PathArmor: Practical Context-Sensitive CFI

Dennis Andriesse[†‡], **Victor van der Veen**[†‡],
Enes Göktaş[‡], Ben Gras[‡], Lionel Sambuc[‡], Asia Slowinska[§],
Herbert Bos[‡], Cristiano Giuffrida[‡]
[†]*Joint first authorship*

[‡]Vrije Universiteit Amsterdam
[§]Lastline, Inc.

CCS 2015

Introduction

Control-Flow Integrity

- CFI introduced over 10 years ago (Abadi et al.)
- Still struggling to balance security vs. performance!

Context-Sensitive CFI

- Context-Insensitive CFI ($\overline{\mathrm{C}}$CFI) enforces valid target *per edge*
- $\overline{\mathrm{C}}$CFI exploitable, e.g. call-site gadgets and entry-point gadgets
- Context-Sensitive CFI (CCFI) considers *context of prior edges*
- CCFI proposed in original CFI paper, dismissed as impractical
- We implement CCFI efficiently on commodity hardware

```c
{ channel_pre[SSH_CHANNEL_OPEN]    = &channel_pre_open_13;
  channel_pre[SSH_CHANNEL_DYNAMIC] = &channel_pre_dynamic;
```

```c
  channel_post[SSH_CHANNEL_OPEN]    = &channel_post_open;
  channel_post[SSH_CHANNEL_DYNAMIC] = &channel_post_open; }
```

```c
void channel_prepare_select(fd_set **readsetp, fd_set **writesetp) {
    channel_handler(channel_pre, *readsetp, *writesetp);
}
```

```c
void channel_after_select(fd_set * readset, fd_set * writeset) {
    channel_handler(channel_post, readset, writeset);
}
```

```c
void channel_handler(chan_fn *ftab[], fd_set * readset, fd_set * writeset) {
    Channel *c;

    for(int i = 0; i < channels_alloc; i++) {
        c = channels[i];
        (*ftab[c->type])(c, readset, writeset);
    }
}
```

```
{ channel_pre[SSH_CHANNEL_OPEN]    = &channel_pre_open_13;
  channel_pre[SSH_CHANNEL_DYNAMIC] = &channel_pre_dynamic; }
```

```
void channel_prepare_select(fd_set **readsetp, fd_set **writesetp) {
    channel_handler(channel_pre, *readsetp, *writesetp);
}
```

```
void channel_after_select(fd_set * readset, fd_set * writeset) {
    channel_handler(channel_post, readset, writeset);
}
```

```
void channel_handler(chan_fn *ftab[], fd_set * readset, fd_set * writeset) {
    Channel *c;

    for(int i = 0; i < channels_alloc; i++) {
        c = channels[i];
        (*ftab[c->type])(c, readset, writeset);
    }
}
```

```
{ channel_post[SSH_CHANNEL_OPEN]    = &channel_post_open;
  channel_post[SSH_CHANNEL_DYNAMIC] = &channel_post_open; }
```

```
void channel_prepare_select(fd_set **readsetp, fd_set **writesetp) {
    channel_handler(channel_pre, *readsetp, *writesetp);
}
```

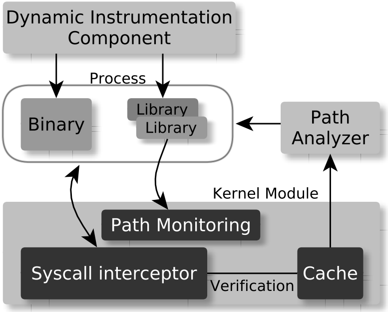
```
void channel_after_select(fd_set * readset, fd_set * writeset) {
    channel_handler(channel_post, readset, writeset);
}
```

```
void channel_handler(chan_fn *ftab[], fd_set * readset, fd_set * writeset) {
    Channel *c;

    for(int i = 0; i < channels_alloc; i++) {
        c = channels[i];
        (*ftab[c->type])(c, readset, writeset);
    }
}
```

Overview

- Kernel module verifies paths leading up to system calls
- Upon system call, check validity of edges in LBR
- JIT analyzer validates paths using interprocedural CFG



PathArmor

Challenges

- **Path monitoring:** continuous path tracking is expensive
  - Key obstacle in original CFI proposal by Abadi et al.
  - *PathArmor* uses LBR to efficiently track control transfers
- **Path verification:** cannot scale to validate every program state
  - Aggregate verification at security-sensitive system calls
  - (Persistently) cache results for future lookups
- **Path analysis:** static analysis of all paths leads to explosion
  - *PathArmor* uses on-demand JIT analysis on normalized CFG

Path Monitoring

Kernel module Branch Record core (Intel LBR API)

- Circular buffer which tracks last 16 (indirect) branches per process-thread
- Instrumentation uses `ioctl()` interface to safely toggle LBR tracking (avoid in-library LBR pollution)

LBR pollution

- Library calls may pollute LBR with library-internal edges
- Temporarily disabling LBR tracking prevents this

Path Verification

System call interceptor

- Alternative syscall handler validates paths to dangerous syscalls (policy driven) using JIT analyzer
- `mprotect`, `mmap`, `exec`, `sigaction`, `signal`, `raise`, `kill`
- Turing-completeness without syscalls does not allow system compromise
- Cache MD4 hash of valid paths (second-preimage resistance prevents path crafting attacks)

Path Analysis

JIT analyzer

- Lazily validate LBR paths in static interprocedural CFG
  - Modular indirect call resolution component
  - Collapse direct intraprocedural edges (prevent path explosion)
  - Policy-driven context sensitivity (default policy below)
- *Backward edge context sensitivity:* call/return matching
- *Forward edge context sensitivity:* code pointer tracking

Evaluation – Performance

Practical CFI: low overhead

Server	**Normalized Run Time** +*LInstr*	+*PathVer*
vsftpd	1.000	1.000
proftpd	1.000	1.000
pure-ftpd	1.053	1.074
lighttpd	1.236	1.275
nginx	1.178	1.174
openssh	1.003	1.020
exim	1.019	1.079
geomean	1.066	1.085

Practical CFI: low overhead

Server	**Normalized Run Time** +*LInstr*	+*PathVer*
vsftpd	1.000	1.000
proftpd	1.000	1.000
pure-ftpd	1.053	1.074
lighttpd	1.236	1.275
nginx	1.178	1.174
openssh	1.003	1.020
exim	1.019	1.079
geomean	1.066	1.085

Practical CFI: low overhead

Server	**Normalized Run Time** +*LInstr*	+*PathVer*
vsftpd	1.000	1.000
proftpd	1.000	1.000
pure-ftpd	1.053	1.074
lighttpd	1.236	1.275
nginx	1.178	[illegible]
openssh	1.003	[illegible]
exim	1.019	1.079
geomean	1.066	1.085

Many library calls

$1,209,081$

Practical CFI: low overhead

Server	**Normalized Run Time** +*LInstr*	
vsftpd	1.000	[illegible]
proftpd	1.000	[illegible]
pure-ftpd	1.053	1.074
lighttpd	1.236	[illegible]
nginx	1.178	[illegible]
openssh	1.003	[illegible]
exim	1.019	1.079
geomean	1.066	1.085

Not so many library calls

$35,883$

$171,440$

Many library calls

$1,209,081$

Evaluation – Performance

Practical CFI: low overhead

Server	**Normalized Run Time** +*LInstr*	+*PathVer*
vsftpd	1.000	1.000
proftpd	1.000	1.000
pure-ftpd	1.053	1.074
lighttpd		
nginx		
openssh		
exim		
geomean	1.066	1.085

Verification is fast

- Few lookups (~ 231)
- Cache hits ($\sim 90\%$)

Evaluation – Performance

Practical CFI: low overhead

Server	Normalized Run Time +*LInstr*	+*PathVer*
vsftpd	1.000	1.000
proftpd	1.000	1.000
pure-ftpd	1.053	1.074
lighttpd	1.236	1.275
nginx	1.178	1.174
openssh	1.003	1.020
exim	1.019	1.079
geomean	1.066	1.085

More benchmark details in the paper

SPEC CPU2006: $\sim 3\%$ overhead

Security

Server	coarse-grained		fine-grained		PathArmor	
	$\|G\|$	$[G_{\text{Len}}]$	$\|G\|$	$[G_{\text{Len}}]$	$\|G\|$	$[G_{\text{Len}}]$
vsftpd	543.26	3.5	3.17	8.0	1.27	13.1
proftpd	3249.55	2.2	19.96	4.0	6.11	7.5
pure-ftpd	403.57	2.2	5.37	4.5	1.94	5.1
lighttpd	561.00	2.0	2.77	4.8	1.00	5.5
nginx	1482.08	2.8	23.40	9.3	14.90	9.9
openssh	1725.20	2.1	16.02	3.9	4.37	7.2
exim	2588.53	2.2	25.10	4.4	11.05	11.1

Statistics captured at run-time

Evaluation

$|G|$ decreases

Less gadgets available

Sec

Server	coarse-grained $\|G\|$	coarse-grained $[G_{\text{Len}}]$	fine-grained $\|G\|$	fine-grained $[G_{\text{Len}}]$	PathArmor $\|G\|$	PathArmor $[G_{\text{Len}}]$
vsftpd	543.26	3.5	3.17	8.0	1.27	13.1
proftpd	3249.55	2.2	19.96	4.0	6.11	7.5
pure-ftpd	403.57	2.2	5.37	4.5	1.94	5.1
lighttpd	561.00	2.0	2.77	4.8	1.00	5.5
nginx	1482.08	2.8	23.40	9.3	14.90	9.9
openssh	1725.20	2.1	16.02	3.9	4.37	7.2
exim	2588.53	2.2	25.10	4.4	11.05	11.1

Statistics captured at run-time

Evaluation

$|G|$ decreases

Less gadgets available

Sec

Server	coarse-grained $\lvert G\rvert$	coarse-grained $[G_{\text{Len}}]$	fine-grained $\lvert G\rvert$	fine-grained $[G_{\text{Len}}]$	PathArmor $\lvert G\rvert$	PathArmor $[G_{\text{Len}}]$
vsftpd	543.26	3.5	3.17	8.0	1.27	13.1
proftpd	3249.55	2.2	19.96	4.0	6.11	7.5
pure-ftpd	403.57	2.2	5.37	4.5	1.94	5.1
lighttpd	561.00	2.0	2.77	4.8	1.00	5.5
nginx	1482.08	2.8	23.40	9.3	14.90	9.9
openssh	1725.20	2.1	16.02	3.9	4.37	7.2
exim	2588.53	2.2	25.10	4.4	11.05	11.1

Geometric means

-99.7% (coarse-grained) / -61.6% (fine-grained)

Evaluation

$[G_{\text{Len}}]$ increases

Leftover gadgets are longer, more complex

Security

Server	coarse-grained		fine-grained		PathArmor	
	$\|G\|$	$[G_{\text{Len}}]$	$\|G\|$	$[G_{\text{Len}}]$	$\|G\|$	$[G_{\text{Len}}]$
vsftpd	543.26	3.5	3.17	8.0	1.27	13.1
proftpd	3249.55	2.2	19.96	4.0	6.11	7.5
pure-ftpd	403.57	2.2	5.37	4.5	1.94	5.1
lighttpd	561.00	2.0	2.77	4.8	1.00	5.5
nginx	1482.08	2.8	23.40	9.3	14.90	9.9
openssh	1725.20	2.1	16.02	3.9	4.37	7.2
exim	2588.53	2.2	25.10	4.4	11.05	11.1

Statistics captured at run-time

Evaluation

$[G_{\text{Len}}]$ increases

Leftover gadgets are longer, more complex

Security

Server	coarse-grained		fine-grained		PathArmor	
	$\|G\|$	$[G_{\text{Len}}]$	$\|G\|$	$[G_{\text{Len}}]$	$\|G\|$	$[G_{\text{Len}}]$
vsftpd	543.26	3.5	3.17	8.0	1.27	13.1
proftpd	3249.55	2.2	19.96	4.0	6.11	7.5
pure-ftpd	403.57	2.2	5.37	4.5	1.94	5.1
lighttpd	561.00	2.0	2.77	4.8	1.00	5.5
nginx	1482.08	2.8	23.40	9.3	14.90	9.9
openssh	1725.20	2.1	16.02	3.9	4.37	7.2
exim	2588.53	2.2	25.10	4.4	11.05	11.1

Geometric means

$+245\%$ (coarse-grained) / $+53\%$ (fine-grained)

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10   void vuln() {
0x20     strcpy(buf, in);
0x30     return;
0x40   }
0x50
0x60   main() {
0x70     foo();
0x80     vuln();
0x90     bar();
0xa0   }
```

LBR during **benign** execution

Source	Destination
???	&main

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10  void vuln() {
0x20    strcpy(buf, in);
0x30    return;
0x40  }
0x50
0x60  main() {
0x70    foo();
0x80    vuln();
0x90    bar();
0xa0  }
```

LBR during **benign** execution

Source	Destination
???	&main
0x70	&foo

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10  void vuln() {
0x20    strcpy(buf, in);
0x30    return;
0x40  }
0x50
0x60  main() {
0x70    foo();
0x80    vuln();
0x90    bar();
0xa0  }
```

LBR during **benign** execution

Source	Destination
???	&main
0x70	&foo
foo.ret	0x80

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10  void vuln() {
0x20    strcpy(buf, in);
0x30    return;
0x40  }
0x50
0x60  main() {
0x70    foo();
0x80    vuln();
0x90    bar();
0xa0  }
```

LBR during **benign** execution

Source	Destination
???	&main
0x70	&foo
foo.ret	0x80
0x80	0x10

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10  void vuln() {
0x20    strcpy(buf, in);
0x30    return;
0x40  }
0x50
0x60  main() {
0x70    foo();
0x80    vuln();
0x90    bar();
0xa0  }
```

LBR during **benign** execution

Source	Destination
???	&main
0x70	&foo
foo.ret	0x80
0x80	0x10
0x30	0x90

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10  void vuln() {
0x20    strcpy(buf, in);
0x30    return;
0x40  }
0x50
0x60  main() {
0x70    foo();
0x80    vuln();
0x90    bar();
0xa0  }
```

LBR during **benign** execution

Source	Destination
???	&main
0x70	&foo
foo.ret	0x80
0x80	0x10
0x30	0x90
0x90	&bar

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10  void vuln() {
0x20    strcpy(buf, in);
0x30    return;
0x40  }
0x50
0x60  main() {
0x70    foo();
0x80    vuln();
0x90    bar();
0xa0  }
```

LBR during **benign** execution

Source	Destination
???	&main
0x70	&foo
foo.ret	0x80
0x80	0x10
0x30	0x90
0x90	&bar

Valid

According to the CFG

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10  void vuln() {
0x20    strcpy(buf, in);
0x30    return;
0x40  }
0x50
0x60  main() {
0x70    foo();
0x80    vuln();
0x90    bar();
0xa0  }
```

LBR when **exploiting** `vuln()`

Source	Destination
???	&main

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10  void vuln() {
0x20    strcpy(buf, in);
0x30    return;
0x40  }
0x50
0x60  main() {
0x70    foo();
0x80    vuln();
0x90    bar();
0xa0  }
```

LBR when **exploiting** `vuln()`

Source	Destination
???	&main
0x70	&foo

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10   void vuln() {
0x20     strcpy(buf, in);
0x30     return;
0x40   }
0x50
0x60   main() {
0x70     foo();
0x80     vuln();
0x90     bar();
0xa0   }
```

LBR when **exploiting** `vuln()`

Source	Destination
`???`	`&main`
`0x70`	`&foo`
`foo.ret`	`0x80`

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10   void vuln() {
0x20     strcpy(buf, in);
0x30     return;
0x40   }
0x50
0x60   main() {
0x70     foo();
0x80     vuln();
0x90     bar();
0xa0   }
```

LBR when **exploiting** `vuln()`

Source	Destination
???	&main
0x70	&foo
foo.ret	0x80
0x80	0x10

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10  void vuln() {
0x20    strcpy(buf, in);
0x30    return;
0x40  }
0x50
0x60  main() {
0x70    foo();
0x80    vuln();
0x90    bar();
0xa0  }
```

LBR when **exploiting** `vuln()`

Source	Destination
???	&main
0x70	&foo
foo.ret	0x80
0x80	0x10
0x30	ioctl()

Attacks

Instrumentation Tampering

- Use our `ioctl` LBR disabling code as gadget
- Endpoint verification will detect control-flow diversion

```
0x10  void vuln() {
0x20    strcpy(buf, in);
0x30    return;
0x40  }
0x50
0x60  main() {
0x70    foo();
0x80    vuln();
0x90    bar();
0xa0  }
```

LBR when **exploiting** `vuln()`

Source	Destination
???	&main
0x70	&foo
foo.ret	0x80
0x80	0x10
0x30	ioctl()

Invalid

Edge not in CFG

Attacks

History Flushing Attacks (attacks against kBouncer, ROPecker)

- 16 *NOP-like gadgets* to flush the ROP chain
- *Termination gadget* to restore arguments

Attacks

History Flushing Attacks (atta

- 16 *NOP-like gadgets* to flu
- *Termination gadget* to restore arguments

Entire *path* must be valid

Hard to maintain register state

History Flushing Attacks (atta

- 16 *NOP-like gadgets* to flus
- *Termination gadget* to restore arguments

Entire *path* must be valid

Hard to maintain register state

Abide to the CFG

Restore right before exec

History Flushing Attacks (atta

- 16 *NOP-like gadgets* to flu
- *Termination gadget* to restore arguments

Entire *path* must be valid

Hard to maintain register state

Abide to the CFG

Restore right before exec

Award!

History Flush PathArmor and get a prize!

- Exploit *existing* C code
- Send us the PoC


HENDRIK-JAN

Conclusion (1/2)

Conclusion

Practical Context-Sensitive CFI

- Context-sensitive CFI *can* be implemented efficiently
- Low overhead by leveraging hardware features
- Improved security (fine-grained, context-insensitive CFI)
- *Enabling framework*

Conclusion (2/2)

No Vaporware!

- *PathArmor* released open-source!
  `https://github.com/dennisaa/patharmor`

Conclusion (2/2)

No Vaporware!

- *PathArmor* released open-source!
  `https://github.com/dennisaa/patharmor`

BAndroid (shameless advertisement)

- Google killed 2FA but does not care
- `http://www.few.vu.nl/~vvdveen/bandroid.html`