Fast Byte-Granularity Software Fault Isolation

Manuel Costa
Microsoft Research, Cambridge

Joint work with:

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

The problem with drivers

- operating systems run many drivers
- (most) drivers are fully trusted
  - run in the kernel, can write anywhere
- driver bugs cause serious problems
  - data corruption, crashes, security breaches

driver bugs are a major cause of unreliability in operating systems

Isolating existing drivers is hard

- previous solutions are not enough
  - user level drivers, hardware fault isolation [Nooks], software fault isolation [SFI, …], safe languages
  - require changes to driver code, or hardware, or have poor performance, or provide weak isolation
- because drivers use a complex kernel API
  - fine-grained spatial and temporal sharing
  - incorrect use may cause writes anywhere
  - or execution of arbitrary code

BGI: Byte-Granularity Isolation

- isolates drivers in separate protection domains
  - allows domains to share the same address space
- uses byte-granularity memory protection
  - with several types of memory access rights
  - can grant/revoke access precisely and quickly
  - can check accesses efficiently

BGI properties

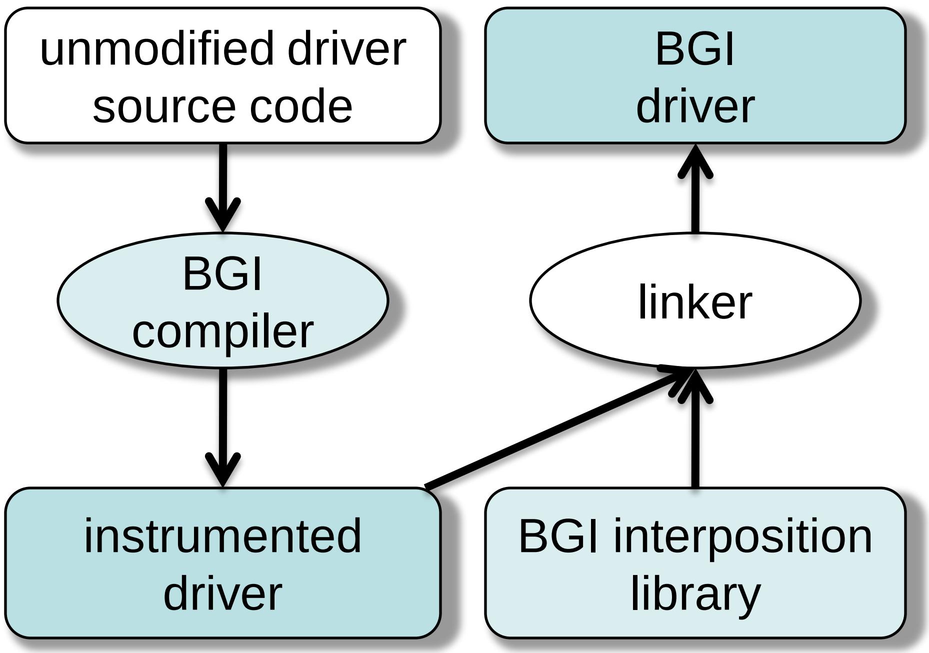
- strong isolation even with complex kernel API
  - **write integrity**: prevents bad writes
  - **control-flow integrity**: prevents bad control flow
  - **type safety for kernel objects**: correct use of API
- without changes to drivers or hardware
- with low overhead
  - average increase in CPU usage: 6.4%
  - space overhead: ~12.5%

Outline

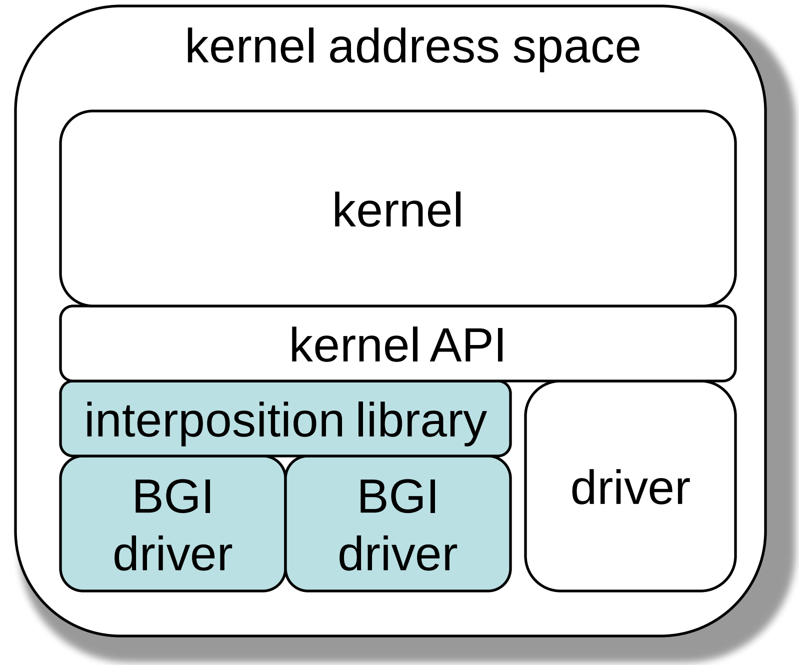
- introduction
- **overview**
- protection model
- implementation
- results

Overview

Building a BGI driver



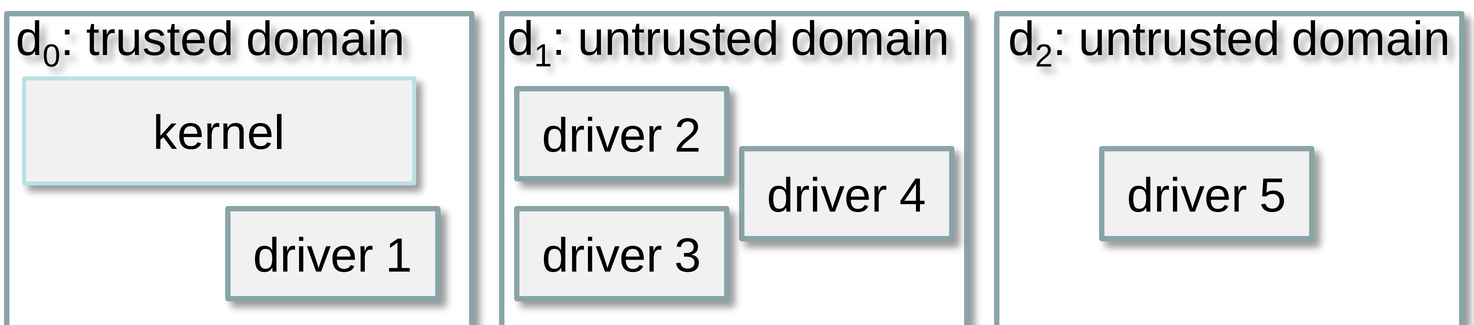
Running a BGI driver



- BGI drivers share the kernel address space
- compiler/interposition library enforce protection model

Protection model: domains and ACLs

- kernel runs in a trusted protection domain
  - some drivers may share this domain
- most drivers run in untrusted domains
  - drivers may share untrusted domains
- each byte of virtual memory has an ACL
- ACL lists domains and access rights



Protection model: access rights

- different types of memory access rights
  - **read** is the default, allows reads
    - BGI does not restrict read accesses
  - **write** allows reads and writes
  - **icall** allows indirect calls and reads
  - type rights for different types of kernel objects:
    - **io**, **dispatcher**, **mdl**, **mutex**, …
    - allow driver to pass object in calls expecting type
  - **ownership** rights: allow driver to free memory

Primitives to manipulate rights

- **`CheckRight(p,s,r)`**
  - checks if driver has right **r** for bytes [**p**,**p**+**s**[
- **`SetRight(p,s,r)`**
  - changes ACLs of bytes [**p**,**p**+**s**[to grant **r** to driver
  - SetRight(p, s, **read**) revokes right

- called by the BGI interposition library
- or inserted by the BGI compiler
- driver cannot call these primitives directly

Dynamic type checking

- **`SetType(p,s,r)`**
  - marks **p** as the start of an object of type **r**
  - prevents writes to the object
  - like SetRight(p,1,r); SetRight(p+1,s-1,**read**)
- **`CheckType(p,r)`**
  - checks if p is the start of an object of type r
  - like CheckRight(p,1,r) but more efficient
- check arguments used in kernel API calls
  - a form of dynamic typestate checking
  - objects can be used if they have the correct type
  - type is set when objects are in an appropriate state

BGI compiler uses primitives

- compiler adds instrumentation to:
  - check rights on writes and indirect calls
  - grant and revoke write access to stack
- records list of address-taken functions
  - to be used on driver load

Interposition library uses primitives

- grant rights on driver load
  - grant write access to globals
  - grant icall right for address-taken functions
- grant/revoke rights to function arguments
  - grant type rights to received objects, grant write access to fields that can be written
  - revoke rights according to call semantics
- check rights for function arguments
  - check if arguments have the correct type
  - check if arguments can be written

Example: read request

```
void ProcessRead(IRP *irp
   KEVENT e;
   KeInitializeEvent(&e);
```

allocate a kernel event object on the driver's stack

initialize event object

```
   KeWaitForSingleObject(&e);
```

wait on event object

```
   {
      irp->Buffer[j] ^= key;
   }
   IoCompleteRequest(irp);
}
```

Example: read request

```
void ProcessRead(IRP *irp) {
   KEVENT e;
   KeInitializeEvent(&e);
```

```
_bgi_KeInitializeEvent(PRKEVENT e)
{
   CheckRight(e,sizeof(KEVENT),write);
   SetType(e,sizeof(KEVENT),event);
   KeInitializeEvent(e);
}
```

```
   }
   IoCompleteRequest(irp);
}
```

Example: read request

```
void ProcessRead(IRP *irp) {
   KEVENT e;
   KeInitializeEvent(&e);

   IoSetCompletionRoutine(irp,&ReadDone,&e)
   IoCallDriver(diskDevice,irp);
   KeWaitForSingleObject(&e);
}
```

```
_bgi_KeWaitForSingleObject(PRKEVENT e)
{
  CheckType(e,event);
  KeWaitForSingleObject(e);
}
```

Example: read request

```
void ProcessRead(IRP *irp) {
   KEVENT e;
   KeInitializeEvent(&e);

   IoSetCompletionRoutine(irp,&ReadDone,&e)
   IoCallDriver(diskDevice,irp);
```

- BGI compiler inserts inline check before write:
  `CheckRight(p,4,write)`

```
   {
      irp->Buffer[j] ^= key;
   }
   IoCompleteRequest(irp);
}
```

Dynamic typestate checking

- rights change as driver calls kernel functions
- BGI enforces complex kernel API usage rules
  - should not use **e** before it is initialized
  - should not write to **e** after it is initialized but
  - can pass **e** in kernel API calls that expect events

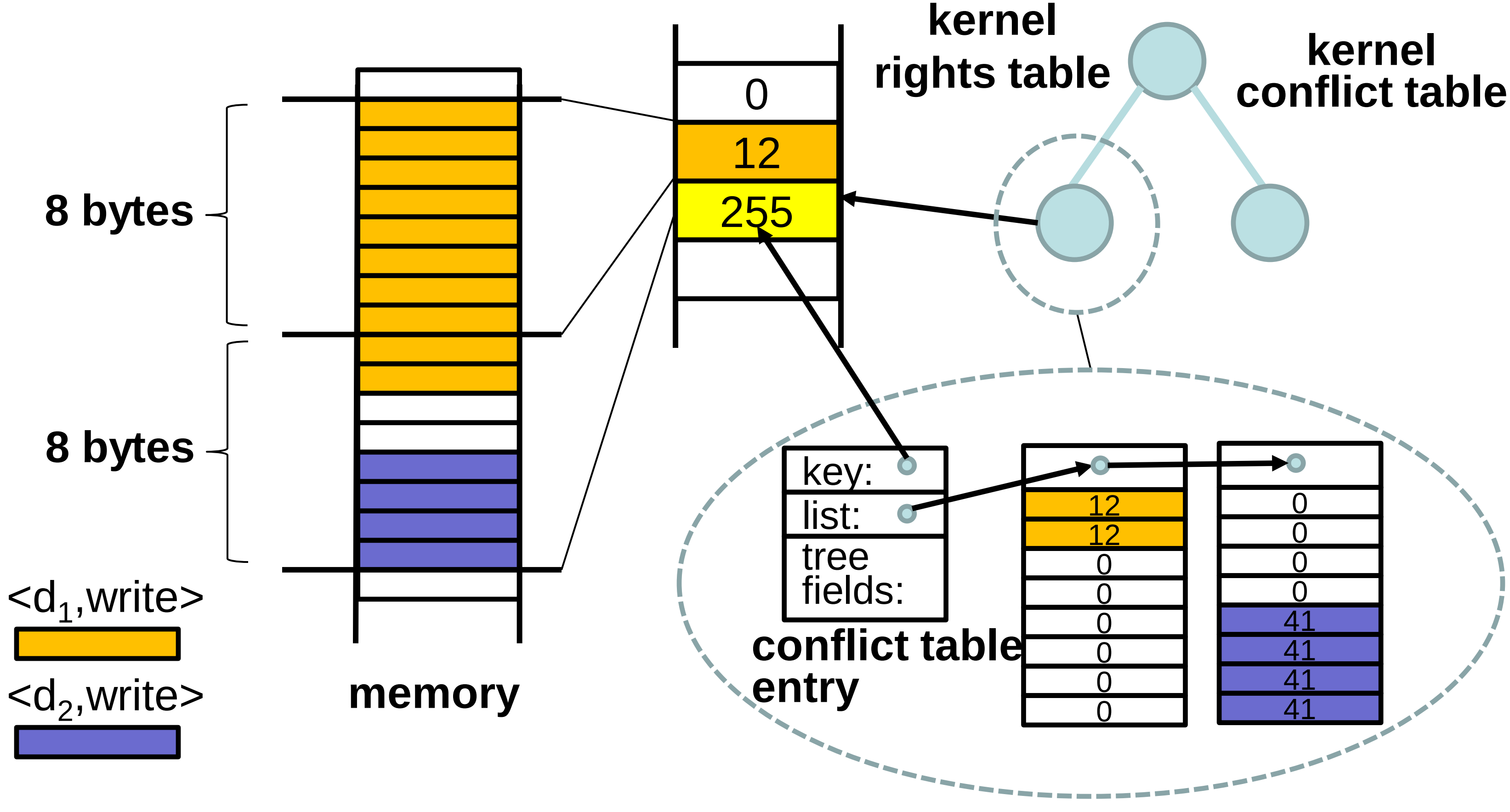
Implementation needs to be fast

- fast crossing of protection domains
  - no changes of page protections, no separate heaps or stacks, no copies of objects
- fast granting/revoking/checking of rights
  - fast data structures, fast code sequences
  - compiler support: inlined checks, data alignment
- careful choice of checks/guarantees
  - BGI does not restrict reads

ACL data structures

- **(domain,right)** pairs encoded as **dright (**integer)
- BGI uses several drights tables:
  - one *kernel-table* to cover kernel address space
  - one *user-table* per process user address space
  - tables are arrays for efficient access
  - 1-byte dright for each 8-byte memory slot
  - optimized for: all 8 bytes in slot have same ACL, ACL has one element
- extra *conflict-tables* handle general case
  - rarely used, splay tree with list of arrays of 8 drights

Rights tables



Avoiding accesses to conflict tables

- BGI compiler aligns data
  - compiler lays out locals/globals in 8-byte slots
  - 8-byte aligns fields in driver-local structs
- heap objects are 8-byte aligned
- special drights for writable half-slots (4 bytes)
- BGI does not restrict read access
  - more likely that ACLs have a single element

Fast code sequence: SetRight

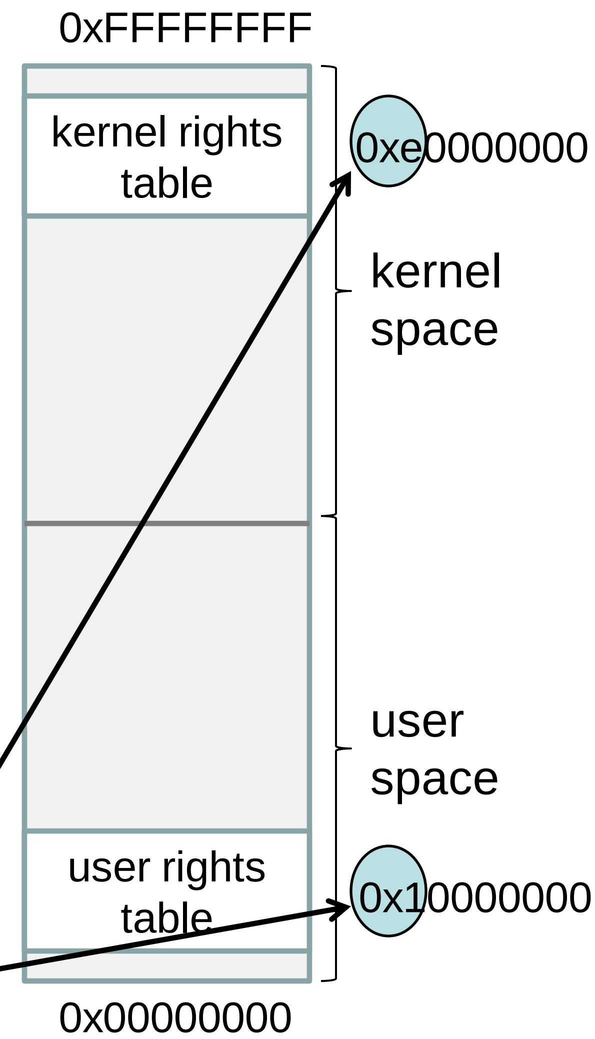
- **`SetRight(p,32,write)`** implemented as:

```
mov  eax, p
sar  eax, 3
btc  eax,0x1C
mov dword ptr [eax], 0x12121212
```

compute rights table entry

store dright

address space	p's 4 most signifcant bits	after sar eax,3	after btc eax,0x1C
kernel	1XXX	1111	1110 (0xe)
user	0XXX	0000	0001 (0x1)



Fast code sequence: CheckRight

- **`CheckRight(p,1,write)`** implemented as:

```
mov  eax, ebx
sar  eax, 3
btc  eax,0x1C
cmp byte ptr [eax],12
je L1
push ebx
call CheckConflictTable
L1:mov byte ptr [ebx],48
```

compute rights table entry

fast check

slow check

- check and access are not atomic
  - improved performance with little coverage loss

Recovery

- BGI has a recovery mechanism [Nooks,…]
- driver code runs inside a try block
- when a check fails, raise an exception
  - driver stops servicing requests
  - scan rights tables to release resources
  - unload driver
  - restart driver

Evaluation

- tested 16 Windows Vista device drivers
  - mix of complex kernel APIs (WDM, WDF, NDIS)
  - including network, disk, USB, and file system
  - more than 400,000 lines of code
- measured ability to contain faults
- measured performance overhead

Fault containment

- injected faults in the source of **fat** and **intelpro**
  - fault types follow previous bug studies
- measured BGI’s ability to contain faults that
  - cause a crash outside the driver

driver	contained	not contained
fat	45 (100%)	0
intelpro	116 (98%)	2

- BGI contained 161 out of 163 faults

Performance: file IO

- ran the Postmark file system benchmark
- measured throughput (Tx/s) and CPU time

driver	increase in kernel CPU time	decrease in throughput
disk+classpnp	2.8%	1.4%
ramdisk	0.0%	0.0%
fat	10.0%	12.3%
usbport+usbehci	0.9%	0.0%
usbhub	4.2%	0.0%

- low overhead in all cases

Performance: network

- 10 Gbps Neterion Xframe card
- measured throughput and CPU time with ttcp

	increase in kernel CPU time	decrease in throughput
TCP send	11.9%	2.5%
TCP receive	3.1%	3.6%
UDP send	16.0%	10.2%
UDP receive	6.8%	0.1%

- low overhead in all cases

New bugs found

- BGI found 28 new bugs in widely used drivers

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**

- BGI is also a good bug finding tool

Conclusion

- BGI improves reliability and security
  - isolates existing drivers with low overhead
  - can find bugs during testing/diagnostics
  - can contain faults during production
    - prevent system corruption
    - prevent attackers from “0wning” host
    - can recover faulty domain