

Fast Byte-Granularity Software Fault Isolation

Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado,
Periklis Akritidis*, Austin Donnelly, Paul Barham, Richard Black

Microsoft Research
Cambridge, UK

ABSTRACT

Bugs in kernel extensions remain one of the main causes of poor operating system reliability despite proposed techniques that isolate extensions in separate protection domains to contain faults. We believe that previous fault isolation techniques are not widely used because they cannot isolate existing kernel extensions with low overhead on standard hardware. This is a hard problem because these extensions communicate with the kernel using a complex interface and they communicate frequently. We present BGI (Byte-Granularity Isolation), a new software fault isolation technique that addresses this problem. BGI uses efficient byte-granularity memory protection to isolate kernel extensions in separate protection domains that share the same address space. BGI ensures type safety for kernel objects and it can detect common types of errors inside domains. Our results show that BGI is practical: it can isolate Windows drivers without requiring changes to the source code and it introduces a CPU overhead between 0 and 16%. BGI can also find bugs during driver testing. We found 28 new bugs in widely used Windows drivers.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection; D.4.5 [Operating Systems]: Reliability; D.4.8 [Operating Systems]: Performance

General Terms

Reliability, Security, Performance, Measurement

1. INTRODUCTION

Kernel extensions are used to customize commodity operating systems. A typical operating system runs tens of kernel extensions including device drivers, file systems, and

*Work done while an intern at MSR Cambridge. Periklis Akritidis is affiliated with Cambridge University.

even web servers. To achieve good performance, most of these extensions are fully trusted: they are written in unsafe languages and they share the address space of the operating system kernel. Empirical evidence shows that this trust is misplaced. Bugs in kernel extensions are one of the main causes of poor reliability in operating systems [7, 42] and they can often be exploited by attackers [16, 21, 25].

Previous work proposed techniques that isolate kernel extensions in separate protection domains to contain faults (e.g., [11, 28, 35, 36, 41–43, 45]) or that move extensions to user mode processes (e.g., [12, 13, 18, 19, 26]). Unfortunately, the reliability of commodity operating systems has benefited little from this work. Both Linux and Windows support user mode device drivers [9, 30] but only some types of devices are supported and there is a significant performance penalty. Therefore, the vast majority of extensions still runs in the kernel address space without isolation. We believe that fault isolation techniques are not widely used because they cannot isolate existing kernel extensions with low overhead on standard hardware. This is a hard problem because extensions communicate with the kernel using a *complex interface* and they *communicate frequently*.

We present BGI (Byte-Granularity Isolation), a new software fault isolation technique that addresses this problem. BGI uses efficient byte-granularity memory protection to isolate kernel extensions. It can isolate existing Windows drivers with low overhead and no modifications to the source code. BGI can also be used to isolate extensions to other software systems, for example, Internet browser plug-ins and extensions to Microsoft Office. This paper focuses on isolating kernel extensions because we believe this is the most compelling application for the technology.

BGI is implemented as a compiler plug-in that generates instrumented code for kernel extensions, and an interposition library that mediates communication between the extensions and the kernel. BGI runs extensions in separate protection domains that share the same address space. It associates an access control list (ACL) with each byte of virtual memory that lists the domains that can access the byte and how they can access it. Access rights are granted and revoked by code inserted by our compiler and by the interposition library according to the semantics of the operation being invoked. Protection is enforced by inline checks inserted by our compiler and by checks performed by the interposition library.

Previous work on fine-grained memory protection [45] relies on special hardware to achieve good performance. BGI achieves good performance with a software implementation

by using a combination of compile time changes to the layout of data, careful design of the data structures that store ACLs, static analysis, and judicious tradeoffs between performance and isolation. Like other systems [11, 42, 43], BGI does not check ACLs before reads, and checks before other types of access are not performed atomically with the access [11]. This enables efficient but still effective isolation of extensions that communicate frequently with the kernel. Additionally, BGI can detect common types of errors inside domains, for example, corruption of return addresses and exception handler pointers, and sequential buffer overflows and underflows.

Access rights in BGI include not only *read* and *write* but also *icall* and *type* rights. BGI grants an *icall* right on the first byte of functions that can be called indirectly or passed in kernel calls that expect function pointers. This is used to enforce a form of control flow integrity [1, 24] that prevents extensions from bypassing our checks and ensures that control flow transfers across domains target only allowed entry points. Cross domain control transfers are implemented as simple function calls without stack switches, extra copies, or page table changes.

Type rights are used to enforce dynamic type safety for kernel objects. There is a different *type* right for each type of kernel object, for example, mutex and device. When an extension receives an object from the kernel, the interposition library grants the appropriate *type* right on the first byte of the object, grants *write* access to fields that can be written directly by the extension, and revokes *write* access to the rest of the object. Access rights change as the extension interacts with the kernel, for example, rights are revoked when the object is deleted. The interposition library also checks if extensions pass objects of the expected type in cross domain calls. This ensures type safety: extensions can only create, delete, and manipulate kernel objects using the appropriate interface functions. Type safety prevents many incorrect uses of the kernel interface.

We implemented a BGI prototype that can isolate existing Windows Vista drivers in x86 computers. We chose Windows because it is widely used, there are many millions of lines in drivers that use the Windows driver interfaces, and these interfaces are complex. We believe that previous fault isolation systems would be unable to provide strong isolation guarantees for Windows drivers or would incur an unacceptable performance overhead.

We used BGI to isolate 16 Windows drivers including the FAT file system and the driver for a 10Gb/s network card. Our experimental results show that BGI can isolate faults injected in these drivers effectively and that BGI’s CPU overhead is between 0 and 16% with an average of only 6.4%. We believe this overhead is low enough for BGI to be used in practice to improve reliability, security, and availability of Windows drivers. BGI can prevent errors in isolated drivers from corrupting state elsewhere in the operating system, it can prevent attackers from exploiting these errors, and it can recover drivers with errors.

BGI is also a good bug finding tool. We ran existing tests on drivers isolated with BGI and found 28 new bugs due to incorrect use of kernel interfaces. Since BGI has low overhead, it can be used not only during pre-release testing but also during production to collect better debugging information for error reporting tools (e.g. [15]).

```
void ProcessRead(PDEVICE_OBJECT d, IRP *irp) {
    KEVENT e;
    int j=0;
    PIO_STACK_LOCATION isp;

    KeInitializeEvent(&e, NotificationEvent, FALSE);

    SetParametersForDisk(irp);
    IoSetCompletionRoutine(irp, &DiskReadDone, &e,
                          TRUE, TRUE, TRUE);
    IoCallDriver(diskDevice, irp);
    KeWaitForSingleObject(&e, Executive,
                        KernelMode, FALSE, NULL);

    isp = IoGetCurrentIrpStackLocation(irp);
    for(; j < isp->Parameters.Read.Length; j++) {
        irp->AssociatedIrp.SystemBuffer[j] ^= key;
    }
    IoCompleteRequest(irp, IO_DISK_INCREMENT);
}
```

Figure 1: Example extension code: processing a read request in an encrypted file system driver.

2. OVERVIEW

Figure 1 highlights some of the difficulties in isolating kernel extensions. It shows how a simplified encrypted file system driver might process a read request in the Windows Driver Model (WDM) [31]. We will use this example to illustrate how BGI works. We omit error handling for clarity.

The driver registers the function `ProcessRead` with the kernel when it is loaded. When an application reads data from a file, the kernel creates an I/O Request Packet (IRP) to describe the operation and calls `ProcessRead` to process it. `ProcessRead` sends the request to a disk driver to read the data from disk and then decrypts the data (by XORing the data with `key`). `SetParametersForDisk` and `DiskReadDone` are driver functions that are not shown in the figure.

`ProcessRead` starts by initializing an event synchronization object `e` on the stack. Then it modifies the IRP to tell the disk driver what to do. Windows drivers are stacked and IRPs contain a stack location for each driver in the driver stack. This IRP stack location has parameters for the operation that the driver should perform and a completion routine that works like a return address. `ProcessRead` sets the parameters for the disk in the next IRP stack location and it sets `DiskReadDone` as the completion routine (with argument `&e`). Then it passes the IRP to the disk driver and waits on `e`. When the disk read completes, the kernel calls the `DiskReadDone` function, which signals event `e`. This unblocks `ProcessRead` allowing it to decrypt the data read from disk. When it finishes, the driver calls `IoCompleteRequest` to complete the request and tell the kernel the data is ready.

The example shows some of the complexity of the WDM interface and it shows that drivers interact with the kernel frequently. A practical fault isolation mechanism for WDM drivers could have a significant impact on the reliability of Windows because the vast majority of Windows extensions use the WDM interface. Some previous fault isolation techniques target existing extensions and commodity operating systems [11, 28, 41–43, 46] but we believe they are unable to provide strong isolation guarantees for WDM drivers with acceptable overhead.

SafeDrive [46] implements a bounds checking mechanism

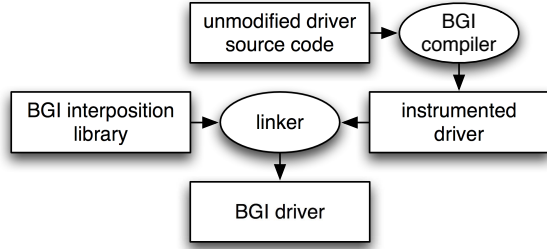


Figure 2: Producing a BGI extension.

for C that requires programmers to annotate extension source code. The overhead of SafeDrive is low but it provides weak isolation guarantees. SafeDrive does not prevent temporal errors, for example, if `DiskReadDone` erroneously completes the IRP, SafeDrive does not prevent the decryption code from writing to a deleted buffer. SafeDrive prevents out-of-bounds reads and BGI does not but this can be a problem because SafeDrive does not distinguish between read and write accesses. For example, SafeDrive would allow the driver to write to fields in the IRP that can be read but not modified by extensions.

The other isolation techniques also fail to provide strong isolation guarantees for WDM drivers. They allocate a range of addresses to an extension and allow writes to addresses in this range. Since the stacks and heaps used by the extension lie within this range, these isolation techniques do not prevent writes to kernel objects stored in extension stacks or heaps. They also fail to ensure that these objects are initialized before they are used. For example, they do not prevent writes to event `e` in the example above. This is a problem because the event object includes a linked list header that is used internally by the kernel. The driver can cause the kernel to write to arbitrary addresses if it overwrites the event or fails to initialize it.

These techniques also perform poorly when extensions interact with the kernel frequently because they copy objects passed by reference in cross domain calls. For example, they would copy the buffer to allow `ProcessRead` to write to the buffer during decryption. Nooks [41, 42] incurs additional overhead when switching domains because it uses hardware page protection for isolation. XFI [11] can avoid these copies at the expense of slow write checks. XFI checks if writes target an address within the range allocated to the extension and, if this fails, checks if the target address is within an address range in a list of exceptions. This performs poorly with WDM drivers because the list of exceptions is long.

Byte-granularity memory protection provides BGI with adequate spatial and temporal resolution to avoid these problems. BGI can grant access to precisely the bytes that a domain should access and it can check accesses to these bytes efficiently regardless of where they are in memory. It can also control precisely when the domain is allowed to access these bytes because it can grant and revoke access efficiently. Therefore, BGI can provide strong isolation guarantees for WDM drivers with low overhead and no changes to the source code. For example, it can contain all the errors discussed above within the domain of the extension.

We designed our data structures carefully to allow a compact representation for ACLs. In the common case, we store information about ACLs in an array that has one byte of information for each 8 bytes of memory, which adds a space overhead of approximately 12.5%. This array can be ac-

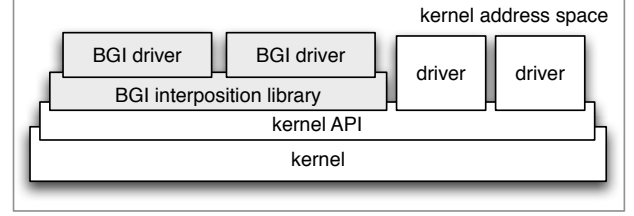


Figure 3: Kernel address space with BGI.

cessed efficiently to check and modify ACLs: we check ACLs for a byte range by comparing consecutive entries in the array with small integer values, and we modify ACLs by setting consecutive entries in the array. There is a slow path that covers the general case but our compiler changes the layout of data to ensure this slow path is rarely taken.

BGI is not designed to isolate malicious code. Attackers can control the input to a driver but we assume that they do not write the driver code. BGI is designed to contain faults due to errors in the driver and to contain attacks that exploit these errors. We trust driver writers to use our compiler. The compiler could be distributed with the Windows Driver Kit [31], which already includes a compiler that is used by most driver writers.

Nooks and SafeDrive assume a similar attacker model but SFI [43] and XFI were designed to isolate malicious extensions. XFI uses a static verifier [32] to verify that untrusted code has appropriate checks before it is allowed to execute. The BGI compiler generates code that satisfies properties similar to those enforced by XFI’s verifier and BGI performs a superset of XFI’s runtime checks. So in principle, we would be able to isolate malicious code if we implemented a verifier similar to XFI’s. In practice, there are several issues that make it hard. First, a malicious driver can subvert SFI and XFI by calling host functions in the wrong order or with the wrong arguments. BGI’s dynamic type checks can prevent these attacks but it is hard to prove that they are sufficient given the complexity of the interfaces. Second, the inline access checks in XFI and BGI are not atomic with the access to improve performance: thread t can check if an access is allowed, access can be revoked by thread t' , and then t may perform an access that is no longer allowed. This small window of vulnerability is hard to exploit if the attacker does not control the driver code but it is easily exploitable by malicious code. Third, SFI, XFI, and BGI improve performance by not checking reads, which is not acceptable when isolating malicious code that can read secrets from memory. Fourth, none of these systems prevents malicious code from programming device hardware to write to arbitrary memory locations. To isolate malicious code, they would have to be extended to use emerging IO MMU technology. It would be possible to address all these issues at the expense of decreased performance but this would hinder adoption.

To isolate a Windows driver using BGI, a user compiles the driver with the BGI compiler and links the driver with the BGI interposition library as shown in Figure 2. This driver runs on Windows Vista (with the patch that we describe in Section 6.2). BGI-isolated drivers can run alongside trusted extensions on the same system, as shown in Figure 3. The instrumentation inserted by the compiler and the interposition library enforce isolation as described next.

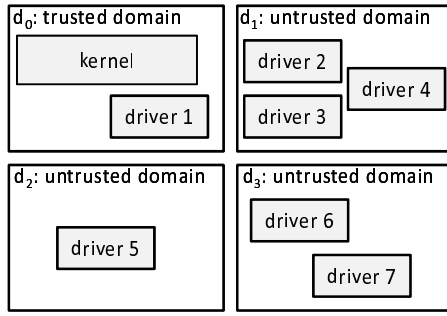


Figure 4: Example partition into domains.

3. PROTECTION MODEL

BGI runs code in separate protection domains to contain faults. There is one trusted domain where the kernel and trusted extensions run and one or more untrusted domains. Each untrusted domain can run one or more untrusted extensions. We say that a thread is running in a domain when it executes code that runs in that domain. BGI does not constrain memory accesses performed by threads running in the trusted domain.

Virtual memory includes both *system space* and *user space*. System space is shared by all processes and each process has a private user space. All BGI domains share system space and the process user spaces but BGI associates an access control list (ACL) with each byte of virtual memory to control memory accesses. An ACL lists the domains that can access the byte and the access rights they have to the byte. Some extensions can access user space directly in the context of the user process that initiates the I/O operation. Extensions do not access physical addresses directly.

The access rights in BGI are *read*, *write*, and several *icall*, *type* and *ownership* rights. All access rights allow read access and, initially, all domains have the *read* right to every byte in virtual memory. BGI does not constrain read accesses because checking reads in software is expensive and the other checks prevent errors due to incorrect reads from propagating outside the domain. We already mentioned *icall*, and *type* rights. *ownership* rights are used to keep track of the domain that should free an allocated object and which deallocation function to use. In the current implementation, a domain cannot have more than one of these rights to the same byte of virtual memory.

The interposition library and the code inserted by the compiler use two primitives to manipulate ACLs: *SetRight*, which grants and revokes access rights, and *CheckRight*, which checks access rights before memory accesses. When a thread running in domain d calls *SetRight*(p, s, r) with $r \neq \text{read}$, BGI grants d access right r to all the bytes in the range $[p, p + s)$. For example, *SetRight*(p, s, write) gives the domain write access to the byte range. To revoke any other access rights to byte range $[p, p + s)$, a thread running in the domain calls *SetRight*(p, s, read). To check ACLs before a memory access to byte range $[p, p + s)$, a thread running in domain d calls *CheckRight*(p, s, r). If d does not have right r to all the bytes in the range, BGI throws an exception.

BGI defines variants of *SetRight* and *CheckRight* that are used with *icall* and *type* rights. *SetType*(p, s, r) marks p as the start of an object of type r and size s that can be used by the domain and prevents writes to the range $[p, p + s)$. *CheckType*(p, r) checks if the domain has the right to use p

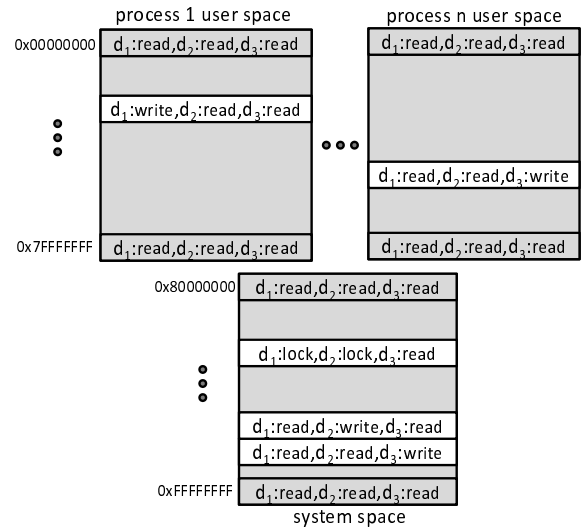


Figure 5: Example access control lists (ACLs).

as the start of an object of type r . *SetType*(p, s, r) is semantically equivalent to *SetRight*($p, 1, r$); *SetRight*($p+1, s-1, \text{read}$), and *CheckType*(p, r) is equivalent to *CheckRight*($p, 1, r$) but we have specific optimizations for these variants.

Figure 4 shows an example partition of seven kernel drivers into domains. Driver 1 runs in the trusted domain with the kernel code because it is trusted. The other drivers are partitioned into three untrusted domains. There is a single driver in domain d_2 but the other untrusted domains run more than one driver. Frequently, the functionality needed to drive a device is implemented by multiple driver binaries that communicate directly through custom interfaces. These drivers should be placed in the same domain. Figure 5 shows example ACLs for the domains in Figure 4. The greyed boxes correspond to the default ACL that only allows domains to read the byte. The other ACLs grant some of the domains more rights on accesses to the corresponding byte, for example, one of them grants domains d_1 and d_2 the right to use a shared lock.

4. INTERPOSITION LIBRARY

An untrusted extension is linked with the BGI interposition library that mediates all communication between the extension and the kernel. The library contains two types of wrappers: *kernel wrappers* wrap kernel functions that are called by the extension and *extension wrappers* wrap extension functions that are called by the kernel. Wrappers use the memory protection primitives to grant, revoke, and check access rights according to the semantics of the functions they wrap. Since these primitives target the domain of the thread that calls them, we say that the wrappers run in the extension's domain even though they are trusted.

Wrappers: BGI wrappers serve a similar purpose to Nooks wrappers [42] but they transfer control across domains using a simple function call without changing page tables, stacks, or copying arguments. Additionally, byte-granularity memory protection allows extensive checking with low overhead.

A kernel wrapper does not trust the caller but trusts the callee. It checks rights to the arguments supplied by the extension, it may revoke rights to some of those arguments,

it calls the wrapped kernel function, and it may grant rights to some objects returned by the function. The sequence of steps executed by an extension wrapper is different because the caller is trusted but the callee is not. An extension wrapper may grant rights to some arguments, it calls the wrapped extension function, it may revoke rights to some arguments, and it checks values returned by the extension. There are different types of argument and result checks for the different types of rights: *write*, *ownership*, *icall*, and *type*.

Write and ownership checks prevent extension errors from making threads in other domains write to arbitrary locations. A kernel wrapper calls *CheckRight(p, s, write)* for each memory range argument, $[p, p + s)$, that may be written to by the function being wrapped. An extension wrapper performs similar checks on memory ranges returned by the extension that may be written to by the caller.

Write access is granted and revoked by the interposition library and code inserted by the compiler. The wrapper for the extension initialization function grants write access to global variables. Write access to local variables is granted and revoked by code inserted by the compiler.

Most fault isolation systems require a separate heap per domain but fine-grained memory protection allows all domains to share the same heap. The kernel wrappers for heap allocation functions grant write access to allocated memory and the wrappers for deallocation functions revoke it. The wrappers for allocation functions also grant *ownership* rights on a guard before or after the allocated memory (depending on whether the allocation is smaller or larger than a page). Guards are 8-byte memory areas that are not writable by extensions. *ownership* rights are used to identify the allocation function and the domain that allocated the memory. The wrappers for deallocation functions check that the calling domain has the appropriate ownership right and that it has write access to the region being freed. If these checks fail, they signal an error. Otherwise, they revoke the ownership right and write access to the memory being freed. This ensures that only the domain that owns an object can free the object, that it must use the correct deallocation function, and that an object can be freed at most once.

Call checks prevent extension errors from making threads in other domains execute arbitrary code. Some kernel functions take function pointer arguments. Since the kernel may call the functions they point to, the interposition library checks if the extension has the appropriate *icall* right to these functions. Kernel wrappers call *CheckType(p, icallN)* on each function pointer argument p before calling the kernel function they wrap, where N is the number of stack bytes used by the arguments to an indirect call through p . The `stdcall` calling convention used in Windows drivers requires the callee to remove its arguments from the stack before returning. Therefore, the *icall* rights encode N to prevent stack corruption when functions with the wrong type are called indirectly. Extension wrappers check function pointers returned by extension functions.

The *icall* rights are also granted and revoked by the interposition library with help from the compiler. The compiler collects the addresses of all functions whose address is taken by the extension code and the number of bytes consumed by their arguments on the stack. This information is stored in a section in the extension binary. The wrapper for

the driver initialization function calls *SetType(p, 1, icallN)* for every function address p and byte count N in this section. When kernel functions return function pointers, their wrappers replace these pointers by pointers to the corresponding kernel wrappers and grant the appropriate *icall* rights. Since BGI does not grant *icall* rights in any other case, cross-domain calls into the domain can only target valid entry points: functions whose address was taken in the code of an extension running in the domain and kernel wrappers whose pointers were returned by the interposition library.

Type checks are used to enforce a form of dynamic type safety for kernel objects. There is a different *type* right for each type of kernel object. When a kernel function allocates or initializes a kernel object with address p , size s , and type t , its wrapper calls *SetType(p, s, t)* and grants write access to any fields that can be written directly by the extension. The wrappers for kernel functions that receive kernel objects as arguments check if the extension has the appropriate *type* right to those arguments, and wrappers for kernel functions that deallocate or uninitialized objects revoke the *type* right to the objects. Since many kernel objects can be stored in heap or stack memory allocated by the extension, BGI also checks if this memory holds active kernel objects when it is freed. Together these checks ensure that extensions can only create, delete, and manipulate kernel objects using the appropriate kernel functions. Furthermore, extensions in an untrusted domain can only use objects that were received from the kernel by a thread running in the domain.

The checks performed by BGI go beyond traditional type checking because the type, i.e., the set of operations that are allowed on an object, changes as the extension interacts with the kernel. BGI implements a form of dynamic type-state [38] analysis. For example in Figure 1, the extension wrapper for *ProcessRead* grants *irp* right to the first byte of the IRP. This allows calls to *IoSetCompletionRoutine* and *IoCallDriver* that check if they receive a valid IRP. But the *irp* right is revoked by the wrapper for *IoCallDriver* to prevent modifications to the IRP while it is used by the disk driver. The extension wrapper for *DiskReadDone* grants the *irp* right back after the disk driver is done. Then the right is revoked by the wrapper for *IoCompleteRequest* because the IRP is deleted after completion. These checks enforce interface usage rules that are documented but were not enforced.

Another example are the checks in the wrappers for kernel functions that manage splay trees. Objects that can be inserted in splay trees contain a field of type `RTL_SPLAY_LINKS`. The wrapper for the insertion function takes a pointer p to one of these fields, calls *CheckRight(p, sizeof(*p), write)* followed by *SetType(p, sizeof(*p), splay)* and inserts the object in the tree. The wrapper for the remove function calls *CheckType(p, splay)*, removes the object from the tree, and calls *SetRight(p, sizeof(*p), write)*. This prevents many incorrect uses of the interface, e.g., an object cannot be removed from a tree before being inserted, it cannot be inserted a second time without first being removed, and the extension cannot corrupt the tree pointers while the object is inserted in the tree. Wrappers for list functions are similar.

In addition to using access rights to encode object state, some kernel wrappers use information in the fields of objects to decide whether a function can be called without corrupting kernel state. This is safe because BGI prevents the extension from modifying these fields.

```

_BGI_KeInitializeDpc(PRKDPC d,
    PKDEFERRED_ROUTINE routine, PVOID a) {
    CheckRight(d, sizeof(KDPC), write);
    CheckFuncType(routine, PKDEFERRED_ROUTINE);
    KeInitializeDpc(d, routine, a);
    SetType(d, sizeof(KDPC), dpc);
}

BOOLEAN
_BGI_KeInsertQueueDpc (PRKDPC d, PVOID a1,
    PVOID a2) {
    CheckType(d, dpc);
    return KeInsertQueueDpc(d, a1, a2);
}

```

Figure 6: Kernel wrappers for KeInitializeDpc and KeInsertQueueDpc.

Figure 6 shows two example kernel wrappers. The first one wraps the `KeInitializeDpc` function that initializes a data structure called a deferred procedure call (DPC). The arguments are a pointer to a memory location supplied by the extension that is used to store the DPC, a pointer to an extension function that will be later called by the kernel, and a pointer argument to that function. The wrapper starts by calling `CheckRight(d, sizeof(KDPC), write)` to check if the extension has write access to the memory region where the kernel is going to store the DPC. Then it checks if the extension has the appropriate *icall* right to the function pointer argument. `CheckFuncType` is a macro that converts the function pointer type into an appropriate *icall* right and calls `CheckType`. In this case, it calls `CheckType(routine, icall16)`. If these checks succeed, the DPC is initialized and the wrapper grants the extension *dpc* right to the byte pointed to by *d* and revokes write access to the DPC object. It is important to prevent the extension from writing directly to the object because it contains a function pointer and linked list pointers. If the extension corrupted the DPC object, it could make the kernel execute arbitrary code or write to an arbitrary location. `KeInsertQueueDpc` is one of the kernel functions that manipulate DPC objects. Its wrapper performs a type check to ensure that the first argument points to a valid DPC. These type checks prevent several incorrect uses of the interface, for example, they prevent a DPC object from being initialized more than once or being used before it is initialized.

In Figure 6, the wrapper for `KeInitializeDpc` passes the function pointer `routine` to the kernel. In some cases, the wrapper replaces the function pointer supplied by the extension by a pointer to an appropriate extension wrapper. This is not necessary in the example because `routine` returns no values and it is invoked with arguments that the extension already has the appropriate rights to (*d*, *a*, *a1*, and *a2*).

We have implemented 262 kernel wrappers and 88 extension wrappers. These cover the most common WDM, WDF, and NDIS interface functions [31] and include all interface functions used by the drivers in our experiments. Most of the wrappers are as simple as the ones in our example and could be generated automatically from source annotations similar to those proposed in [17, 46]. There are 16700 lines of code in the interposition library. Although writing wrappers represents a significant amount of work, it only needs to be done once for each interface function by the OS vendor. Driver writers do not need to write wrappers or change their source code.

5. COMPILER

Windows kernel extensions are written in C or C++. The BGI compiler instruments untrusted extensions to redirect kernel function calls to the interposition library, to grant and revoke access rights to stack locations, and to check access rights before writes and indirect calls. We used the Phoenix framework [29] to implement the BGI compiler. We also implemented a binary rewriter based on Vulcan [37] that performs similar transformations on binaries with symbols. The rewriter allows driver writers to use compilers other than ours but we do not discuss it further because it currently produces somewhat slower code.

To ensure that all communication between untrusted extensions and the kernel is mediated by the interposition library, the compiler rewrites all calls to kernel functions to call the corresponding wrappers in the interposition library. The compiler also modifies extension code that takes the address of a kernel function to take the address of the corresponding kernel wrapper in the interposition library. This ensures that indirect calls to kernel functions are also redirected to the interposition library.

The compiler inserts calls to `SetRight` in function prologues to grant the domain write access to local variables on function entry. In the example in Figure 1, it inserts `SetRight(&e, sizeof(e), write)` in the prologue of the function `ProcessRead`. To revoke access to local variables on function exit, the compiler modifies function epilogues to first verify that local variables do not store active kernel objects and then call `SetRight` to revoke access.

The compiler inserts a check before each write in the extension code to check if the domain has write access to the target memory locations. It inserts `CheckRight(p, s, write)` before a write of *s* bytes to address *p*. The compiler also inserts checks before indirect calls in the extension code. It inserts `CheckType(p, icallN)` before an indirect call through pointer *p* that uses *N* bytes for arguments in the stack.

To prevent untrusted extensions from executing privileged instructions or bypassing our checks because of programmer mistakes, the compiler does not allow inline assembly or calls to BGI’s memory protection primitives. Use of inline assembly in Windows drivers is already discouraged [31].

Together the checks inserted by the compiler and performed by the interposition library are sufficient to ensure control flow integrity: untrusted extensions cannot bypass the checks inserted by the compiler, direct calls target functions in the extension or wrappers of kernel functions that are named explicitly in the extension code, indirect calls target functions whose address was taken in the extension code or whose address was returned by the interposition library, returns transfer control back to the caller, and exceptions transfer control to the appropriate handler. BGI does not need additional checks on returns or exception handling because write checks prevent corruption of return addresses and exception handler pointers. Since we control the compiler, we can ensure that there are no other types of indirect control flow transfer, e.g., jump tables for switch statements.

We also prevent many attacks internal to a domain. Control flow integrity prevents the most common attacks that exploit errors in extension code because these attacks require control flow to be transferred to injected code or to chosen locations in code that is already loaded. BGI also prevents sequential buffer overflows and underflows that can be used to mount attacks that do not violate control flow integrity.

These are prevented by the write checks because heap memory has guards and the compiler modifies the layout of global and local variables such that there is a guard between consecutive variables (as in [2]). The checks performed by heap allocation functions also prevent attackers from exploiting errors that corrupt heap metadata (e.g., double frees).

6. BYTE-GRANULARITY PROTECTION

We have shown how byte-granularity memory protection enables BGI to provide strong isolation guarantees for existing Windows kernel extensions. But this protection mechanism must have low overhead for BGI to be practical. This section describes our efficient software implementation of byte-granularity memory protection.

6.1 Drights

To store ACLs compactly, BGI uses a small integer, a *dright*, to encode a pair with a domain and an access right. When a domain is created to contain a set of extensions, we count the number of distinct access rights used by the extensions and we allocate the same number of distinct drights to the domain. These drights are unique across domains except for the one corresponding to the *read* access right, which is zero for all domains. When a domain is destroyed, its drights can be reclaimed for use by other domains.

We use two optimizations to reduce the number of bits needed to encode drights. The first one reduces the number of distinct access rights used in extensions by exploiting a form of subtyping polymorphism in the kernel. Several types of kernel object have the same super type: they have a common header with an integer field that indicates the subtype. For example, there are 17 subtypes of *dispatcher object* in the WDM interface, which include events, mutexes, DPCs, and threads. We take advantage of this by using a single type right for the super type and checking the field that indicates the subtype when necessary. This is safe because we prevent write access to this field. This optimization is very effective because this pattern is common.

The other optimization reduces the number of distinct *icall* rights. We use a single *icall* right per domain and store the number of bytes used by function arguments in an integer just before the function code. The write checks prevent untrusted extensions from modifying this integer. The indirect call checks start by checking if the domain has *icall* right to the first byte in the target function and then check the number of bytes stored before the function.

Currently, our x86 implementation uses 1-byte drights and we used 47 distinct access rights across our 16 test drivers. Many of these rights are used only by a particular type of driver (e.g., a file system or a network driver) or by drivers that use a particular version of the interface (e.g., WDF [31]). If every untrusted domain used all these rights, we would be able to support five separate untrusted domains but this is unlikely. In practice, we expect to support up to 15 or more independent untrusted domains in addition to the trusted domain. We believe this is sufficient for most scenarios because we expect many drivers developed by the operating system vendor to run in the trusted domain and we expect each untrusted domain to run several related drivers.

6.2 Tables

BGI uses four data structures to store ACLs at runtime: a *kernel table*, a *user table* per process, a *kernel conflict table*,

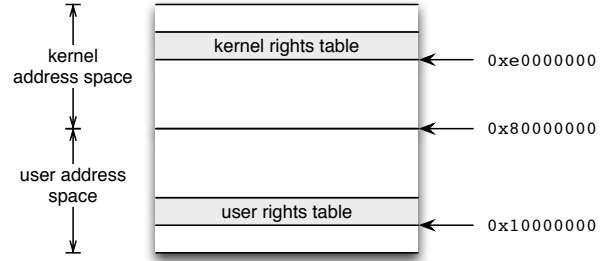


Figure 7: Kernel and user tables in an x86 Windows address space.

and a *user conflict table* per process. These tables are shared by all domains. The kernel and user tables are implemented as large arrays of drights to enable efficient access. The kernel table stores a dright for each 8-byte slot of virtual memory in kernel space and a user table stores a dright for each 8-byte slot of virtual memory in a process' user space to support direct access to user space (see Section 3).

These data structures are optimized for the case when there is a single dright associated with an 8-byte memory slot, i.e., when only one domain has an access right different from *read* to bytes in the slot and that domain has the same access right to all the bytes in the slot. These tables store a special dright value *conflict* in the entries for slots that do not satisfy these conditions. This value indicates that the drights for the bytes in the slot are stored in a conflict table. A conflict table is a splay tree that maps the address of a slot to a list of arrays with 8 drights. Each array in the list corresponds to a different domain and each dright in an array corresponds to a byte in the slot. The kernel conflict table is used for slots in kernel space and a process' user conflict table is used for slots in the process' user space.

Figure 7 shows the location of the user and kernel tables in the address space of an x86 Windows operating system. The conflict tables are allocated in kernel space and we modified process objects to include a pointer to their user conflict table. We also modified the kernel to reserve virtual address space for the kernel table at address 0xe0000000 when the system boots, and to reserve virtual address space in every process at address 0x10000000 when the process is created. Therefore, we have a single kernel table and a user table per process that is selected automatically by the virtual memory hardware. The kernel allocates physical pages to the tables on demand when they are first accessed and zeroes them to set access rights to *read* for all domains. This prevents incorrect accesses by default, for example, it protects the tables themselves from being overwritten. Since some extension code cannot take page faults, we also modified the kernel to preallocate physical pages to back kernel table entries that correspond to pinned virtual memory pages.

The same strategy could be used to implement BGI on the x64 architecture. Even though it is necessary to reserve a large amount of virtual memory for the tables in a 64-bit architecture, only top level page table entries need to be allocated to do this. Additional page meta-data and physical pages only need to be allocated to the tables on demand.

Figure 8 shows an example of how the tables are used to record ACLs. There are two untrusted domains in this example d_1 and d_2 . As shown in the diagram for system memory, d_1 has write access to the 10 bytes shaded light

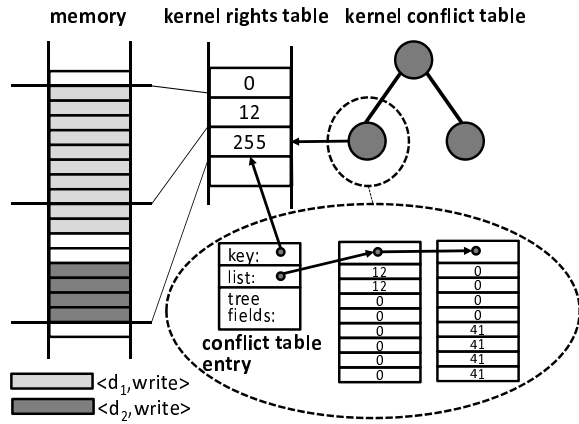


Figure 8: Example rights and data structures.

grey and d_2 has write access to the 4 bytes shaded dark grey. The figure also shows how these bytes are partitioned into slots and the corresponding entries in the kernel table. The first entry in the kernel table has dright zero because all the bytes in the corresponding slot have the default ACL $\langle d_1, read \rangle, \langle d_2, read \rangle$. The second entry has dright 12, which encodes $\langle d_1, write \rangle$, because all the bytes in the corresponding slot are writable by d_1 and no other untrusted domain has rights other than *read* to these bytes. The third entry has dright 255, which encodes a conflict, because both d_1 and d_2 have write access to some bytes in the slot. Since there is a conflict, there is an entry in the kernel conflict table keyed by a pointer to the third entry. The conflict table entry points to a list of arrays. The first array describes the access rights of d_1 and the second the access rights of d_2 . The first two entries in the first array have dright 12 because d_1 can write to the first two bytes in the slot, and the last four entries in the second array have dright 41 because d_2 can write to the last four bytes in the slot and 41 encodes $\langle d_2, write \rangle$ in this example. The other entries in the two arrays have dright zero because the corresponding bytes can only be read by the two domains.

6.3 Avoiding accesses to conflict tables

To achieve good performance both in space and in time, it is important to avoid accesses to the conflict tables. BGI uses several optimizations to ensure that most slots have a single associated dright.

First, BGI does not restrict read accesses. Therefore, supporting the common cases of read-read and read-write sharing across domains does not require accesses to conflict tables. We also observed that it is rare for two extensions to have non-*read* access rights to the same byte at the same time. In fact, we do not allow an untrusted domain to have a non-*read* right to a byte of memory that is writable by another untrusted domain. Similarly, different domains never have *icall* rights to the same function because they are only granted *icall* rights to functions in extensions they contain or to wrappers in the copies of the interposition library linked with those extensions. But it is possible for different extensions to have *type* rights to the same kernel object.

BGI also ensures that, for most slots, a domain has the same access right to all the bytes in the slot. The choice of an 8-byte slot is not a coincidence. We chose 8-byte slots because the dynamic memory allocators in x86 Windows

allocate 8-byte aligned memory in multiples of 8 bytes. The BGI compiler also changes the layout of local and global variables in the extension code: it aligns them on 4 byte boundaries and inserts pads around them. This allows BGI to grant write access to all the slots that overlap a variable location while maintaining guard slots before and after the variable. These guard slots are not writable and prevent sequential overflows and underflows as discussed previously. We borrowed this technique from WIT [2].

A naive implementation of *SetType* and *CheckType* would always access the kernel conflict table because they set and check access rights to the first byte of an object. BGI takes advantage of the fact that most objects are 8-byte aligned to implement these primitives efficiently. The optimized implementation of *SetType*(p, s, r) checks if p is 8-byte aligned and s is at least 8. If this check succeeds, it executes *SetRight*($p, 8, r$); *SetRight*($p+8, s-8, read$), which avoids the access to the conflict table. Otherwise, it executes *SetRight*($p, 1, r$); *SetRight*($p+1, s-1, read$) as before. Similarly, *CheckType*(p, r) checks if p is 8-byte aligned and the dright in the kernel table corresponds to access right r for the domain. Only if this check fails, does it access the conflict table to check if the byte pointed to by p has the appropriate dright. To further reduce the number of accesses to the conflict table, the BGI compiler aligns local variables and fields in local driver **structs** on 8-byte boundaries if they have a kernel object type. Functions are 16-byte aligned.

A final optimization avoids accesses to the conflict table while allowing a domain to have different access rights to the bytes in a slot in two common cases: when a domain has right *write* to the first half of the bytes in the slot and *read* to the second half of the bytes, and when a domain has right *read* to the first half of the bytes in the slot and *write* to the second half of the bytes. BGI uses two additional drights per domain to encode these cases. This is important to avoid accesses to conflict tables when a domain is granted write access to individual fields in a kernel object whose layout cannot be modified.

Our experiments show that these optimizations are very effective. Since most ACLs can be represented without using the conflict tables, ACLs can be modified and checked efficiently as we describe next and BGI introduces a space overhead of approximately 12.5%. It is interesting to point out that the space overhead could be reduced significantly on the x64 architecture because it imposes stricter alignment requirements than the x86. Since in x64 Windows dynamic memory allocators allocate 16-byte aligned memory in multiples of 16 bytes, pointers are 8-byte aligned, and stack variables are 8-byte aligned, we could use 16-byte memory slots without significantly increasing the use of the conflict tables. This should nearly halve the space overhead.

6.4 Table accesses

The optimizations described above enable efficient implementations of *SetRight* and *CheckRight*. In the common case, *SetRight* sets the access rights to all the bytes in a sequence of slots to *write* or *read*. Figure 9 shows an efficient code sequence that implements *SetRight*($p, 32, write$) in this case. In the examples in this section, the dright for the *write* right and the domain that runs the sample code sequences is 0x02. The dright values are immediates in the code. When the extension is added to a domain, these values must be changed to match the drights allocated to the domain. This


```

mov     eax,ebx
sar     eax,3
btc     eax,0x1C
mov     dword ptr [eax],0x02020202

```

Figure 9: Code sequence that implements *SetRight(p,32,write)* for the x86.

is similar to a relocation and requires the compiler to record the locations of these constants in the code.

Initially, pointer p is in register `ebx` and it can point either to kernel or to user space. The first instruction moves p into register `eax`. Then, the `sar` and `btc` instructions compute the address of the entry for the slot pointed to by p in either the kernel or the user table. They store this address in `eax`. This code sequence is interesting because it computes the address of the entry in the right table without checking if p points to kernel or user space and without using the base addresses of the tables. We refer to Figure 7 to explain how this works. Addresses in user space have the most significant bit set to 0 and addresses in kernel space have the most significant bit set to 1. The `sar` instruction shifts `eax` right by 3 bits and makes the 3 most significant bits equal to the most significant bit originally in `eax`. After executing `sar`, the four most significant bits in `eax` will be 1111 for a kernel address and 0000 for a user address. The `btc` instruction complements the least significant of these 4 bits. So the most significant 4 bits in the result are 0xe when p is a kernel address and 0x1 when it is a user address. The remaining bits in `eax` are the index of the table entry for the slot pointed to by p . The final `mov` instruction sets four entries in the table to 0x02, which grants the domain write access to $[p, p + 32)$.

We could use this code sequence on the x64 architecture by replacing 32-bit by 64-bit registers (because pointers are 64 bits long), shifting by 4 instead of 3 (because we would use 16-byte slots), and complementing a different bit (because the table bases would be at different addresses).

The BGI compiler inserts a code sequence similar to the one in Figure 9 in the prologues of instrumented functions to grant write access to local variables. The code sequence to revoke write access to local variables on function exit is more complicated because it must check if a local variable stores an active kernel object. We show an example in Figure 10. The code stores the address of the guard before the first local variable in `eax` (after saving `eax`) and the `sar` and `btc` instructions compute the address of the kernel table entry for the guard. The `add` instruction updates `eax` to point to the table entry right after the last table entry mod-

```

push     eax
lea      eax,[ebp-38h]
sar      eax,3
btc      eax,0x1C
add      eax,5
xor      dword ptr [eax-4],0x02020202
jne      L1
L2: pop   eax
...
ret      4
...
L1: push  eax
lea      eax,[ebp-38h]
push     eax
push     6
call     _BGI_slowRevokeAccess
jmp      L2

```

Figure 10: Code sequence that revokes access to local variables on function epilogues.

```

mov     eax,ebx
sar     eax,3
btc     eax,1Ch
cmp     byte ptr [eax],2
je      L1
push    ebx
call    _BGI_slowCheck1

```

L1:

Figure 11: Code sequence that implements *CheckRight(p,1,write)* for the x86.

ified by the `xor`. It adds 5 to `eax` to account for the guard slot before the local variable and the 4 slots occupied by the variable. If the local variable does not store any kernel object, the `xor` revokes access to the local variable and the branch is not taken. Otherwise, the branch is taken and the `_BGI_slowRevokeAccess` function is called. This function undoes the failed `xor` and checks if the table entries for the slots occupied by the local variables have drights corresponding to kernel objects. If it finds an active kernel object, it signals an error. When functions have more local variables, the compiler adds another `add`, `xor`, and `jne` for each variable.

The BGI compiler also inserts checks before writes and indirect calls. Figure 11 shows an efficient code sequence that implements *CheckRight(p,1,write)*. Initially, p is in `ebx`. The code computes the address of the table entry for the slot pointed to by p in the same way as *SetRight*. Then, the `cmp` instruction checks if the entry has dright 0x02. If the check fails, the code calls one of the `_BGI_slowCheck` functions. These functions receive a pointer to the memory range being checked and their name encodes the size of the range. In this case, the code calls `_BGI_slowCheck1`, which checks if the table entry contains a dright that encodes write access to the half slot being accessed and, if this fails, checks the dright for the appropriate byte in the conflict table. The indirect call check is similar but it also checks the number of stack bytes stored before the function and it does not have a slow path because functions are always 16-byte aligned.

The BGI compiler uses simple static analysis to eliminate *SetRight* and *CheckRight* sequences. It does not add *SetRight* for local variables that are not arrays or `structs` and whose address is not taken. It also eliminates *CheckRight* before the writes to these variables.

The interposition library implements *SetRight(p,s,r)* similarly to the compiler but uses `memset` to set drights when s is not known statically or is large. Additionally, it must deal with the case where p or $p+s$ are not 8-byte aligned. In this case, the interposition library sets drights as before for slots that are completely covered by the byte range but calls a function to deal with the remaining slots. This function sets the corresponding table entries to the drights that encode write access to half a slot when possible. Otherwise, it records drights for individual bytes in the appropriate conflict table. The interposition library implements *CheckRight* as in Figure 11 but it iterates the check for larger memory ranges. To improve performance, we implemented a function that compares four drights at a time when checking write access to large memory ranges.

6.5 Synchronization of table accesses

BGI avoids synchronization on table accesses as much as possible to achieve good performance. We use enough synchronization to ensure that there are no false positives (i.e., that we only signal an error when there is one) but we may fail to isolate errors in some uncommon schedules when there

are races. We believe it is hard for attackers to exploit these races to escape containment given that we assume they do not write the extension code.

We use no synchronization when granting or revoking write access to all the bytes in a slot, which is the common case. Synchronization is not necessary because: (1) it is an error for an untrusted domain to have non-*read* rights to a byte of memory that is writable by another untrusted domain and (2) there is a race in the code when threads running in the same domain attempt to set conflicting drights on the same byte. But we need synchronization when granting and revoking *type* rights and when granting and revoking write access to half a slot. We use atomic compare-and-swap on kernel or user table entries to prevent false positives in this case. Similarly, we need synchronization when granting or revoking rights involves an access to a conflict table. We use an atomic swap to record the conflict in the appropriate kernel or user table entry and we have a spin lock per conflict table. Since these tables are rarely used, contention for the lock is not a problem.

There is no synchronization in the fast path when checking rights. To prevent false positives, the slow path retries the check after a memory barrier and uses spin locks to synchronize accesses to the conflict tables when needed. The right checks are not atomic with the access they check. This can never lead to false positives because there is a race in the code when the right is revoked between the check and the access, but we may fail to prevent the access in some schedules if there is such a race.

7. RECOVERY

We prototyped a simple recovery mechanism for misbehaving domains [42, 46]. When an extension in a domain fails, BGI can unload the extensions in the domain, release all resources they hold, and then reload and restart them.

BGI uses Structured Exception Handling (SEH) to implement recovery: extension wrappers call the extension function within a *try* clause, BGI raises an exception when it detects a failure, and the *except* clause handles the exception by starting recovery (if not started already). If the domain is already recovering, the *except* clause simply returns an appropriate error code. Recovery support requires wrapping of additional extension functions to ensure there is an exception handler in the stack whenever extension code runs, e.g., functions in deferred procedure calls must be wrapped.

We avoid running extension code during recovery because domain state may already be corrupt [42, 46]. Extension wrappers return an appropriate error code when recovery is in progress without calling the extension function. Kernel wrappers check if the domain is recovering after the kernel function returns and they raise an exception if it is. This exception is handled by the *except* clause in an extension wrapper, which returns an appropriate error code.

BGI uses the Plug and Play (PnP) manager in the Windows kernel to unload and restart misbehaving domains. This simplifies recovery because the PnP manager deals with many of the synchronization issues necessary to unload a driver safely, but our current prototype only supports recovery of PnP drivers. When BGI starts recovery, it invokes the PnP manager to send a sequence of requests (IRPs) to the recovering extensions: the first IRP asks an extension if a device it manages can be removed and the second informs the extension that the device is being removed. The

extension wrappers for the functions that handle these IRPs perform driver-independent processing to advance the PnP state machine until the extension is unloaded. The extension wrapper that handles the second IRP also invokes a small device-specific function to reset the device hardware. We expect device vendors to provide these functions.

When there are no more references to the devices managed by an extension, the PnP manager invokes the extension's unload function. The extension wrapper for the last unload function called by the PnP manager releases resources held by the extensions in the domain being recovered. Since the drights in BGI's tables record all the information necessary to release these resources, there is no need for a separate object tracker (which can introduce a significant overhead [42]). The wrapper walks the BGI tables to find drights belonging to the domain. It frees heap memory and calls the appropriate functions to release different types of kernel objects, for example, it completes IRPs owned by the domain. Our prototype is not complete yet: there are some types of kernel objects that are not released by the wrapper. After the wrapper returns, the PnP manager reloads and restarts the extensions in the domain.

8. EVALUATION

We ran fault-injection experiments to evaluate BGI's ability to isolate extensions and we measured the overhead introduced by BGI. The results show that BGI provides strong isolation guarantees with low overhead. We also found new bugs by running existing driver tests with BGI. This section describes these experiments.

We used the kernel extensions listed in Table 1 to evaluate BGI. **classpnp** implements common functionality required by storage class drivers like **disk**. The last five drivers sit at the bottom of the USB driver stack in Windows Vista. **usbhcci**, **usbuhci**, **usbohci**, and **usbwhci** implement different USB host controller interfaces and **usbport** implements common functionality shared by these drivers. These 16 extensions have a total of more than 400,000 lines of code and use 350 different functions from WDM, IFS, NDIS, and KMDF interfaces [31]. The source code for the first 8 extensions is available in the Windows Driver Kit [31].

All experiments ran on HP xw4600 workstations with an Intel Core2 Duo CPU at 2.66 GHz and 4GB of RAM, running Windows Vista Enterprise SP1. The workstations were connected with a Buffalo LSW100 100 Mbps switching hub for the Intel PRO/100 driver tests and with an HP 10-GbE CX4 cable for the Neterion Xframe driver tests. Experiments with the FAT file system and disk drivers ran on a freshly formatted Seagate Barracuda ST3160828AS 160GB disk. Experiments with the USB drivers used a SanDisk 2GB USB2.0 Flash drive. The extensions were compiled, with and without BGI instrumentation, using the Phoenix [29] compiler with the -O2 (maximize speed) option. The experiments ran without recovery support except where noted. All the performance results are averages of at least three runs.

8.1 Effectiveness

We injected faults into the **fat** and **intelpro** drivers to measure BGI's effectiveness at detecting faults before they propagate outside a domain.

We used a methodology similar to the one described in [46]. We injected bugs from the five types in Table 2 into the

Extension	#lines	Description
xframe	39,952	Neterion Xframe 10Gb/s Ethernet driver
fat	68,409	FAT file system
disk	13,330	Disk class driver
classpnp	27,032	Library for storage class drivers
intelpro	21,153	Intel PRO 100Mb/s Ethernet driver
kmdf	2,646	Benchmark: store/retrieve buffer
ramdisk	1,023	RAM disk driver
serial	19,466	Serial port driver
rawether	5,523	Ethernet packet capture driver
topology	7,560	Network topology discovery driver
usbhub	74,921	USB hub driver
usbport	77,367	USB host controller port driver
usbhci	22,133	USB EHCI miniport driver
usbuhci	8,730	USB UHCI miniport driver
usbhci	8,218	USB OHCI miniport driver
usbwhci	5,191	USB WHCI miniport driver

Table 1: Kernel extensions used to evaluate BGI.

source code of the drivers. Empirical evidence indicates that these types of bugs are common in operating systems [8, 40]. The random increment to loop upper bounds and to the number of bytes copied was 8 with 50% probability, 8 to 1K with 44% probability and 1K to 2K with 6% probability.

We produced buggy drivers by choosing a bug type and injecting five bugs of that type in random locations in the driver source. Then we tested the buggy drivers that compiled successfully without BGI. We tested 304 buggy **fat** drivers by formatting a disk, copying two files onto it, checking the file contents, running the Postmark benchmark [23], and finally running **chkdsk**. We tested 371 buggy **intelpro** drivers by downloading a large file over **http** and checking its contents.

There were 173 **fat** tests and 256 **intelpro** tests that failed without BGI. There were different types of failure: blue screen inside driver, blue screen outside driver, operating system hang, test program hang, and test program failure. A blue screen is inside the driver if the faulting thread was executing driver code. We say injected faults *escape* when the test ends with a blue screen outside the driver or an operating system hang. These are faults that we know affected the rest of the system. We say injected faults are *internal* when the test ends with one of the other types of failure.

We isolated buggy drivers with escaping faults in a separate BGI domain and repeated the tests. Table 3 reports the number of faults that BGI was able to contain before they caused an operating system hang or a blue screen outside the driver. We investigated the faults that caused hangs and found infinite loops and resource leaks. These faults affect the rest of the system not by corrupting memory but by consuming an excessive amount of CPU or other resources. It is surprising that BGI can contain 60% of the hangs in **fat** and 47% in **intelpro** because it does not include explicit checks to contain this type of faults. BGI is able to contain some hangs because it can detect some internal driver errors before they cause the hang, for example, it can prevent buffer overflows from overwriting a loop control variable. We could improve containment by modifying extension wrap-

Fault type	Description
flip if condition	swap the “then” and “else” cases
lengthen loop	increase loop upper bounds
larger memcpy	increase number of bytes copied
off by one	replace “>” with “>=” or similar
delete assignment	remove an assignment statement

Table 2: Types of faults injected in extensions.

driver	type	contained	not contained
fat	blue screen	45 (100%)	0
	hang	3 (60%)	2
intelpro	blue screen	116 (98%)	2
	hang	14 (47%)	16

Table 3: Number of injected faults contained.

pers to impose upper bounds on the time to execute driver functions, and by modifying the kernel wrappers that allocate and deallocate resources to impose an upper bound on resources allocated to the driver [35, 44].

BGI can contain more than 98% of the faults before they cause a blue screen outside the drivers. These faults corrupt state: they violate some assumption about the state in code outside the driver. These are the faults that BGI was designed to contain and they account for 90% of the escaping faults in **fat** and 80% in **intelpro**. BGI can contain a number of interesting faults, for example, three buggy drivers complete IRPs incorrectly. This bug is particularly hard to debug without isolation because it can corrupt an IRP that is now being used by the kernel or a different driver. BGI contains these faults and pinpoints the call to **IoCompleteRequest** that causes the problem.

We investigated the two faults that cause a blue screen outside the **intelpro** driver. One is due to a bug in another driver in the stack that is triggered by the modified behavior in this variant of **intelpro**. We were unable to understand the other fault.

Next, we ran experiments to evaluate recovery with the **intelpro** driver. We did not run recovery experiments with **fat** because, currently, BGI can recover only PnP drivers. We selected 50 buggy drivers at random from the set of buggy drivers with escaping blue screens that BGI can contain. We loaded them into a BGI domain with recovery support and then activated the injected faults. We recovered the 21 drivers that raised a BGI exception while attempting to download a large file. After recovery, we loaded the version of the driver without injected faults and checked if the recovered driver was able to download a large file. BGI recovered all these drivers successfully except two (which failed to recover due to limitations in the current prototype).

We also ran experiments to evaluate BGI’s ability to detect internal faults. We isolated buggy drivers with internal faults in a separate BGI domain and repeated the tests. Table 4 shows the number of internal errors that BGI can detect before they are detected by checks in the test program or checks in the driver code. The results show that BGI can simplify debugging by detecting many internal errors early.

8.2 Performance

We also measured the overhead introduced by BGI. For the disk, file system, and USB drivers, we used the PostMark [23] file system benchmark that simulates the workload of an email server. For **fat**, we configured PostMark to use cached I/O with 10,000 files and 1 million transactions. For the other drivers, we used synchronous I/O with 100 files and 10,000 transactions. We disabled caching for these drivers because, otherwise, most I/O is serviced from the cache masking the BGI overhead.

driver	detected	not detected
fat	54 (44%)	68
intelpro	36 (33%)	72

Table 4: Number of internal faults detected by BGI.

We ran **disk** and **classnp** in the same protection domain because **classnp** provides services to **disk**. This is common in Windows: a port driver implements functionality common to several miniport drivers to simplify their implementation. Similarly, we ran **usbport** and **usbehci** in the same domain. We tested the other drivers separately.

We measured throughput in transactions per second (Tx/s) as reported by PostMark, and we measured kernel CPU time with the kernrate kernel profiler [31]. Table 5 shows the percentage difference in kernel CPU time and throughput. BGI increases kernel CPU time by a maximum of 10% in these experiments. The throughput degradation is negligible in the 4 test cases that use synchronous I/O because the benchmark is I/O bound. For **fat**, the benchmark is CPU bound and throughput decreases by only 12%.

These results are significantly better than those reported for Nooks [42]. Nooks increased kernel time by 185% in a FAT file system benchmark. BGI performs better because it does not change page tables or copy objects in cross domain calls. Additionally, the BGI tables that store ACLs keep track of the information needed to recover domains. Nooks required a separate object tracking mechanism that added a significant overhead.

Next, we measured the overhead introduced by BGI to isolate the network card drivers. For our TCP tests, we used socket buffers of 256KB and 32KB messages with **intelpro** and socket buffers of 1MB and 64KB messages with **xframe**. We enabled IP and TCP checksum offloading with **xframe**. We used 16-byte packets for our UDP tests with both drivers. We measured throughput with the `ttcp` utility and we measured kernel CPU time with `kernrate`. These tests are similar to those used to evaluate SafeDrive [46]. Table 6 shows the percentage difference in kernel CPU time and throughput due to BGI.

The results show that isolating the drivers with BGI has little impact on throughput. There is almost no throughput degradation with **intelpro** because this is a driver for a slow 100Mb/s Ethernet card. There is a maximum degradation of 10% with **xframe**. BGI reduces UDP throughput with **xframe** by less than SafeDrive [46]: 10% versus 11% for sends and 0.2% versus 17% for receives. But it reduces TCP throughput by more than SafeDrive: 2.5% versus 1.1% for sends and 3.7% versus 1.3% for receives. We should note that the SafeDrive results were obtained with a Broadcom Tigon3 1Gb/s card while **xframe** is a driver for a faster 10Gb/s Neterion Xframe E.

The average CPU overhead introduced by BGI across the network benchmarks is 8% and the maximum is 16%. For comparison, Nooks [42] introduced a CPU overhead of 108% on TCP sends and 46% on TCP receives using similar benchmarks with 1Gb/s Ethernet. Nexus [44] introduced a CPU overhead of 137% when streaming a video using an isolated 1Gb/s Ethernet driver. Nexus runs drivers in user space, which increases the cost of cross domain switches. BGI’s CPU overhead is similar to the CPU overhead reported for

Driver	Δ CPU(%)	Δ Tx/s(%)
disk+classnp	2.88	-1.37
ramdisk	0.00	0.00
fat	10.01	-12.31
usbport+usbehci	0.91	0.00
usbhub	4.20	0.00

Table 5: Overhead of BGI on disk, file system, and USB drivers, when running PostMark.

Driver	Benchmark	Δ CPU(%)	Δ bps(%)
xframe	TCP Send	11.94	-2.53
	TCP Recv	3.12	-3.65
	UDP Send	16.06	-10.26
	UDP Recv	6.86	-0.18
intelpro	TCP Send	13.57	0.00
	TCP Recv	11.69	0.00
	UDP Send	-1.74	-0.41
	UDP Recv	3.89	-0.84

Table 6: Overhead of BGI on network device drivers.

SafeDrive in the network benchmarks [46] but BGI provides stronger isolation.

Finally, we used the **kmddf** benchmark driver to compare the performance of BGI and XFI [11]. We measured the number of transactions per second for different buffer sizes. In each transaction, a user program stores and retrieves a buffer from the driver. BGI and XFI have similar performance in this benchmark but XFI cannot provide strong isolation guarantees for drivers that use the WDM, NDIS, or IFS interfaces (as we discussed in Section 2). XFI was designed to isolate drivers that use the new KMDF interfaces. KMDF is a library that simplifies development of Windows drivers. KMDF drivers use the simpler interfaces provided by the library and the library uses WDM interfaces to communicate with the kernel. However, many KMDF drivers have code that uses WDM interfaces directly because KMDF does not implement all the functionality in the WDM interfaces. For example, the KMDF driver **serial** writes to some fields in IRPs directly and calls WDM functions that manipulate IRPs. Therefore, it is unclear whether XFI can provide strong isolation guarantees for real KMDF drivers.

Our experimental results show that the overhead introduced by BGI is low. We believe it could be used in practice to isolate Windows drivers in production systems.

8.3 Real bugs

In our last experiment, we tested 6 of the extensions with BGI. We used existing test suites that achieve good code coverage. BGI found 28 new bugs in these widely used Windows extensions. Table 8 shows the different types of bugs found by BGI.

BGI found 3 cases where an event object is reinitialized. These bugs can corrupt the list of threads waiting on the event, which can lead to hangs and corruption elsewhere in memory. BGI detected these bugs because the wrapper for the event initialization function checks if the extension has write access to the event and then revokes write access to disallow further writes. BGI also found 4 cases where an event is passed to a kernel function without being initialized. These were detected because the kernel wrappers that receive events check if the extension has the appropriate *type* right, which is granted by the event initialization function.

There were 5 incorrect uses of functions that manipulate linked lists, for example, reinitializing a list head, removing an entry from a list twice, and freeing an entry that is still in a list. These bugs can also lead to memory corruption and hangs. BGI detects these bugs using checks similar to

Buffer size	Δ Tx/s(%)
1	-8.76
512	-7.08
4K	-2.48
64K	-1.14

Table 7: Overhead of BGI on the kmddf driver.

Bug type	Count
reinitialization of event	3
use of invalid event	4
incorrect use of list interface	5
write to invalid device extension	5
use of invalid device object	1
failure to uninitialize object	2
null pointer dereference	2
abstraction violation	6
Total	28

Table 8: Real bugs found while testing kernel extensions isolated using BGI.

those used for events. Whereas removing an entry from a list twice can be found using simple assertions in the list functions, the other errors cannot.

The write checks inserted by the compiler found 5 writes to invalid device extension objects. In four cases the objects had been deleted and in one the object had not been allocated. BGI also found one case where a deleted device object is passed to a kernel function. The type checks in kernel wrappers find these bugs because the *type* right is revoked when an object is deleted.

BGI found two bugs where an extension failed to uninitialize a kernel object before freeing the heap block where the object was stored. This can lead to resource leaks and memory corruption if the kernel uses the object after the memory is reallocated. The bugs were found by the check for active objects when heap memory is freed.

BGI also found 2 cases where extensions passed *null* pointers to the kernel. These bugs were found by the write checks in the kernel wrappers for these functions. It would be easy to find these two bugs with simple assertions.

Finally, the write checks inserted by the compiler and the type checks in kernel wrappers found 6 cases of abstraction violation. These are bugs where the extension writes directly to object fields that should be modified only by kernel functions. For example, writing to flags that encode the state of an object or inserting a list entry by hand. These bugs do not result in incorrect behavior while the kernel does not change but may cause subtle problems when a new version of the kernel is released. They were immediately fixed after we reported them to the developers.

These results show that BGI is also a good bug-finding tool. It can find incorrect uses of the kernel interface in drivers that have been extensively tested and analyzed using dynamic tools like driver verifier [31] and static tools like static driver verifier and prefast [31]. Since BGI has low overhead, it can be used not only for pre-release testing but also to collect better information about bugs in customer machines than is currently available to error reporting tools like Microsoft’s Windows Error Reporting [15].

9. RELATED WORK

Many systems proposed techniques to isolate faults in kernel extensions. Some systems isolate extensions by running them in user space (e.g., [12, 18, 19, 26]) but this introduces high overhead. Microdrivers [13] improve performance by splitting a driver into a kernel and a user component with help from programmer annotations, but they do not isolate the kernel component. Nexus [44] can enforce safety properties beyond what previous user-level driver systems provide but it requires additional device-specific safety specifi-

cations. Extensions can also be isolated on virtual machines (e.g., [4, 34]) by running a dedicated operating system with the extension in a separate virtual machine (e.g., [27, 39]), but this also has high overhead.

Other systems isolate extensions while running them in the kernel address space. Nooks [41, 42] was the first system to provide isolation of existing extensions for a commodity operating system, but it has high overhead because it uses hardware page protection. Mondrix [45] uses fine-grained hardware memory protection to isolate kernel extensions with low overhead, but it requires special hardware, does not check for incorrect uses of the extension interface, and it only supports 4 byte granularity (which we found insufficient without major changes to the operating system).

Software fault isolation techniques like SFI [43], Pittsfield [28] and XFI [11] can isolate kernel extensions with low overhead but they do not deal with the complex extension interfaces of commodity operating systems as we discussed in Section 2. Other software fault isolation techniques have similar problems as discussed in [36]. Vino [35] describes a new operating system designed to deal with misbehaving kernel extensions.

We also discussed SafeDrive [46] in Section 2. It is an efficient software fault-isolation technique but it provides weaker isolation than BGI, e.g., it does not provide protection from temporal errors. It also requires programmers to annotate the extensions. The Ivy project aims to combine SafeDrive with Shoal [3] and HeapSafe [14], which should provide improved isolation when completed. However, this combination would still be missing BGI’s dynamic typestate checking and it would likely perform worse than BGI because both Shoal and HeapSafe can introduce a large overhead.

SVA [10] can enforce some safety properties for commodity systems, but it does not isolate extensions and it incurs higher overhead than BGI. Writing extensions in type safe languages [5, 6, 20, 22, 33] can provide strong isolation guarantees but it requires rewriting the extensions and using a different runtime system. Proving that extensions are correct makes runtime checks to enforce isolation unnecessary (e.g., [32]) but is hard to achieve for complex extensions.

10. CONCLUSION

Faults in extensions continue to be the primary cause of unreliability in commodity operating systems. Previous fault isolation techniques are not widely used because they cannot isolate existing kernel extensions with low overhead on standard hardware. We presented BGI, a new software fault isolation technique that addresses this problem using efficient byte-granularity memory protection. We applied BGI to 16 extensions that use the complex extension interface of the Windows operating system. Our results show that BGI incurs only a 6.4% CPU overhead on average. Therefore, we believe BGI can be used to isolate existing kernel extensions in production systems. The results also show that BGI can be used effectively as a bug-finding tool: we found 28 new bugs in widely used Windows extensions.

Acknowledgments

We thank Solom Heddaya for encouraging us to pursue this work. We had many interesting discussions with Solom, Jonathan Morrison, Gretchen Lohle, Neil Clift, Nar Ganapathy, John Lee, Landy Wang, Daniel Mihai, Matt Thom-

linson, and Tim Burrell. We thank Jonathan Morrison for evaluating BGI by developing a driver and injecting common faults into it. Nar helped us understand the semantics of WDM interfaces. We thank Martin Borve, Glen Slick, Robbie Harris, and Tomas Perez-Rodriguez for helping us test USB drivers isolated with BGI. Gloria Mainar-Ruiz helped developing BGI. Andy Ayers, Chris McKinsey, and Matt Moore answered many questions about Phoenix. We also thank our shepherd Andrew Myers and the anonymous reviewers for comments on earlier drafts of this paper.

11. REFERENCES

- [1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *ACM CCS*, 2005.
- [2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*, 2008.
- [3] Z. Anderson, D. Gay, and M. Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *PLDI*, 2009.
- [4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.
- [5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, 1995.
- [6] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *OPENARCH*, 2002.
- [7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *SOSP*, 2001.
- [8] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults - based on field data. In *FTCS*, 1996.
- [9] P. Chubb. Get more device drivers out of the kernel! In *Linux Symposium*, 2004.
- [10] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *SOSP*, 2007.
- [11] U. Erlingsson, M. Abadi, M. Vrabie, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *OSDI*, 2006.
- [12] A. Forin, D. Golub, and B. Bershad. An I/O system for Mach 3.0. In *Proc. USENIX Mach Symposium*, 1991.
- [13] V. Ganapathy, M. Renzelmann, A. Balakrishnan, M. Swift, and S. Jha. The Design and Implementation of Microdrivers. 2008.
- [14] D. Gay, R. Ennals, and E. Brewer. Safe manual memory management. In *ISMM*, 2007.
- [15] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *SOSP*, 2009.
- [16] L. H. Linux Kernel Heap Tampering Detection. *Phrack*, 13(66), 2009.
- [17] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.
- [18] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ -kernel-based systems. In *SOSP*, 1997.
- [19] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: a highly reliable, self-repairing operating system. *SIGOPS OSR*, 40(3):80–89, 2006.
- [20] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS OSR*, 41(2):37–49, 2007.
- [21] A. Ionescu. Pointers and Handles: A Story of Unchecked Assumptions in the Windows Kernel. In *Black Hat*, 2008.
- [22] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, 2002.
- [23] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997.
- [24] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution via Program Shepherd. In *USENIX Security Symposium*, 2002.
- [25] K. Kortchinsky. Real World Kernel Pool Exploitation. In *SyScan'08 Hong Kong*, 2008.
- [26] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5), 2005.
- [27] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *OSDI*, 2004.
- [28] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security Symposium*, 2006.
- [29] Microsoft. Phoenix SDK. <http://connect.microsoft.com/Phoenix>.
- [30] Microsoft. User-Mode Driver Framework. <http://www.microsoft.com/whdc/driver/wdf/UMDF.msp>.
- [31] Microsoft. Windows Driver Kit. <http://www.microsoft.com/wdk>.
- [32] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, 1996.
- [33] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. *SIGPLAN Not.*, 37(1):128–139, 2002.
- [34] L. Seawright and R. MacKinnon. VM/370—A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.
- [35] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *OSDI*, 1996.
- [36] C. Small and M. Seltzer. MiSFIT: A tool for constructing safe extensible C++ systems. *IEEE Concurrency*, 6(3):34–41, 1998.
- [37] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.
- [38] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1), 1986.
- [39] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, 2001.
- [40] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. In *FTCS*, 1991.
- [41] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM TOCS*, 24(4):333–360, 2006.
- [42] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM TOCS*, 23(1):77–110, 2005.
- [43] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.
- [44] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, 2008.
- [45] E. Witchel, J. Rhee, and K. Asanović. Mondrix: memory isolation for Linux using mondriaan memory protection. In *SOSP*, 2005.
- [46] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.