by arbitrary pointers. This is convenient because Linux kernel interfaces often have an object corresponding to every instance of an abstraction that a principal tries to capture. For example, a network device driver would use the address of its `net_device` structure as the principal name to separate different network interfaces from each other. Adding explicit names for principals would require extending existing Linux data structures to store this additional name, which would require making changes to the Linux kernel, and potentially break data structure invariants, such as alignment or layout.

Annotation	Semantics
pre(**copy**(c,*ptr*,[*size*]))	Check that *caller* owns capability c for $[ptr, ptr+size)$ before calling function. Copy capability c from *caller* to *callee* for $[ptr, ptr+size)$ before the call.
post(**copy**(c,*ptr*,[*size*]))	Check that *callee* owns capability c for $[ptr, ptr+size)$ after the call. Copy capability c from *callee* to *caller* for $[ptr, ptr+size)$ after the call.
pre(**transfer**(c,*ptr*,[*size*]))	Check that *caller* owns capability c for $[ptr, ptr+size)$ before calling function. Transfer capability c from *caller* to *callee* for $[ptr, ptr+size)$ before the call.
post(**transfer**(c,*ptr*,[*size*]))	Check that *callee* owns capability c for $[ptr, ptr+size)$ after the call. Transfer capability c from *callee* to *caller* for $[ptr, ptr+size)$ after the call.
pre(**check**(c,*ptr*,[*size*]))	Check that the caller has the (c,*ptr*,[*size*]) capability.
pre(**check**(`skb_iter`(*ptr*)))	Check that the caller has all capabilities returned by the programmer-supplied `skb_iter` function.
pre(**if** (*c-expr*) *action*) **post**(**if** (*c-expr*) *action*)	Run the specified action if the expression *c-expr* is true; used for conditional annotations based on return value. LXFI allows *c-expr* to refer to function arguments, and (for **post** annotations) to the return value.
principal(p)	Use p as the *callee* principal; in the absence of this annotation, LXFI uses the module's shared principal.

Figure 3: Examples of LXFI annotations, using the grammar shown in Figure 2, and their semantics.

One complication with LXFI's pointer-based principal naming scheme is that a single instance of an module's abstraction may have a separate data structure that is used for different interfaces. For instance, a PCI network device driver may be invoked both by the network sub-system and by the PCI sub-system. The network sub-system would use the pointer of the `net_device` structure as the principal name, and the PCI sub-system would use the pointer of the `pci_dev` structure for the principal. Even though these two names may refer to the same logical principal (i.e., a single physical network card), the names differ.

To address this problem, LXFI separates principals from their names. This allows a single logical principal to have multiple names, and LXFI provides a function called `lxfi_princ_alias` that a module can use to map names to principals. The special values *global* and *shared* can be used as an argument to a **principal** annotation to indicate the module's global and shared principals, respectively. For example, this can be used for functions that require access to the entire module's privileges, such as adding or removing sockets from a global linked list in `econet`.

3.4 Annotation example

To give a concrete example of how LXFI's annotations are used, consider the interfaces shown in Figure 1, and their annotated version in Figure 4. LXFI's annotations are underlined in Figure 4. Although this example involves a significant number of annotations, we specifically chose it to illustrate most of LXFI's mechanisms.

To prevent modules from arbitrarily enabling PCI devices, the `pci_enable_device` function on line 67 in Figure 4 has a **check** annotation that ensures the caller has a REF capability for the corresponding `pci_dev` object. When the module is first initialized for a particular PCI device, the `probe` function grants it such a REF capability (based on the annotation for the `probe` function pointer on line 45). Note that if the `probe` function returns an error code, the **post** annotation on the `probe` function transfers ownership of the `pci_dev` object back to the caller.

Once the network interface is registered with the kernel, the kernel can send packets by invoking the `ndo_start_xmit` function. The annotations on this function, on line 60, grant the module access to the packet, represented by the `sk_buff` structure. Note that the `sk_buff` structure is a complicated object, including a pointer to a separate region of memory holding the actual packet payload. To compute the set of capabilities needed by an `sk_buff`, the programmer writes a capability iterator called `skb_caps` that invokes LXFI's `lxfi_cap_iterate` function on all of the capabilities that make up the `sk_buff`. This function in turn performs the requested operation (**transfer**, in this case) based on the context in which the capability iterator was invoked. As with the PCI example above, the annotations transfer the granted capabilities back to the caller in case of an error.

Note that, when the kernel invokes the device driver through `ndo_start_xmit`, it uses the pointer to the `net_dev` structure as the principal name (line 60), even though the initial PCI `probe` function used the `pci_dev` structure's address as the principal (line 45). To ensure that the module has access to the same set of capabilities in both cases, the module developer must create two names for the corresponding logical principal, one using the `pci_dev` object, and one using the `net_device` object.

To do this, the programmer modifies the module's code as shown in lines 72–73. This code creates a new name, `ndev`, for an existing principal with the name `pcidev` on line 73. The check on line 72 ensures that this code will only execute if the current principal already has privileges for the `pcidev` object. This ensures that an adversary cannot call the `module_pci_probe` function with some other `pcidev` object and trick the code into setting up arbitrary aliases to principals. LXFI's control flow integrity ensures that an adversary is not able to transfer control flow directly to line 73. Moreover, only direct control flow transfers to `lxfi_princ_alias` are allowed. This ensures that an adversary cannot invoke this function by constructing and calling a function pointer at runtime; only statically defined calls, which are statically coupled with a preceding check, are allowed.

4 COMPILE-TIME REWRITING

When compiling the core kernel and modules, LXFI uses compiler plugins to inserts calls and checks into the generated code so that the LXFI runtime can enforce the annotations for API integrity and principals. LXFI performs different rewriting for the core kernel and for modules. Since LXFI assumes that the core kernel is fully trusted, it can omit most checks for performance. Modules are not fully trusted, and LXFI must perform more extensive rewriting there.

4.1 Rewriting the core kernel

The only rewriting that LXFI must perform on core kernel code deals with invocation of function pointers that may have been supplied by a module. If a module is able to supply a function pointer that the core kernel will invoke, the module can potentially increase its privileges, if it tricks the kernel into performing a call that the module itself could not have performed directly. To ensure this is not the case, LXFI performs two checks. First, prior to invoking a function pointer from the core kernel, LXFI verifies that the module principal that supplied the pointer (if any) had the appropriate CALL capability for that function. Second, LXFI ensures that the annotations for the function supplied by the module and the function pointer type match. This ensures that a module cannot change the effective

```
43 struct pci_driver {
44   int (*probe) (struct pci_dev *pcidev, ...)
45     principal(pcidev)
46     pre(copy(ref(struct pci_dev), pcidev))
47     post(if (return < 0)
48            transfer(ref(struct pci_dev), pcidev));
49 };
50
51 void skb_caps(struct sk_buff *skb) {
52   lxfi_cap_iterate(write, skb, sizeof(*skb));
53   lxfi_cap_iterate(write, skb->data, skb->len);
54 }
55
56 struct net_device_ops {
57   netdev_tx_t (*ndo_start_xmit)
58     (struct sk_buff *skb,
59      struct net_device *dev)
60     principal(dev)
61     pre(transfer(skb_caps(skb)))
62     post(if (return == -NETDEV_BUSY)
63            transfer(skb_caps(skb)))
64 };
65
66 void pci_enable_device(struct pci_dev *pcidev)
67   pre(check(ref(struct pci_dev), pcidev));
68
69 int
70 module_pci_probe(struct pci_dev *pcidev) {
71   ndev = alloc_etherdev(...);
72   lxfi_check(ref(struct pci_dev), pcidev);
73   lxfi_princ_alias(pcidev, ndev);
74   pci_enable_device(pcidev);
75   ndev->dev_ops->ndo_start_xmit = myxmit;
76   netif_napi_add(ndev, napi, my_poll_cb);
77   return 0;
78 }
```

Figure 4: Annotations for parts of the API shown in Figure 1. The annotations follow the grammar shown in Figure 2. Annotations and added code are underlined.

```
79 handler_func_t handler;
80 handler = device->ops->handler;
81 lxfi_check_indcall(&device->ops->handler);
82                    /* not &handler */
83 handler(device);
```

Figure 5: Rewriting an indirect call in the core kernel. LXFI inserts checking code with the address of a module-supplied function pointer.

annotations on a function by storing it in a function pointer with different annotations.

To implement this check, LXFI's kernel rewriter inserts a call to the checking function `lxfi_check_indcall(void **pptr, unsigned ahash)` before every indirect call in the core kernel, where `pptr` is the address of the module-supplied function pointer to be called, and `ahash` is the hash of the annotation for the function pointer type. The LXFI runtime will validate that the module that writes function f to `pptr` has a CALL capability for f. To ensure that annotations match, LXFI compares the hash of the annotations for both the function and the function pointer type.

To optimize the cost of these checks, LXFI implements writer-set tracking. The runtime tracks the set of principals that have been granted a WRITE capability for each memory location after the last time that memory location was zeroed. Then, for each indirect-call check in the core kernel, the LXFI runtime first checks whether any principal could have written to the function pointer about to be invoked. If not, the runtime can bypass the relatively expensive capability check for the function pointer.

To detect the original memory location from which the function pointer was obtained, LXFI performs a simple intra-procedural analysis to trace back the original function pointer. For example, as shown in Figure 5, the core kernel may copy a module-supplied function pointer `device->ops->handler` to a local variable `handler`, and then make a call using the local variable. In this case LXFI uses the address of the original function pointer rather than the local variable for looking up the set of writer principals. We have encountered 51 cases that our simple analysis cannot deal with, out of 7500 indirect call sites in the core kernel, in which the value of the called pointer originates from another function. We manually verify that these 51 cases are safe.

4.2 Rewriting modules

LXFI inserts calls to the runtime when compiling modules based on annotations from the kernel and module developers. The rest of this subsection describes the types of instrumentation that LXFI performs for module C code.

Annotation propagation. To determine the annotations that should apply to a function, LXFI first propagates annotations on a function pointer type to the actual function that might instantiate that type. Consider the structure member `probe` in Figure 4, which is a function pointer initialized to the `module_pci_probe` function. The function should get the annotations on the `probe` member. LXFI propagates these annotations along initializations, assignments, and argument passing in the module's code, and computes the annotation set for each function. A function can obtain different annotations from multiple sources. LXFI verifies that these annotations are exactly the same.

Function wrappers. At compile time, LXFI generates wrappers for each module-defined function, kernel-exported function, and indirect call site in the module. At runtime, when the kernel calls into one of the module's functions, or when the module calls a kernel-exported function, the corresponding function wrapper is invoked first. Based on the annotations, the wrapper sets the appropriate principal, calls the actions specified in `pre` annotations, invokes the original function, and finally calls the actions specified in `post` annotations.

The function wrapper also invokes the LXFI runtime at its entry and exit, so that the runtime can capture all control flow transitions between the core kernel and the modules. The relevant runtime routines switch principals and enforce control flow integrity using a shadow stack, as we detail in the next section (§5).

Module initialization. For each module, LXFI generates an initialization function that is invoked (without LXFI's isolation) when the module is first loaded, to grant an initial set of capabilities to the module. For each external function (except those functions defined in LXFI runtime) imported in the module's symbol table, the initialization function grants a CALL capability for the corresponding function wrapper. Note that the CALL capabilities granted to the module are only for invoking wrappers. A module is not allowed to call any external functions directly, since that would bypass the annotations on those functions. For each external data symbol in the module's symbol table, the initialization function likewise grants a WRITE capability. The initial capabilities are granted to the module's shared principal, so that they are accessible to every other principal in the module.

Memory writes. LXFI inserts checking code before each memory write instruction to make sure that the current principal has the WRITE capability for the memory region being written to.

5 RUNTIME ENFORCEMENT

To enforce the specified API integrity, the LXFI runtime must track capabilities and ensure that the necessary capability actions are performed on kernel/module boundaries. For example, before a module invokes any kernel functions, the LXFI runtime validates whether the module has the privilege (i.e., CALL capability) to invoke the function at that address, and if the arguments passed by the module are safe to make the call (i.e., the **pre** annotations allow it). Similarly, before the kernel invokes any function pointer that was supplied by a module, the LXFI runtime verifies that the module had the privileges to invoke that function in the first place, and that the annotations of the function pointer and the invoked function match. These checks are necessary since the kernel is, in effect, making the call on behalf of the module.

Figure 6 shows the design of the LXFI runtime. As the reference monitor of the system, it is invoked on all control flow transitions between the core kernel and the modules (at instrumentation points described in the previous section). The rest of this section describes the operations performed by the runtime.

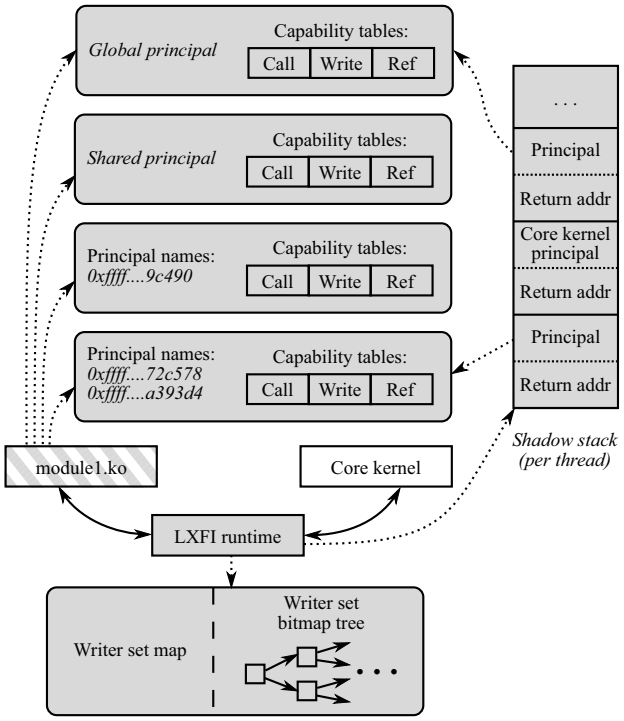

Figure 6: An overview of the LXFI runtime. Shaded components are parts of LXFI. Striped components indicate isolated kernel modules. Solid arrows indicate control flow; the LXFI runtime interposes on all control flow transfers between the modules and the core kernel. Dotted arrows indicate metadata tracked by LXFI.

Principals. The LXFI runtime keeps track of the principals for each kernel module, as well as two special principals. The first is the module's shared principal, which is initialized with appropriate initial capabilities (based on the imports from the module's symbol table); every other principal in the module implicitly has access to the capabilities stored in this principal. The second is the module's global principal; it implicitly has access to all capabilities in all of the module's principals.

Capability table. For each principal, LXFI maintains three capability tables (one per capability type), as shown in Figure 6. Efficiently managing capability tables is important to LXFI's performance. LXFI uses a hash table for each table to achieve constant lookup time. For CALL capabilities and REF capabilities, LXFI uses function addresses and referred addresses, respectively, as the hash keys.

WRITE capabilities do not naturally fit within a hash table, because they are identified by an address range, and capability checks can happen for any address within the range. To support fast range tests, LXFI inserts a WRITE capability into all possible hash table slots covered by its address range. LXFI reduces the number of insertions by masking the least significant bits of the address (the last 12 bits in practice) when calculating hash keys. Since kernel modules do not usually manipulate memory objects larger than a page (2^{12} bytes), in our experience this data structure performs much better than a balancing tree, in which a lookup—commonly performed on WRITE capabilities—takes logarithmic time.

Shadow stack. LXFI maintains a shadow stack for each kernel thread to record LXFI-specific context. The shadow stack lies adjacent to the thread's kernel stack in the virtual address space, but is only accessible to the LXFI runtime. It is updated at the entry and the exit of each function wrapper.

To enforce control flow integrity on function returns, the LXFI runtime pushes the return address onto the shadow stack at the wrapper's entry, and validate its value at the exit to make sure that the return address is not corrupted. The runtime also saves and restores the principal on the shadow stack at the wrapper's entry and exit.

Writer-set tracking. To optimize the cost of indirect call checks, LXFI implements light-weight writer-set tracking (as described in §4.1). LXFI keeps writer set information in a data structure similar to a page table. The last level entries are bitmaps representing whether the writer set for a segment of memory is empty or not. Checking whether the writer set for a particular address is empty takes constant time. The actual contents of non-empty writer sets (i.e., what principal has WRITE access to a range of memory) is computed by traversing a global list of principals. Our experiments to date have involved a small number of distinct principals, leading to acceptable performance.

When a module is loaded, that module's shared principal is added to the writer set for all of its writable sections (including `.data` and `.bss`), because the section may contain writable function pointers that the core kernel may try to invoke. The runtime adds additional entries to the writer set map as the module executes and gains additional capabilities.

LXFI's writer-set tracking introduces both false positives and false negatives. A false positive arises when a WRITE capability of a function pointer was granted to some module's principal, but the principal did not write to the function pointer. This is benign, since it only introduces an unnecessary capability check. A false negative arises when the kernel copies pointers from a location that was modified by a module into its internal data structures, which were not directly modified by a module. At compile time, LXFI detects these cases and we manually inspect such false negatives (see §4.1).

6 USING LXFI

The most important step in enforcing API integrity is specifying the annotations on kernel/module boundaries. If a programmer annotates APIs incorrectly, then an adversary may be able to exploit the mistake to obtain increased privilege. We summarize guidelines for enforcing API integrity based on our experience annotating 10 modules.

Guideline 1. Following the principle of least privilege, grant a REF capability instead of a WRITE capability whenever possible. This ensures that a module will be unable to modify the memory contents of an object, unless absolutely necessary.

Guideline 2. For memory regions allocated by a module, grant WRITE capabilities to the module, and revoke it from the module on free. WRITE is needed because the module usually directly writes the memory it allocates (e.g., for initialization).

Guideline 3. If the module is required to pass a certain fixed value into a kernel API (e.g., an argument to a callback function, or an integer I/O port number to `inb` and `outb` I/O functions), grant a REF capability for that fixed value with a special type, and annotate the function in question (e.g., the callback function, or `inb` and `outb`) to require a REF capability of that special type for its argument.

Guideline 4. When dealing with large data structures, where the module only needs write access to a small number of the structure's members, modify the kernel API to provide stronger API integrity. For example, the `e1000` network driver module writes to only five (out of 51) fields of `sk_buff` structure. This design requires LXFI to grant the module a WRITE capability for the `sk_buff` structure. It would be safer to have the kernel provide functions to change the necessary fields in an `sk_buff`. Then LXFI could grant the module a REF capability, perhaps with a special type of `sk_buff__fields`, and have the annotation on the corresponding kernel functions require a REF capability of type `sk_buff__fields`.

Guideline 5. To isolate instances of a module from each other, annotate the corresponding interface with `principal` annotations. The pointer used as the principal name is typically the main data structure associated with the abstraction, such as a socket, block device, network interface, etc.

Guideline 6. To manipulate privileges inside of a module, make two types of changes to the module's code. First, in order to manipulate data shared between instances, insert a call to LXFI to switch to the module's global principal. Second, in order to create principal aliases, insert a similar call to LXFI's runtime. In both cases, the module developer needs to preface these privileged operations with adequate checks to ensure that the functions containing these privileged operations are not abused by an adversary at runtime.

Guideline 7. When APIs implicitly transfer privileges between the core kernel and modules, explicitly add calls from the core kernel to the module to grant the necessary capabilities. For example, the Linux network stack supports packet schedulers, represented by a `struct Qdisc` object. When the kernel wants to assign a packet scheduler to a network interface, it simply changes a pointer in the network interface's `struct net_device` to point to the `Qdisc` object, and expect the module to access it.

7 IMPLEMENTATION

We implemented LXFI for Linux 2.6.36 running on a single-core *x*86_64 system. Figure 7 shows the components and the lines of code for each component. The kernel is compiled using gcc, invoking the kernel rewriting plugin (the kernel rewriter). Modules are compiled using Clang with the module rewriting plugin (the module rewriter), since Clang provides a more powerful infrastructure to implement rewriting. The current implementation of LXFI has several limitations, as follows.

The LXFI rewriter implements an earlier version of the language defined in §3. Both of the annotation languages can enforce the common idioms seen in the 10 annotated modules, however we believe the new language is more succinct. We expect that the language will evolve further as we annotate more interfaces, and discover other idioms.

Component	Lines of code
Kernel rewriting plugin	150
Module rewriting plugin	1,452
Runtime checker	4,704

Figure 7: Components of LXFI.

The LXFI rewriter does not process assembly code, either in the core kernel or in modules. We manually inspect the assembly functions in the core kernel; none of them contains indirect calls. For modules, instrumentation is required if the assembly performs indirect calls or direct calls to an external function. In this case, developer must manually instrument the assembly by inserting calls to LXFI runtime checker. In our experience, modules use no assembly code that requires annotation.

LXFI requires all indirect calls in a module to be annotated to ensure API integrity. However, in some cases, the module rewriter fails to trace back to the function pointer declaration (e.g., due to an earlier phase of the compiler that optimized it away). In this case, developer has to modify the module's source code (e.g., to avoid the compiler optimization). For the 10 modules we annotated, such cases are rare: we changed 18 lines of code.

API integrity requires a complete set of core kernel functions to be annotated. However, in some cases, the Linux kernel inlines some kernel functions into modules. One approach is to annotate the inlined function, and let the module rewriter disable inlining of such functions. This approach, however, obscures the security boundary because these function are defined in the module, but must be treated the same as a kernel function. LXFI requires the boundary between kernel and module to be in one location by making either all or none of the functions inlined. In our experience, we have found that Linux is already well-written in this regard, and we had to change less than 10 functions (by not inlining them into a module) to enforce API integrity on 10 modules.

As pointed out in § 4.1, for indirect calls performed by the core kernel, LXFI checks that the annotation on function pointer matches the annotation on the invoked function `f`. Current implementation of LXFI performs checks when `f` has annotations, such as module functions that exported to kernel through assignment. A more strict and safe check is to enforce that `f` has annotations. Such check is not implemented because when `f` is defined in the core kernel, `f` may be static and has no annotation. We plan to implement annotation propagation in the kernel rewriter to solve this problem.

8 EVALUATION

This section evaluates the following 4 questions experimentally:

- Can LXFI stop exploits of kernel modules that have led to privilege escalation?
- How much work is required to annotate kernel/module interfaces?
- How much does LXFI slow down the SFI microbenchmarks?
- How much does LXFI slow down a Linux kernel module?

8.1 Security

To answer the first question we inspected 3 privilege escalation exploits using 5 vulnerabilities in Linux kernel modules revealed in 2010 that can lead to privilege escalation. Figure 8 shows three exploits and the corresponding vulnerabilities. LXFI successfully prevents all of the listed exploits as follows.

Exploit	CVE ID	Vulnerability type	Source location
CAN_BCM [17]	CVE-2010-2959	Integer overflow	net/can/bcm.c
Econet [18]	CVE-2010-3849	NULL pointer dereference	net/econet/af_econet.c
	CVE-2010-3850	Missed privilege check	net/econet/af_econet.c
	CVE-2010-4258	Missed context resetting	kernel/exit.c
RDS [19]	CVE-2010-3904	Missed check of user-supplied pointer	net/rds/page.c

Figure 8: Linux kernel module vulnerabilities that result in 3 privilege escalation exploits, all of which are prevented by LXFI.

CAN_BCM. Jon Oberheide posted an exploit to gain root privilege by exploiting an integer overflow vulnerability in the Linux CAN_BCM module [17]. The overflow is in the bcm_rx_setup function, which is triggered when the user tries to send a carefully crafted message through CAN_BCM. In particular, bcm_rx_setup allocates nframes*16 bytes of memory from a slab, where nframes is supplied by user. By passing a large value, the allocation size overflows, and the module receives less memory than it asked for. This allows an attacker to write an arbitrary value into the slab object that directly follows the objects allocated to CAN_BCM. In the posted exploit, the author first arranges the kernel to allocate a shmid_kernel slab object at a memory location directly following CAN_BCM's undersized buffer. Then the exploit overwrites this shmid_kernel object through CAN_BCM, and finally, tricks the kernel into calling a function pointer that is indirectly referenced by the shmid_kernel object, leading to a root privilege escalation.

To test the exploit against LXFI, we ported Oberheide's exploit from *x*86 to *x*86_64, since it depends on the size of pointer. LXFI prevents this exploit as follows. When the allocation size overflows, LXFI will grant the module a WRITE capability for only the number of bytes corresponding to the actual allocation size, rather than what the module asked for. When the module tries to write to an adjacent object in the same slab, LXFI detects that the module has no WRITE capability and raises an error.

Econet. Dan Rosenburg posted a privilege escalation exploit [18] by taking advantage of three vulnerabilities found by Nelson Elhage [8]. Two of them lie in the Econet module, and one in the core kernel. The two Econet vulnerabilities allow an unprivileged user to trigger a NULL pointer dereference in Econet. It is triggered when the kernel is temporarily in a context in which the kernel's check of a user-provided pointer is omitted, which allows a user to write anywhere in kernel space.

To prevent such vulnerabilities, the core kernel should always reset the context so that the check of a user-provided pointer is enforced. Unfortunately, kernel's do_exit failed to obey this rule. do_exit is called to kill a process when a NULL pointer dereference is captured in the kernel. Moreover, the kernel writes a zero into a user provided pointer (task->clear_child_tid) in do_exit. Along with the NULL pointer dereference triggered by the Econet vulnerabilities, the attacker is able to write a zero into an arbitrary kernel space address. By carefully arranging the kernel memory address for task->clear_child_tid, the attacker redirects econet_ops.ioctl to user space, and then gains root privilege in the same way as the RDS exploit. LXFI prevents the exploit by stopping the kernel from calling the indirect call of econet_ops.ioctl after it is overwritten with an illegal address.

RDS. Dan Rosenburg reported a vulnerability in the Linux RDS module in CVE-2010-3904 [19]. It is caused by a missing check of a user-provided pointer in the RDS page copying routine, allowing a local attacker to write arbitrary values to arbitrary memory locations. The vulnerability can be triggered by sending and receiving messages over a RDS socket. In the reported exploit, the attacker overwrites the rds_proto_ops.ioctl function pointer defined in the RDS module with the address of a user-space function. Then it tricks the kernel to indirectly call the rds_proto_ops.ioctl by invoking the ioctl system call. As a result, the local attacker can execute his own code in kernel space.

LXFI prevents the exploit in two ways. First, LXFI does not grant WRITE capabilities for a module's read-only section to the module (the Linux kernel does). Thus, the exploit cannot overwrite rds_proto_ops.ioctl in the first place, since it is declared in a read-only structure. To see if LXFI can defend against vulnerabilities that allow corrupting a writable function pointer, we made this memory location writable. LXFI is able to prevent the exploit, because it checks the core kernel's indirect call to rds_proto_ops.ioctl. The LXFI runtime detects that the function pointer is writable by the RDS module, and then it checks if RDS has a CALL capability for the target function. The LXFI runtime rejects the indirect call because RDS module has no CALL capability for invoking a user-space function. It is worth mentioning that the LXFI runtime would also reject the indirect call if the user overwrites the function pointer with a core kernel function that the module does not have a CALL capability for.

Other exploits. Vulnerabilities leading to privilege escalation are harmful. The attacker can typically mount other types of attacks exploiting the same vulnerabilities. For example, it can be used to hide a rootkit. The Linux kernel uses a hash table (pid_hash) for process lookup. If a rootkit deletes a process from the hash table, the process will not be listed by ps' shell command, but will still be scheduled to run. Without LXFI, a rootkit can exploit the above vulnerability in RDS to unlink itself from the pid_list hash table. Using the same technique as in the RDS exploit, we developed an exploit that successfully hides the exploiting process. The exploit runs as an unprivileged user. It overwrites rds_proto_ops.ioctl to point to a user space function. When the vulnerability is triggered, the core kernel calls the user space function, which calls detach_pid with current_task (both are exported kernel symbols). As before, LXFI prevents the vulnerability by disallowing the core kernel from invoking the function pointer into user-space code, because the RDS module has no CALL capability for that code. Even if the module overwrites the rds_proto_ops.ioctl function pointer to point directly to detach_pid, LXFI still prevents this exploit, because the RDS module does not have a CALL capability for detach_pid.

8.2 Annotation effort

To evaluate the work required to specify contracts for kernel/module APIs, we annotated 10 modules. These modules include several device categories (network, sound, and block), different devices within a category (e.g., two sound devices), and abstract devices (e.g., network protocols). The difficult part in annotating is understanding the interfaces between the kernel and the module, since there is little documentation. We typically follow an iterative process: we annotate the obvious parts of the interfaces, and try to run the module under LXFI. When running the module, the LXFI runtime raises alerts because the module attempts operations that LXFI forbids. We then iteratively go through these alerts, understand what the module is trying to do, and annotate interfaces appropriately.

Category	Module	# Functions		# Function Pointers	
		all	unique	all	unique
net device driver	e1000	81	49	52	47
sound device driver	snd-intel8x0	59	27	12	2
	snd-ens1370	48	13	12	2
net protocol driver	rds	77	30	42	26
	can	53	7	7	3
	can-bcm	51	15	17	1
	econet	54	15	20	3
block device driver	dm-crypt	50	24	24	14
	dm-zero	6	3	2	0
	dm-snapshot	55	16	28	18
Total		334		155	

Figure 9: The numbers of annotated function prototypes and function pointers for 10 modules. An annotation is considered unique if it is used by only one module. The *Total* row reports the total number of distinct annotations.

To quantify the work involved, we count the number of annotations required to support a kernel module. The number of annotations needed for a given module is determined by the number of functions (either defined in the core kernel or other modules) that the module invokes directly, and the number of function pointers that the core kernel and the module call. As Figure 9 shows, each module calls 6–81 functions directly, and is called by (or calls) 7–52 function pointers. For each module, the number of functions and function pointers that need annotating is much smaller. For example, supporting the `can` module only requires annotating 7 extra functions after all other modules listed in Figure 9 are annotated. The reason is that similar modules often invoke the same set of core kernel functions, and that the core kernel often invokes module functions in the same way across multiple modules. For example, the interface of the PCI bus is shared by all PCI devices. This suggests that the effort to support a new module can be small as more modules are supported by LXFI.

Some functions require checking, copying, or transferring a large number of capabilities. LXFI's annotation language supports programmer-defined capability iterators for this purpose, such as `skb_caps` for handling all of the capabilities associated with an `sk_buff` shown in Figure 4. In our experience, most annotations are simple, and do not require capability iterators. For the 10 modules, we wrote 36 capability iterators to handle idioms such as for loops or nested data structures. Each module required 3-11 capability iterators.

A second factor that affects the annotation effort is the rate of change of Linux kernel interfaces. We have inspected Linux kernel APIs for 20 major versions of the kernel, from 2.6.20 to 2.6.39, by counting the numbers of both functions that are directly exported from the core kernel and function pointers that appear in shared data structures using `ctags`. Figure 10 shows our findings. The results indicate that, although the number of kernel interfaces grows steadily, the number of interfaces changed with each kernel version is relatively modest, on the order of several hundred functions. This is in contrast to the total number of lines of code changed between major kernel versions, which is on the order of several hundred thousand lines of code.

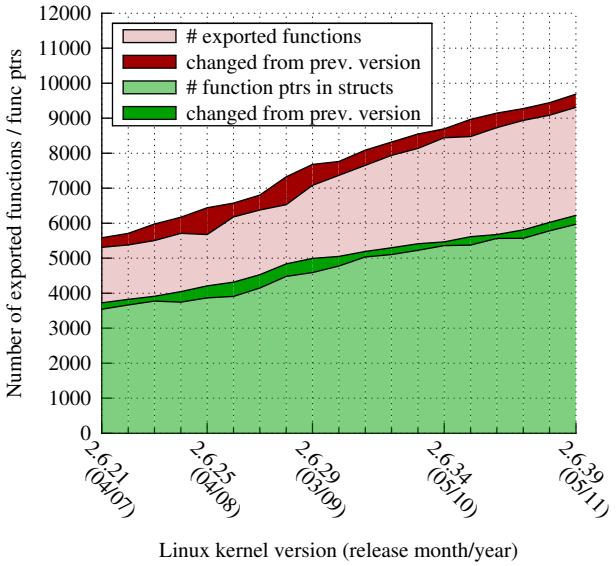

Figure 10: Rate of change for Linux kernel APIs, for kernel versions 2.6.21 through 2.6.39. The top curve shows the number of total and changed exported kernel functions; for example, 2.6.21 had a total of 5,583 exported functions, of which 272 were new or changed since 2.6.20. The bottom curve shows the number of total and changed function pointers in structs; for example, 2.6.21 had a total of 3,725 function pointers in structs, of which 183 were new or changed since 2.6.20.

8.3 Microbenchmarks

To measure the enforcement overhead, we measure how much LXFI slows down the SFI microbenchmarks [23]. To run the tests, we turn each benchmark into a Linux kernel module. We run the tests on a desktop equipped with an Intel(R) Core(TM) i3-550 3.2 GHz CPU, 6GB memory, and an Intel 82540EM Gigabit Ethernet card. For these benchmarks, we might expect a slightly higher overhead than XFI because the stack optimizations used in SFI are not applicable to Linux kernel modules, but on the other hand LXFI, like BGI, uses a compile-time approach to instrumentation, which provides opportunities for compile-time optimizations. We cannot compare directly to BGI because it targets the Windows kernel and no numbers were reported for the SFI microbenchmarks, but we would expect BGI to be faster than LXFI, because BGI's design carefully optimizes the runtime data structures to enable low-overhead checking.

Benchmark	Δ code size	Slowdown
`hotlist`	1.14×	0%
`lld`	1.12×	11%
`MD5`	1.15×	2%

Figure 11: Code size and slowdown of the SFI microbenchmarks.

Figure 11 summarizes the results from the measurements. We compare our result with the slowpath write-only overhead in XFI (Table 1 in [9]). For all benchmarks, the code size is 1.1x-1.2x larger with LXFI instrumentation, while with XFI the code size is 1.1x-3.9x larger. We believe that LXFI's instrumentation inserts less code

Test	Throughput		CPU %	
	Stock	LXFI	Stock	LXFI
TCP_STREAM TX	836 M bits/sec	828 M bits/sec	13%	48%
TCP_STREAM RX	770 M bits/sec	770 M bits/sec	29%	64%
UDP_STREAM TX	3.1 M/3.1 M pkt/sec	2.0 M/2.0 M pkt/sec	54%	100%
UDP_STREAM RX	2.3 M/2.3 M pkt/sec	2.3 M/2.3 M pkt/sec	46%	100%
TCP RR	9.4 K Tx/sec	9.4 K Tx/sec	18%	46%
UDP RR	10 K Tx/sec	8.6 K Tx/sec	18%	40%
TCP RR (1-switch latency)	16 K Tx/sec	9.8 K Tx/sec	24%	43%
UDP RR (1-switch latency)	20 K Tx/sec	10 K Tx/sec	23%	47%

Figure 12: Performance of `netperf` benchmark with stock and LXFI enabled `e1000` driver.

because LXFI does not employ fastpath checks (inlining memory-range tests for the module's data section to handle common cases [9]) as XFI does. Moreover, LXFI targets *x*86_64, which provides more registers, allowing the inserted instructions to be shorter.

Like XFI, LXFI adds almost no overhead for `hotlist`, because `hotlist` performs mostly read-only operations over a linked list, which LXFI does not instrument.

The performance of `lld` under LXFI (11% slowdown) is much better than for XFI (93% slowdown). This is because the code of `lld` contains a few trivial functions, and LXFI's compiler plugin effectively inlined them, greatly reducing the number of guards at function entries and exits. In contrast, XFI uses binary rewriting and therefore is unable to perform this optimization. Since BGI also uses a compiler plug-in, we would expect BGI to do as well or better than LXFI.

The slowdown of `MD5` is also negligible (2% compared with 27% for XFI). `oprofile` shows that most of the memory writes in `MD5` target a small buffer, residing in the module's stack frame. By applying optimizations such as inlining and loop unrolling, LXFI's compiler plugin detects that these writes are safe because they operate on constant offsets within the buffer's bound, and can avoid inserting checks. Similar optimizations are difficult to implement in XFI's binary rewriting design, but BGI again should be as fast or faster than LXFI.

8.4 Performance

To evaluate the overhead of LXFI on an isolated kernel module, we run `netperf` [14] to exercise the Linux `e1000` driver as a kernel module. We run LXFI on the same desktop described in §8.3. The other side of the network connection runs stock Linux 2.6.35 SMP on a desktop equipped with an Intel(R) Core(TM) i7-980X 3.33 GHz CPU, 24 GB memory, and a Realtek RTL8111/8168B PCIE Gigabit Ethernet card. The two machines are connected via a switched Gigabit network. In this section, "TX" means that the machine running LXFI sends packets, and "RX" means that the machine running LXFI receives packets from the network.

Figure 12 shows the performance of `netperf`. Each test runs for 10 seconds. The "CPU %" column reports the CPU utilization on the desktop running LXFI. The first test, TCP_STREAM, measures the TCP throughput of the `e1000` driver. The test uses a send buffer of 16,384 bytes, and a receive buffer of 87,370 bytes. The message size is 16,384 bytes. As shown in Figure 12, for both "TX" and "RX" workloads, LXFI achieves the same throughput as the stock `e1000` driver; the CPU utilization increases by 3.7× and 2.2× with LXFI, respectively, because of the added cost of capability operations.

UDP_STREAM measures UDP throughput. The UDP socket size is 126,976 bytes on the send side, and 118,784 bytes on the receive side. The test sends messages of 64 bytes. The two performance numbers report the number of packets that get sent and received. LXFI achieves 65% of the throughput of the stock version for TX, and achieves the same throughput for RX. The LXFI version cannot achieve the same throughput for TX because the CPU utilization reaches 100%, so the system cannot generate more packets. We expect that using a faster CPU would improve the throughput for TX (although the CPU overhead would remain high).

We run TCP_RR and UDP_RR to measure the impact of LXFI on latency, using the same message size, send and receive buffer sizes as above. We conducted two tests, each with a different network configuration.

In the first configuration, the two machines are connected the same subnet, and there are a few switches between them (but no routers). As shown in the middle rows of Figure 12, with LXFI, the throughput of TCP_RR is almost the same as the stock version, and the CPU utilization increases by 2.6×. For UDP_RR, the throughput decreases by 14%, and the CPU utilization increases by 2.2×.

Part of the latency observed in the above test comes from the network switches connecting the two machines. To understand how LXFI performs in a configuration with lower network latency, we connect the two machines to a dedicated switch and run the test again. As Figure 12 shows, the CPU utilization and the throughput increase for both versions. The relative overhead of LXFI increases because the network latency is so low that the processing of the next incoming packets are delayed by capability actions, slowing down the rate of packets received per second. We expect that few real systems use a network with such low latencies, and LXFI provides good throughput when even a small amount of latency is available for overlap.

Guard type	Guards per pkt	Time per guard (ns)	Time per pkt (ns)
Annotation action	13.5	124	1,674
Function entry	7.1	16	114
Function exit	7.1	14	99
Mem-write check	28.8	51	1,469
Kernel ind-call all	9.2	64	589
Kernel ind-call `e1000`	3.1	86	267

Figure 13: Average number of guards executed by the LXFI runtime per packet, the average cost of each guard, and the total time spent in runtime guards per packet for UDP_STREAM TX benchmark.

To understand the sources of LXFI's overheads, we measure the average number of guards per packet that the LXFI runtime executes, and the average time for each guard. We report the numbers for the UDP_STREAM TX benchmark, because LXFI performs worst for this workload (not considering the 1-switch network configuration). Figure 13 shows the results. As expected, LXFI spends most of the time performing annotation actions (grant, revoke, and check), and checking permissions for memory writes. Both of them are the most frequent events in the system. "Kernel ind-call all" and "Kernel ind-call `e1000`" show that the core kernel performs 9.2 indirect function calls per packet, around 1/3 of which are calls to the `e1000` driver that involve transmitting packets. This suggests that our writer-set

tracking optimization is effective at eliminating 2/3 of checks for indirect function calls.

8.5 Discussion

The results suggests that LXFI works well for the modules that we annotated. The amount of work to annotate is modest, requiring 8–133 annotations per module, including annotations that are shared between multiple modules. Instrumenting a network driver with LXFI increases CPU usage by 2.2–3.7×, and achieves the same TCP throughput as an unmodified kernel. However, UDP throughput drops by 35%. It is likely that we can use design ideas for runtime data structures from BGI to reduce the overhead of checking. In terms of security, LXFI is less beneficial to modules that must perform privileged operations; an adversary who compromises such a module will be able to invoke the privileged operation that the modules is allowed to perform. It would be interesting to explore how to refactor such modules to separate privileges. Finally, some modules have complicated semantics and the LXFI annotation language is not rich enough; for example, file systems have setuid and file permission invariants that are difficult to capture with LXFI annotations. We would like to explore how to increase LXFI's applicability in future work.

9 Related work

LXFI is inspired by XFI [9] and BGI [4]. XFI, BGI, and LXFI use SFI [26] to isolate modules. XFI assumes that the interface between the module and the support interface is simple and static, and does not handle overly permissive support functions. BGI extends XFI to handle more complex interfaces by manually interposing on every possible interaction between the kernel and module, and uses access control lists to restrict the operations a module can perform. Manual interposition for BGI is feasible because the Windows Driver Model (WDM) only allows drivers to access kernel objects, or register callbacks, through well-defined APIs. In contrast, the Linux kernel exposes its internal data objects to module developers. For example, a buggy module may overwrite function pointers in the kernel object to trick the kernel into executing arbitrary code. To provide API integrity for these complex interfaces, LXFI provides a capability and annotation system that programmers can use to express the necessary contracts for API integrity. LXFI's capabilities are dual to BGI's access control lists. Another significant difference between LXFI and BGI is LXFI's support for principals to partition the privileges held by a shared module. Finally, LXFI shows that it can prevent real and synthesized attacks, whereas the focus of BGI is high-performance fault isolation.

Mondrix [27] shows how to implement fault isolation for several parts of the Linux kernel, including the memory allocator, several drivers, and the Unix domain socket module. Mondrix relies on specialized hardware not available in any processor today, whereas LXFI uses software-based techniques to run on commodity *x*86 processors. Mondrix also does not protect against malicious modules, which drives much of LXFI's design. For example, malicious kernel modules in Mondrix can invoke core kernel functions with incorrect arguments, or simply reload the page table register, to take over the entire kernel.

Loki [28] shows how to privilege-separate the HiStar kernel into mutually distrustful "library kernels". Loki's protection domains correspond to user or application protection domains (defined by HiStar labels), in contrast with LXFI's domains which are defined by kernel component boundaries. Loki relies on tagged memory, and also relies on HiStar's simple kernel design, which has no complex subsystems like network protocols or sound drivers in Linux that LXFI supports.

Full formal verification along the lines of seL4 [15] is not practical for Linux, both because of its complexity, and because of its ill-defined specification. It may be possible to use program analysis techniques to check some limited properties of LXFI itself, though, to ensure that an adversary cannot subvert LXFI.

Driver isolation techniques such as Sud [3], Termite [21], Dingo [20], and Microdrivers [10] isolate device drivers at user-level, as do microkernels [7, 11]. This requires significantly re-designing the kernel interface, or restricting user-mode drivers to well-defined interfaces that are amenable to expose through IPC. Many kernel subsystems, such as protocol modules like RDS, make heavy use of shared memory that would not work well over IPC. Although there has been a lot of interest in fault containment in the Linux kernel [16, 24], fault tolerance is a weaker property than stopping attackers.

A kernel runtime that provides type safety and capabilities by default, such as Singularity [13], can provide strong API contracts similar to LXFI. However, most legacy OSes including Linux cannot benefit from it since they are not written in a type-safe language like C#.

SecVisor [22] provides kernel code integrity, but does not guarantee data protection or API integrity. As a result, code integrity alone is not enough to prevent privilege escalation exploits. OSck [12] detects kernel rootkits by enforcing type safety and data integrity for operating system data at hypervisor level, but does not address API safety and capability issues among kernel subsystems.

Overshadow [6] and Proxos [25] provide security by interposing on kernel APIs from a hypervisor. The granularity at which these systems can isolate features is more coarse than with LXFI; for example, Overshadow can just interpose on the file system, but not on a single protocol module like RDS. Furthermore, techniques similar to LXFI would be helpful to prevent privilege escalation exploits in the hypervisor's kernel.

10 Conclusion

This paper presents an approach to help programmers capture and enforce *API integrity* of complex, irregular kernel interfaces like the ones found in Linux. LXFI introduces capabilities and annotations to allow programmers to specify these rules for any given interface, and uses *principals* to isolate privileges held by independent instances of the same module. Using software fault isolation techniques, LXFI enforces API integrity at runtime. Using a prototype of LXFI for Linux, we instrumented a number of kernel interfaces with complex contracts to run 10 different kernel modules with strong security guarantees. LXFI succeeds in preventing privilege escalation attacks through 5 known vulnerabilities, and imposes moderate overhead for a network-intensive benchmark.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Sam King, for their feedback. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, by the DARPA UHPC program, and by NSF award CNS-1053143. Dong Zhou was supported by China 973 program 2007CB807901 and NSFC 61033001. The opinions in this paper do not necessarily represent DARPA or official US policy.

References

[1] Common vulnerabilities and exposures. From `http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux+kernel+2010`.

[2] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg. Security impact ratings considered harmful. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verita, Switzerland, May 2009.

[3] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pages 117–130, Boston, MA, June 2010.

[4] M. Castro, M. Costa, J. P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[5] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011.

[6] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, March 2008.

[7] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.

[8] N. Elhage. CVE-2010-4258: Turning denial-of-service into privilege escalation. `http://blog.nelhage.com/2010/12/cve-2010-4258-from-dos-to-privesc/`, December 2010.

[9] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[10] V. Ganapathy, M. Renzelmann, A. Balakrishnan, M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, March 2008.

[11] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum. Fault isolation for device drivers. In *Proceedings of the 2009 IEEE Dependable Systems and Networks Conference*, Lisbon, Portugal, June–July 2009.

[12] O. Hofmann, A. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, March 2011.

[13] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, WA, October 2005.

[14] R. Jones. Netperf: A network performance benchmark, version 2.45. `http://www.netperf.org`.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[16] A. Lenharth, V. S. Adve, and S. T. King. Recovery domains: An organizing principle for recoverable operating systems. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–60, Washington, DC, March 2009.

[17] J. Oberheide. Linux kernel CAN SLUB overflow. `http://jon.oberheide.org/blog/2010/09/10/linux-kernel-can-slub-overflow/`, September 2010.

[18] D. Rosenberg. Econet privilege escalation exploit. `http://thread.gmane.org/gmane.comp.security.full-disclosure/76457`, December 2010.

[19] D. Rosenberg. RDS privilege escalation exploit. `http://www.vsecurity.com/download/tools/linux-rds-exploit.c`, October 2010.

[20] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, March 2009.

[21] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[22] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[23] C. Small and M. I. Seltzer. Misfit: Constructing safe extensible systems. *IEEE Concurrency*, 6:34–41, 1998.

[24] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), November 2004.

[25] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 279–292, Seattle, WA, November 2006.

[26] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.

[27] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[28] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, pages 225–240, San Diego, CA, December 2008.