

Paper-Reading Group

Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems

Yuanzhong Xu, Weidong Cui, and Marcus Peinado
Microsoft Research

Motivation

- Legacy OS is responsible for isolation and confidentiality
- Commercially relevant OS are huge code bases
 - often contain bugs
 - if compromised allow access to everything
- Shielding systems try to solve this
 - Hypervisor-based (Overshadow, InkTag)
 - Hardware-based (Haven / VC3 on SGX)
 - Applications depend on insulation of applications from attacks

Motivation



- However, they largely ignore side-channels
 - which are a major problem in the “untrusted OS” scenario

Contributions

- A new class of side-channels: controlled-channel attacks
- A no-noise side-channel for shielding systems
- An efficient implementation on Haven and InkTag
- Applicable to wide range of applications

Attack Model

- Premise: Memory management by untrusted OS
- OS can map / revoke page access
- Legacy applications not specially hardened against side-channels
- Does **not** apply to systems like Flickr (only static resources)

Design

```
char* WelcomeMessage( GENDER s ) {  
    char *mesg;  
  
    // GENDER is an enum of MALE and FEMALE  
    if ( s == MALE ) {  
        mesg = WelcomeMessageForMale();  
    } else { // FEMALE  
        mesg = WelcomeMessageForFemale();  
    }  
    return mesg;  
}
```

Fig. 1: Example function with input-dependent control transfer.

```
void CountLogin( GENDER s ) {  
    if ( s == MALE ) {  
        gMaleCount ++;  
    } else {  
        gFemaleCount ++;  
    }  
}
```

Fig. 2: Example function with input-dependent data access.

Page-Fault sequences

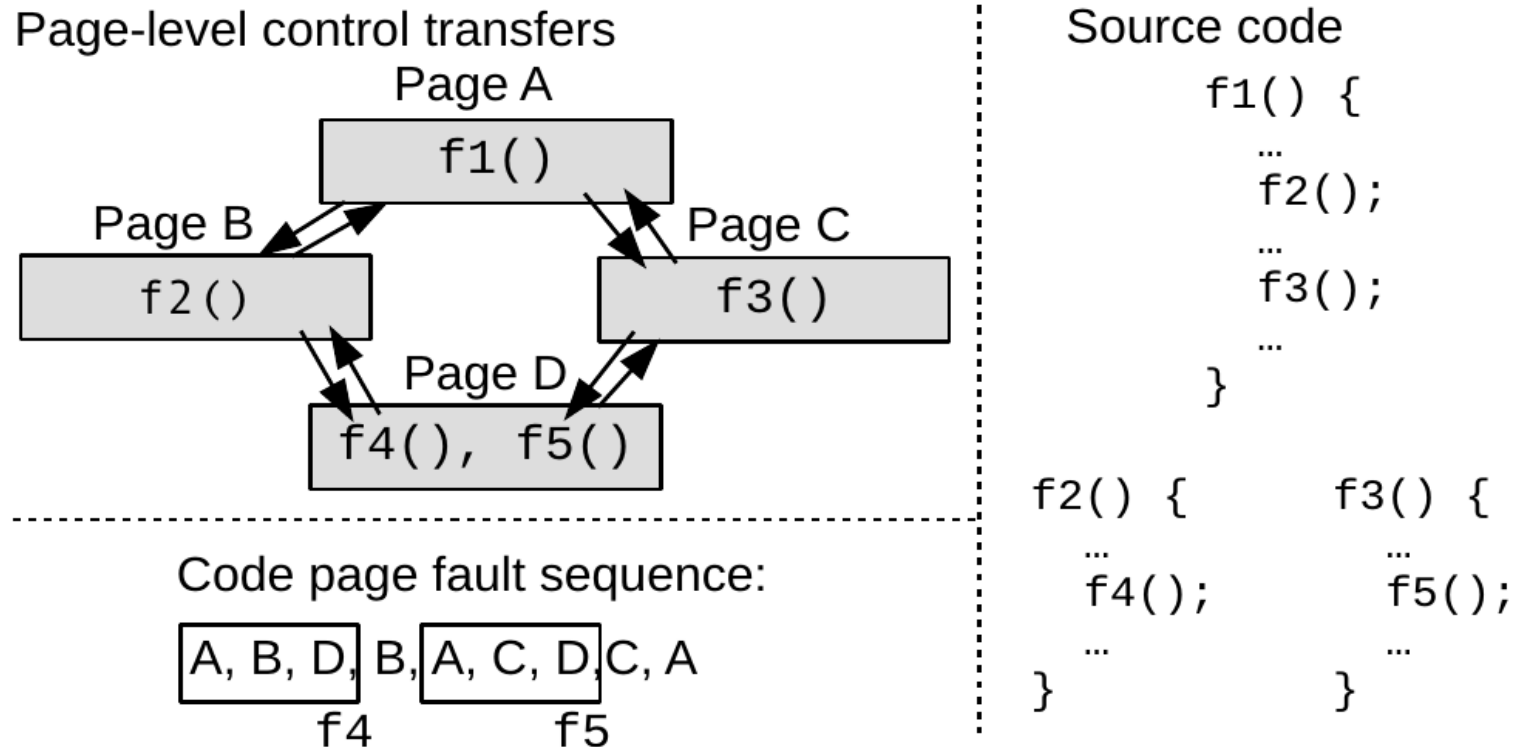


Fig. 3: The attacker can only observe page-level control transfers. However, functions sharing the same page can often be distinguished by different page-fault sequences.

Challenges

- When to revoke page access?
- Multiple page-accesses by single instructions!

Attacks - Freetype

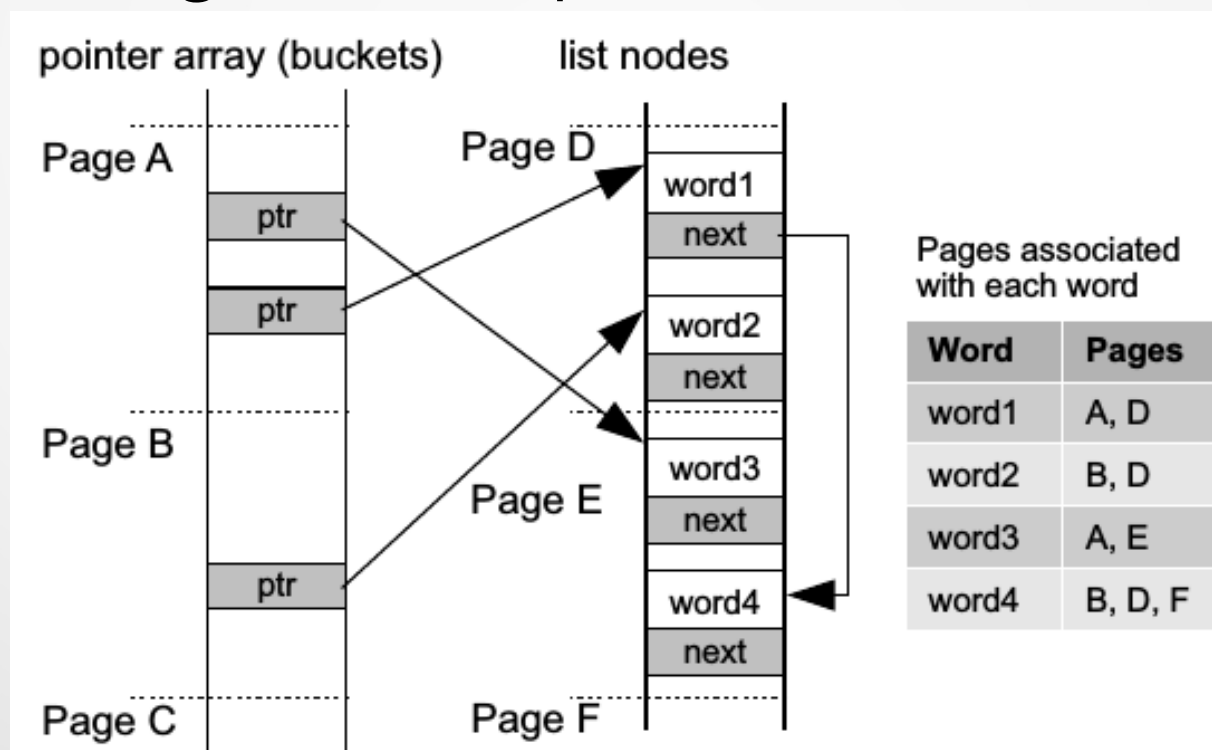
- Each glyph has a unique page-fault sequence
- Intercept TT_Load_Glyph and track from there
- 100% recovery

	whole file		5 KB	
	Haven	InkTag	Haven	InkTag
baseline time (s)	5.16	8.72	0.18	0.21
attack time (s)	19.3	280.21	0.52	6.62
overhead	3.74x	32.1x	2.89x	31.7x
pf count (million)	28.90	52.97	0.69	1.27
time per pf (ns)	489.5	5125.2	491.0	5054.7
post-proc. time (s)	< 100	< 100	< 10	< 10

Fig. 12: Performance of the FreeType attack.

Attacks – Hunspell

- Loads dictionary in alphabetic order
- Builds hash-map on top
- Track `HashMgr::lookup`



Attacks – Hunspell (2)

group size	Haven		InkTag	
	words	%	words	%
1	46864	75.16	48864	78.37
2	9964	15.98	9372	15.03
3	3546	5.69	2880	4.62
4	1100	1.76	852	1.37
5	485	0.78	275	0.44
6	222	0.36	60	0.10
7	49	0.08	14	0.02
8	48	0.08	16	0.03
9	45	0.07	0	0.00
10	30	0.05	20	0.03

accuracy of recovery		Haven		InkTag	
		words	%	words	%
recovered original word	no ambiguity	25320	63.75	27179	68.43
	rec. 2-group	6042	15.21	5751	14.48
	rec. 3-group	1985	5.00	2554	6.43
	rec. ≥ 4 -group	2869	7.22	890	2.24
recovered without affix	no ambiguity	1974	4.97	2291	5.77
	rec. 2-group	602	1.52	460	1.16
	rec. 3-group	213	0.54	145	0.37
	rec. ≥ 4 -group	291	0.73	186	0.47
not recovered		423	1.06	263	0.66

	Haven		InkTag	
	words	%	words	%
recovered exactly	35273	88.81	35760	90.03
recovered without affix	2880	7.25	2896	7.29
not recovered or incorrectly resolved ambiguity	1566	3.94	1063	2.68

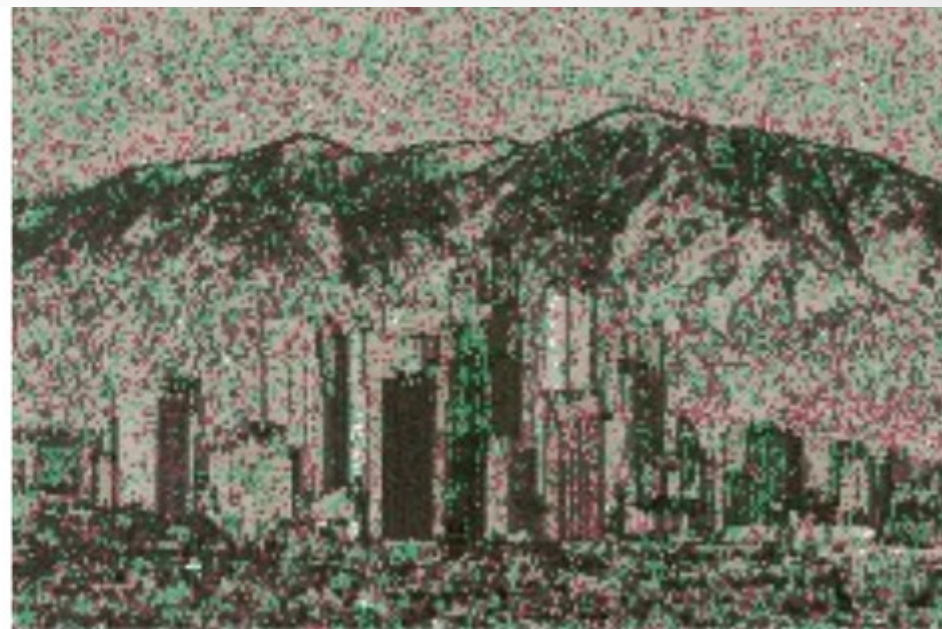
Attacks – Hunspell (3)

	whole file		last chapter
	Haven	InkTag	InkTag
baseline time (s)	0.12	0.12	0.051
attack time (s)	2.94	11.95	0.089
overhead	25.2x	99.6x	1.75x
pf count (million)	5.84	2.41	0.0085
time per pf (ns)	484.2	4955.6	4517.2
post-proc. time (s)	< 20	< 10	< 5

Fig. 13: Performance of the Hunspell attack.

Attacks - libjpeg

- libjpeg decodes 8x8 blocks (IDCT)
- Simplified code-path for “simple”/plain lines of a block
- Can be identified by the page-fault sequence
- Track `jpeg_idct_islow`



Attacks – libjpeg (2)



	562 KB image		36 KB image	
	Haven	InkTag	Haven	InkTag
baseline time (s)	0.08	0.12	0.04	0.014
attack time (s)	16.77	42.59	0.50	2.84
overhead	209.6x	354.9x	12.5x	202.8x
pf count (million)	35.8	8.97	0.95	0.56
time per pf (ns)	482.7	4735.5	466.0	5035.2
post-proc. time (s)	< 5	< 5	< 5	< 5


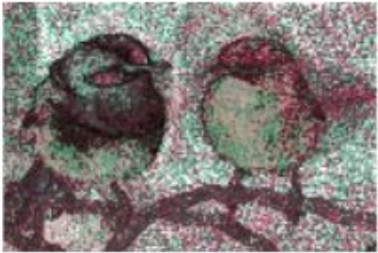

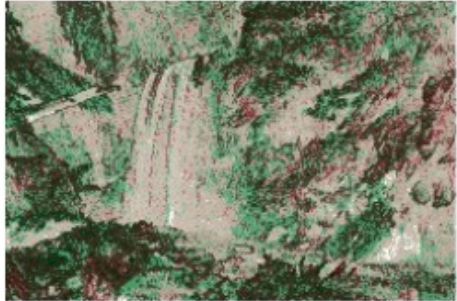







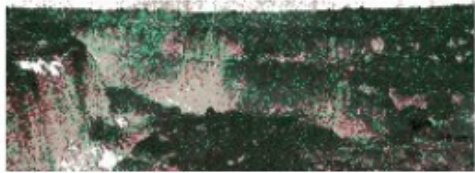



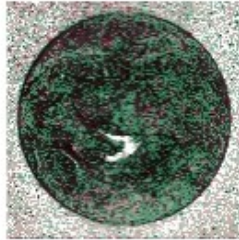
Countermeasures

- Standard cache side-channel mitigations apply
 - rewrite application to avoid secret dependent code-flow or data accesses
 - But harder than for small, simple crypto keys!
 - prohibit paging or self-paging
 - detect attack (slowdown, pf-count)


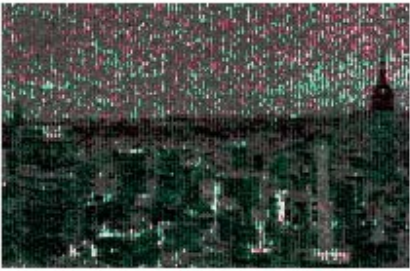






DISCUSS!

- How could you defend against these attacks?
- Are shielding-systems doomed to fail?

Nice pictures

Original	Recovered	Original	Recovered
			
			
			
			

Nice pictures

Original	Recovered
	
	
	
	

Original	Recovered
