Aalto University

UNIVERSITY OF WATERLOO

Hardware-assisted Run-time Protection

N. Asokan[†‡]

https://asokan.org/asokan/

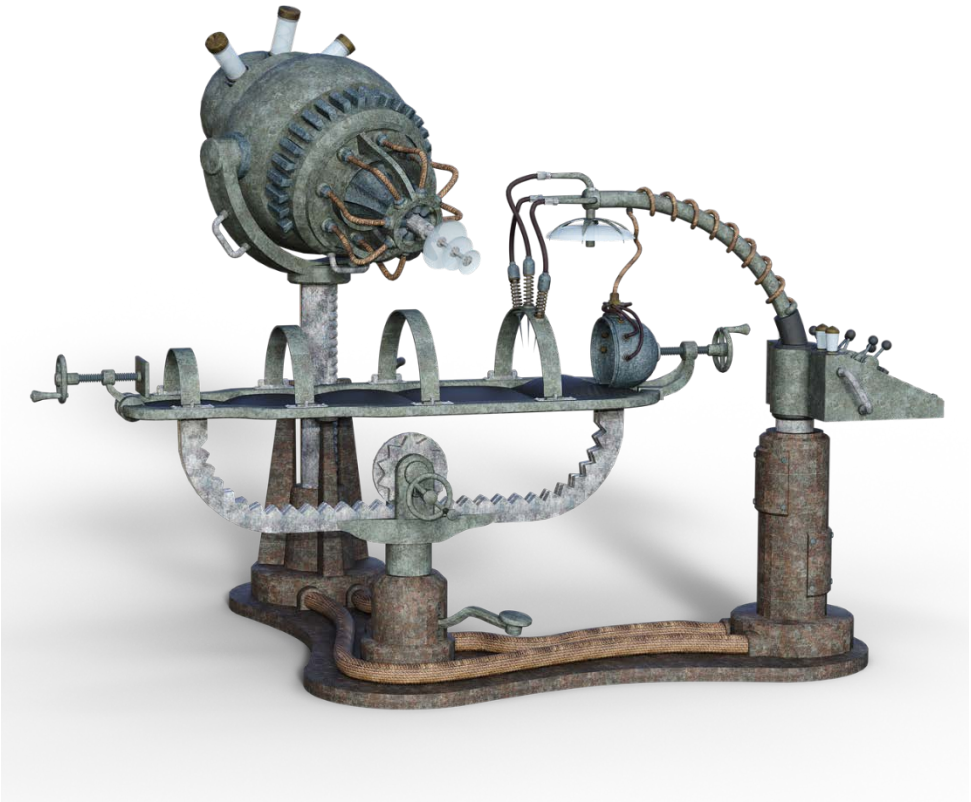
@nasokan

Acknowledgements: Thomas Nyman[‡], *Hans Liljestrand*[†], *Lachlan J Gunn*[‡], *Jan-Erik Ekberg*[‡, §]

[†)] *University of Waterloo,* [‡)] *Aalto University,* [§)] *Huawei Technologies*

You will be learning

- **Part 1: Memory-related run-time attacks**
  - Common attack techniques against C/C++
- **Part 2: Hardware-assisted defenses**
  - Emerging mechanisms in CotS processors
- **Part 3: Theory of run-time attacks**
  - What are “weird machines”?



Example: Buffer overflows caused by missing bounds checks

Software Developer

```c
int main(int argc, char *argv[])
{
    puts("So... The End...");
    doit(argv[1]);
    puts("or... maybe not?");

    return 0;
}
```

```c
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```

Source code

Compiler & Linker

Executable File

```
08048464 <main>:
 8048464:       55                      push   %ebp
 8048465:       89 e5                   mov    %esp,%ebp
 8048467:       68 20 85 04 08          push   $0x8048520
 804846c:       e8 9f fe ff ff          call   8048310 <puts@plt>
 8048471:       83 c4 04                add    $0x4,%esp
 8048474:       8b 45 0c                mov    0xc(%ebp),%eax
 8048477:       83 c0 04                add    $0x4,%eax
 804847a:       8b 00                   mov    (%eax),%eax
 804847c:       50                      push   %eax
 804847d:
 8048482:
 8048485:
 804848a:
 804848f:
 8048492:
 8048497:
 8048498:
 8048499:
 804849b:
 804849d:
 804849f:
```

```
0804843b <doit>:
 804843b:       55                      push   %ebp
 804843c:       89 e5                   mov    %esp,%ebp
 804843e:       83 ec 0c                sub    $0xc,%esp
 8048441:       8d 45 f4                lea    -0xc(%ebp),%eax
 8048444:       89 45 fc                mov    %eax,-0x4(%ebp)
 8048447:       ff 75 08                pushl  0x8(%ebp)
 804844a:       8d 45 f4                lea    -0xc(%ebp),%eax
 804844d:       50                      push   %eax
 804844e:       e8 ad fe ff ff          call   8048300 <strcpy@plt>
 8048453:       83 c4 08                add    $0x8,%esp
 8048456:       ff 75 fc                pushl  -0x4(%ebp)
 8048459:       e8 b2 fe ff ff          call   8048310 <puts@plt>
 804845e:       83 c4 04                add    $0x4,%esp
 8048461:       90                      nop
 8048462:       c9                      leave
 8048463:       c3                      ret
```

missing bounds-checks in call to strcpy!

Run-time behaviour

User

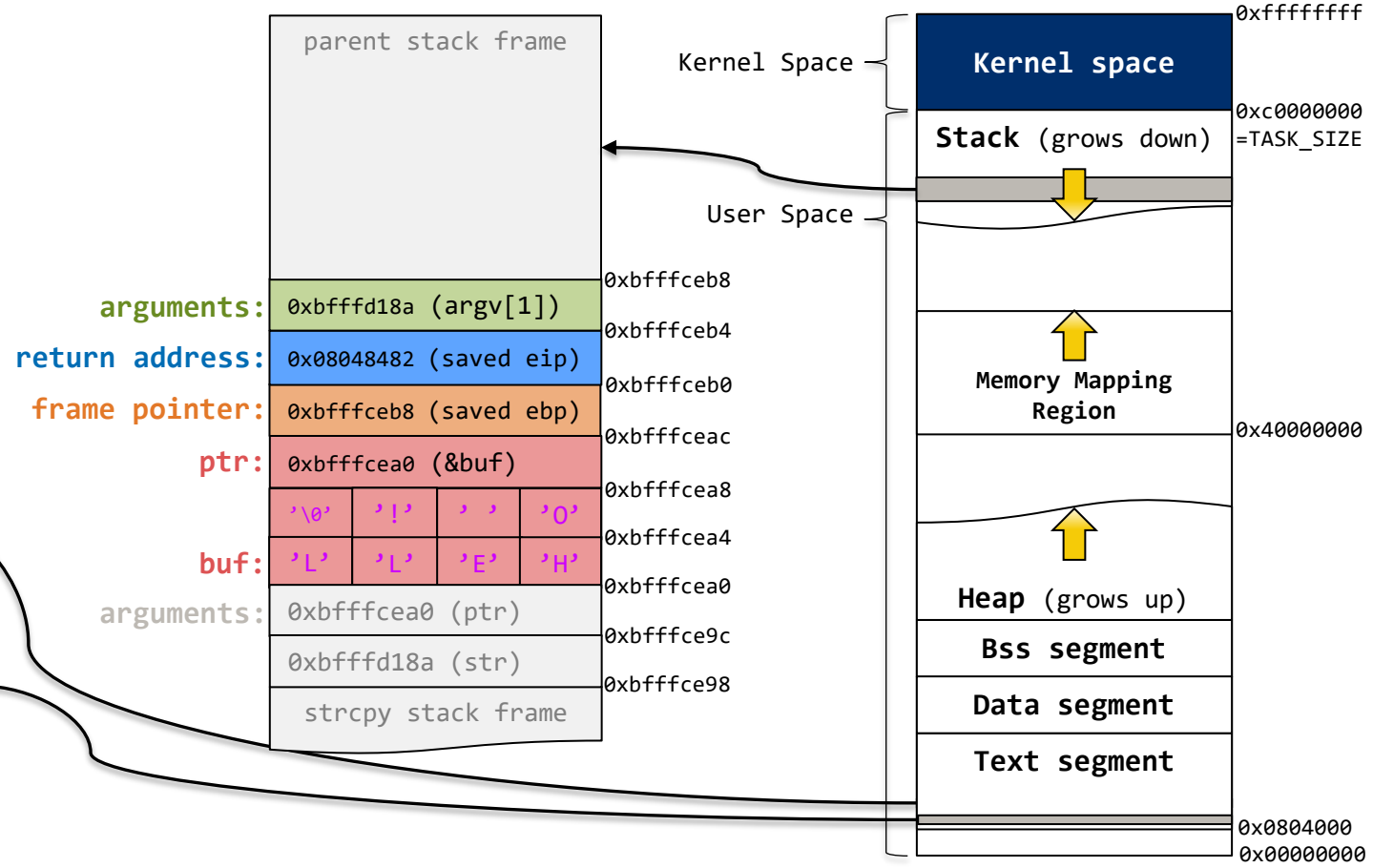
```
$ ./a.out "Hello !"
```

```
08048464 <main>:
                    ...
 8048474:    8b 45 0c          mov    0xc(%ebp),%eax
 8048477:    83 c0 04          add    $0x4,%eax
 804847a:    8b 00             mov    (%eax),%eax
 804847c:    50                push   %eax
 804847d:    e8 b9 ff ff ff    call   804843b <doit>
```

```
0804843b <doit>:
 804843b:    55                push   %ebp
 804843c:    89 e5             mov    %esp,%ebp
 804843e:    83 ec 0c          sub    $0xc,%esp
 8048441:    8d 45 f4          lea    -0xc(%ebp),%eax
 8048444:    89 45 fc          mov    %eax,-0x4(%ebp)
 8048447:    ff 75 08          pushl  0x8(%ebp)
 804844a:    8d 45 f4          lea    -0xc(%ebp),%eax
 804844d:    50                push   %eax
 804844e:    e8 ad fe ff ff    call   8048300 <strcpy@plt>
 8048453:    83 c4 08          add    $0x8,%esp
 8048456:    ff 75 fc          pushl  -0x4(%ebp)
 8048459:    e8 b2 fe ff ff    call   8048310 <puts@plt>
 804845e:    83 c4 04          add    $0x4,%esp
 8048461:    90                nop
 8048462:    c9                leave
 8048463:    c3                ret
```

```
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```

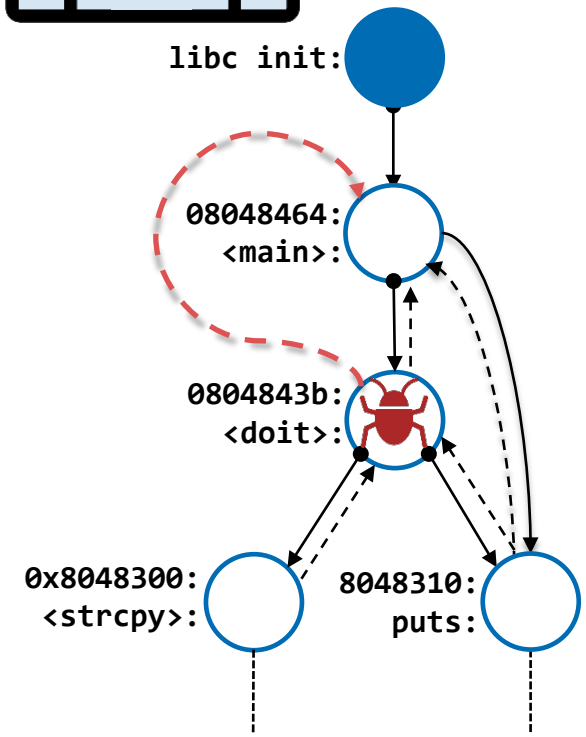


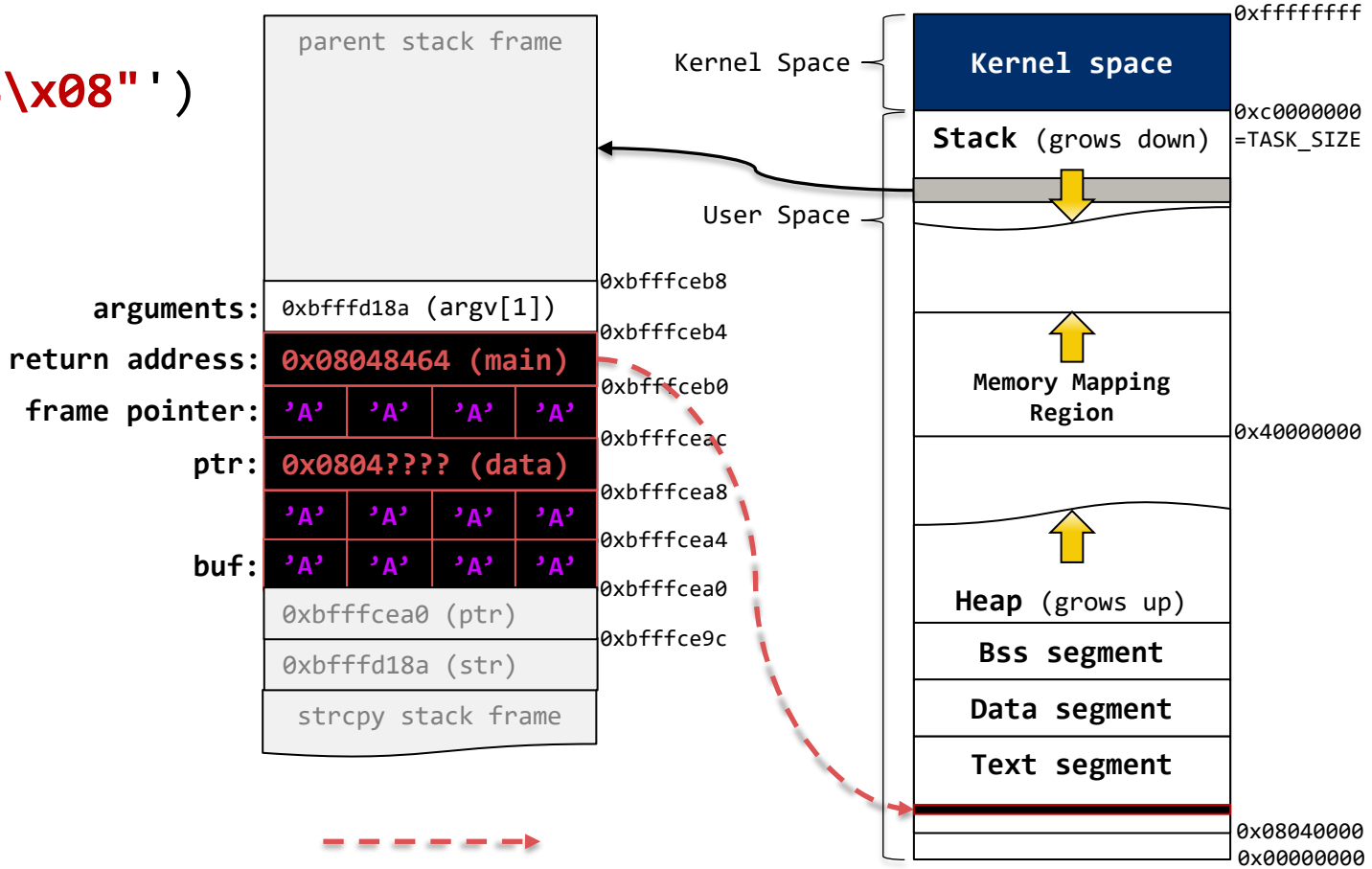
Control-flow hijacking

```
$ ./a.out $(perl -e 'print "A"x8 \
                 ."\x??\x??\x04\x08" \
                 ."A"x4 \
                 ."\x64\x84\x04\x08"')
```

```
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```





corrupt code pointer / control flow

Memory-related run-time attacks

Memory-related run-time attacks

Software written in memory unsafe languages such as C/C++

- Suffer from various memory-related errors

Memory errors may allow run-time attacks to compromise program behaviour

- *Control-flow hijacking / code injection*
- *Return-Oriented Programming* (ROP)
- *Non-control-data attacks*
- *Data-Oriented Programming* (DOP)

Run-time attacks compromise program behaviour

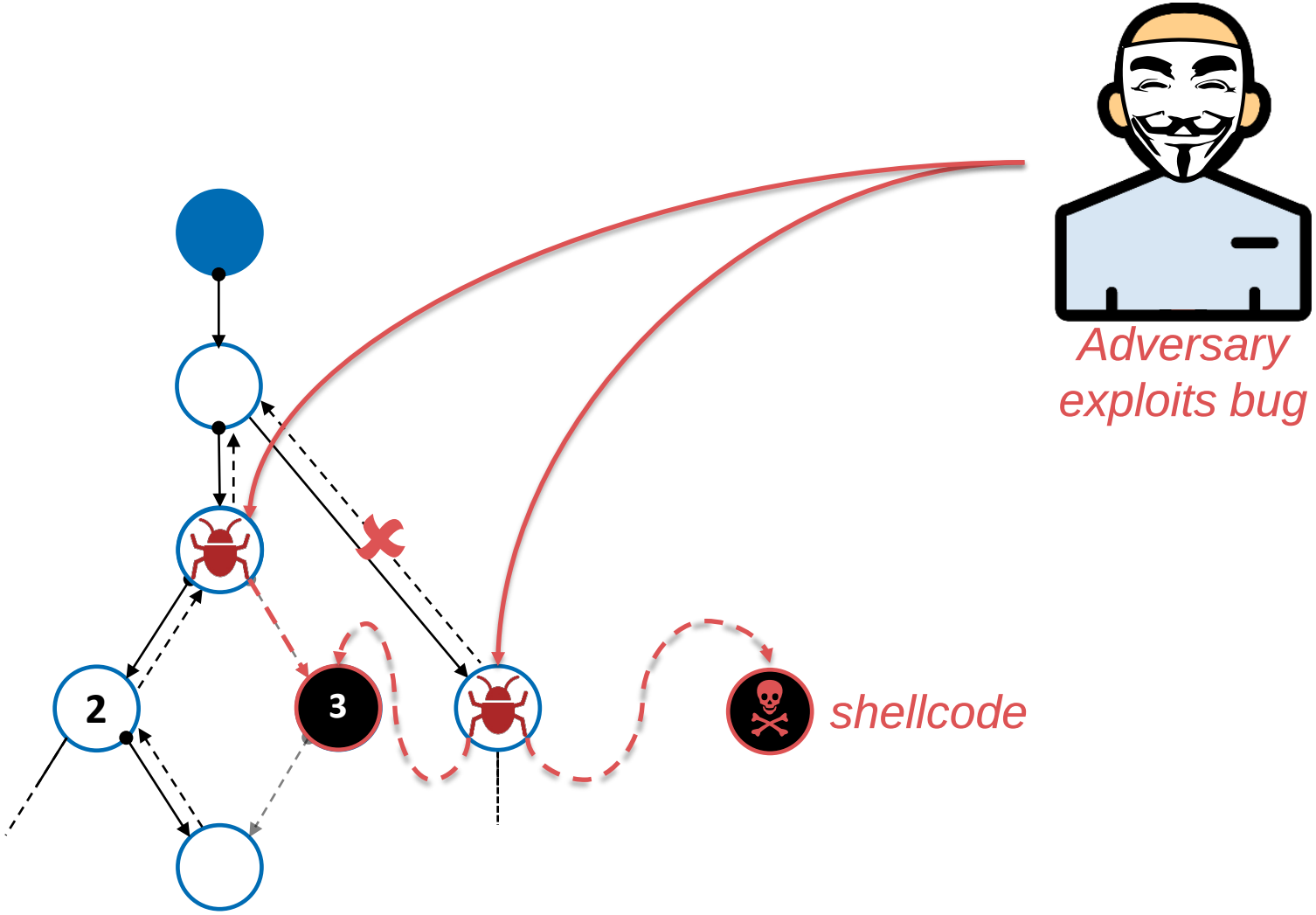
(i) Code-injection attack

(ii) Code-reuse attack

(iii) Non-control-data attack

```
① if (authenticated != true)
     then: call unprivileged()
     else: call   privileged()
…

② unprivileged() { … }
③   privileged() { … }
…
```



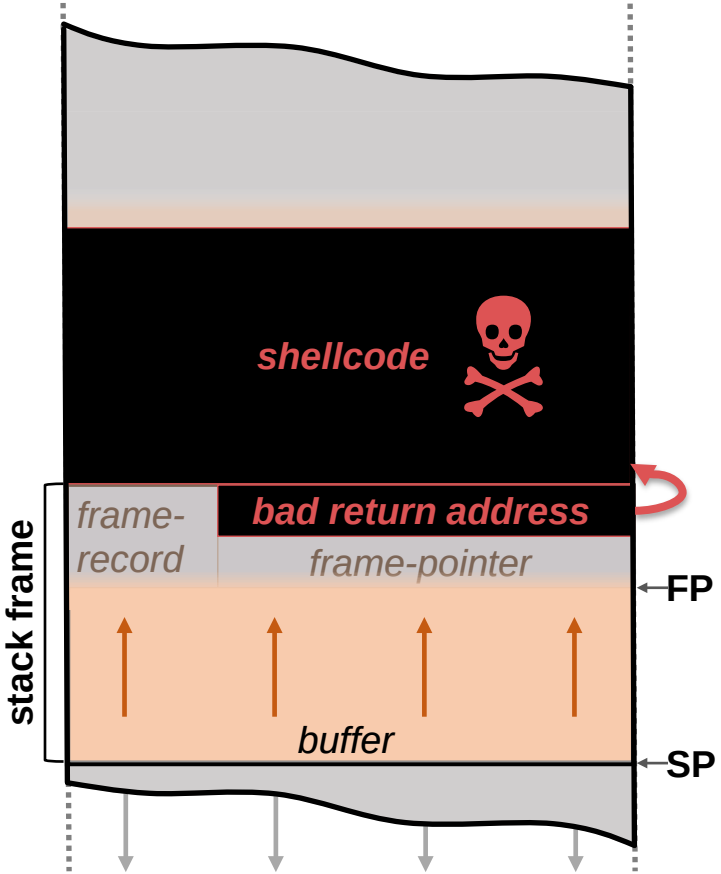
(i) Code-injection attacks

Exploit memory error (e.g. buffer overflow) to:

- Inject shellcode into writable memory (usually stack)
- Corrupt code pointer (usually return address) to redirect execution flow to shellcode

Countermeasures:

- **Stack canaries (1990)**
  Detect sequential overwrites that corrupt ret. addr.
- **W⊕X memory access control policy (2003)**
  Prevent execution of shellcode by ensuring that memory pages are either writable or executable



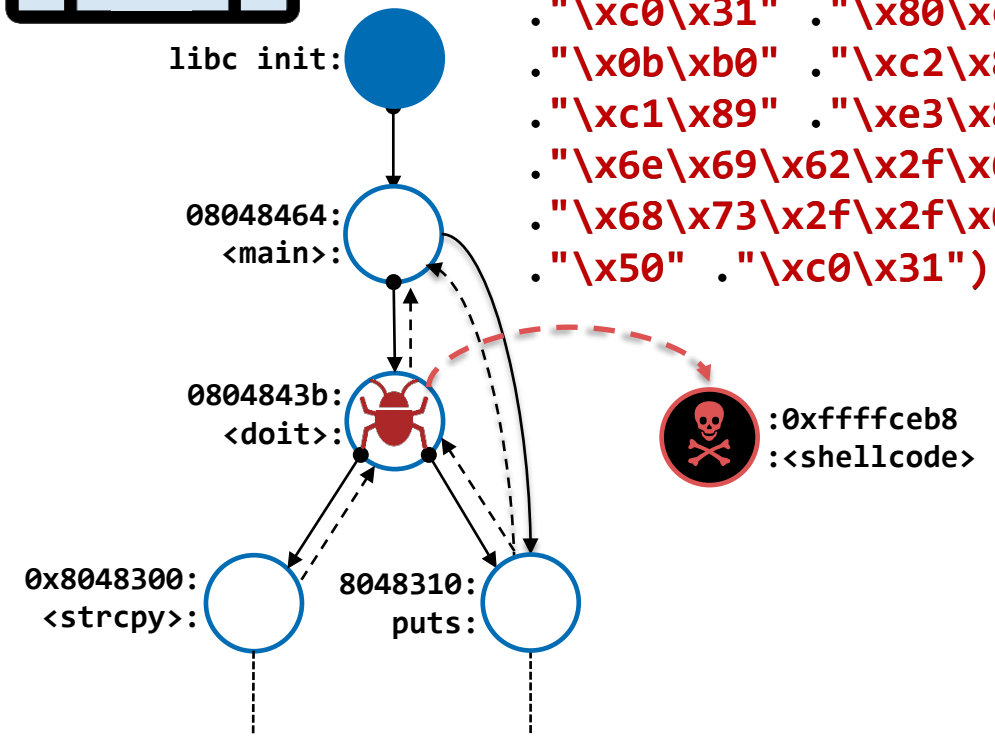
Elias Levy (as *Aleph One*), Smashing the stack for fun and profit, Phrack 7 (1996)

Cowan et al., StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks, USENIX Security (1998)

Szekeres et al., SoK: Eternal War in Memory, IEEE SP (2013)

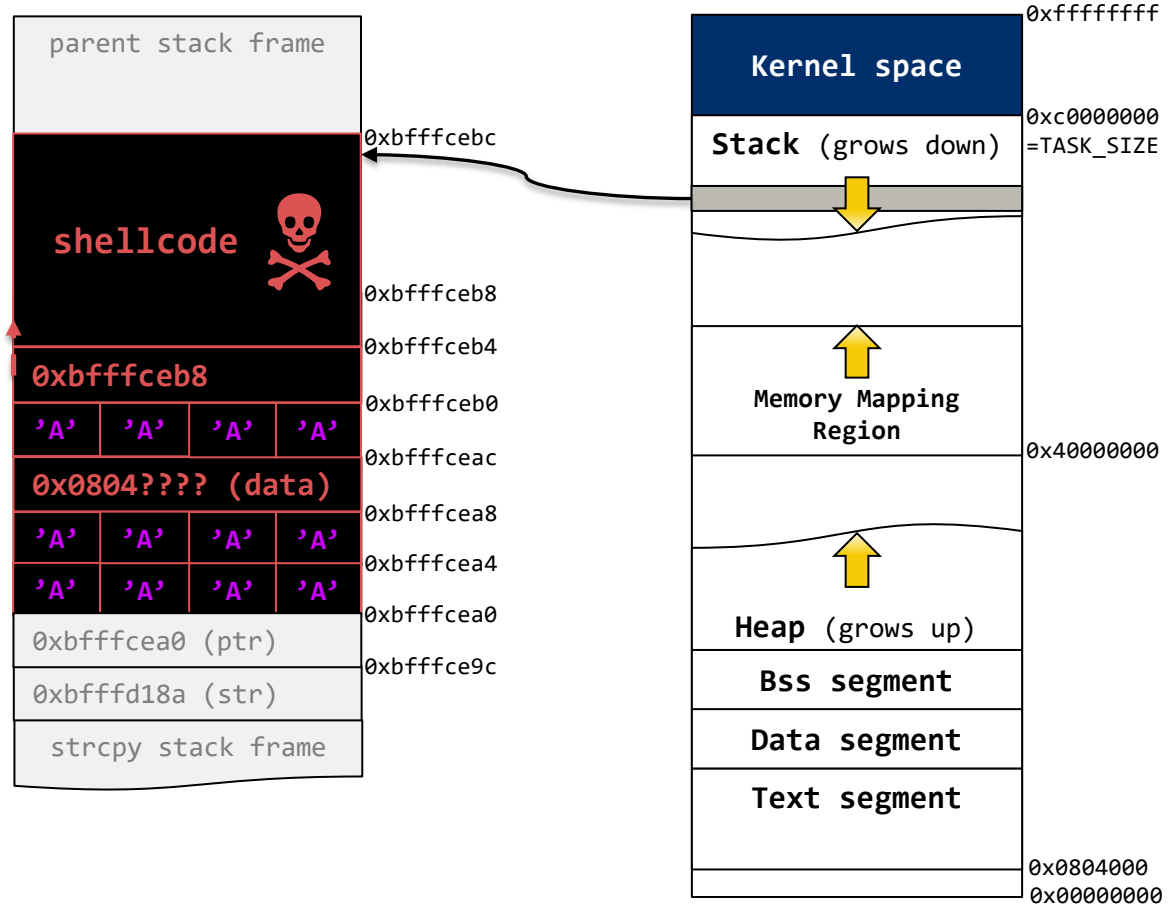
Classic code-injection

```
$ ./a.out $(perl -e 'print "A"x8 \
          ."\x??\x??\x04\x08" \
          ."A"x4 \
          ."\xb8\xce\xff\xfb" \
          ."\x80\xcd" ."\x40" \
          ."\xc0\x31" ."\x80\xcd" \
          ."\x0b\xb0" ."\xc2\x89" \
          ."\xc1\x89" ."\xe3\x89" \
          ."\x6e\x69\x62\x2f\x68" \
          ."\x68\x73\x2f\x2f\x68" \
          ."\x50" ."\xc0\x31")
```

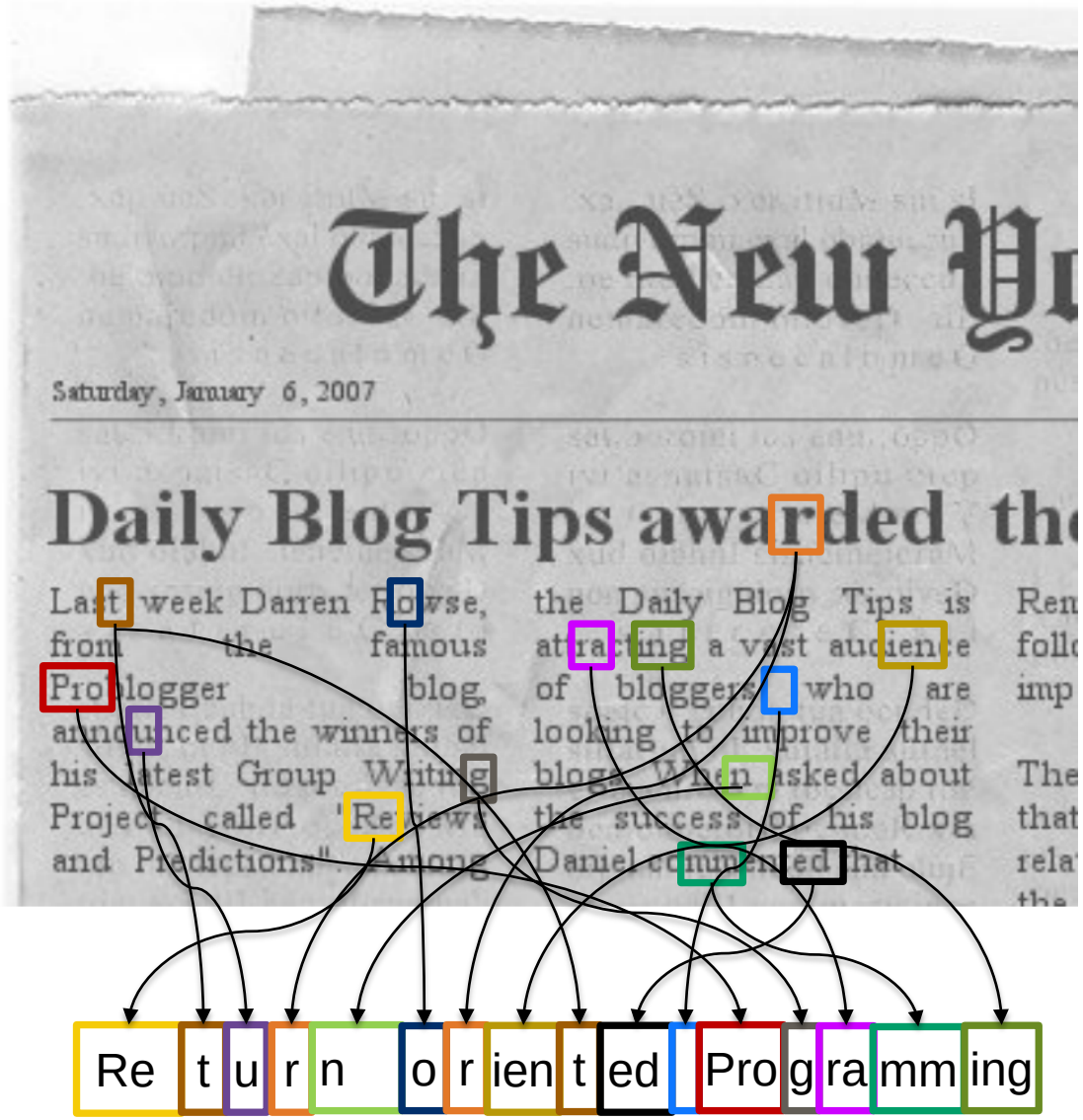



```
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```



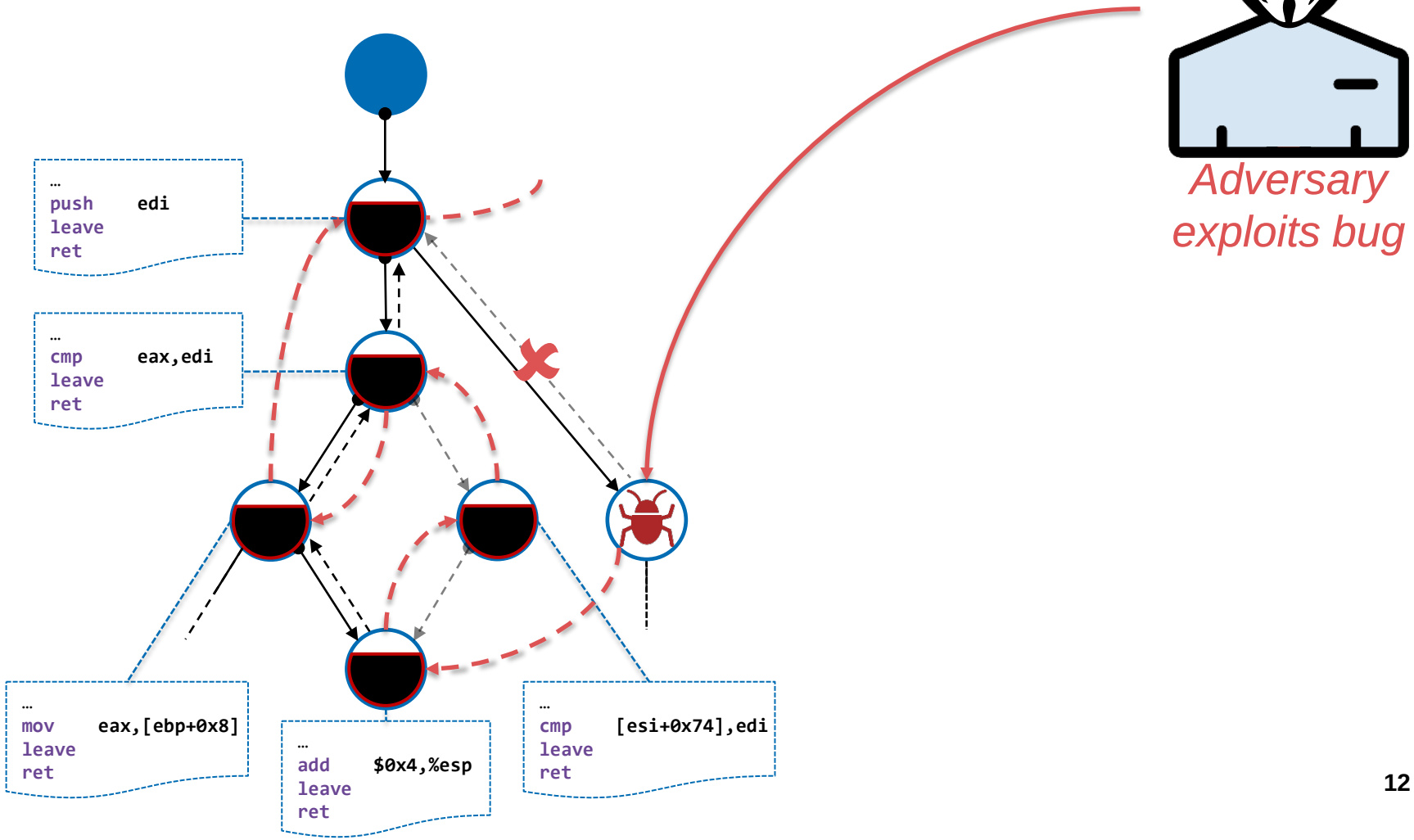
Return-oriented programming (high-level idea)



Return-oriented programming

Attacker arranges call stack with code pointers to existing code sequences ("*gadgets*")

- **Given a suitable gadget set, arbitrary *return-oriented programs* can be constructed**



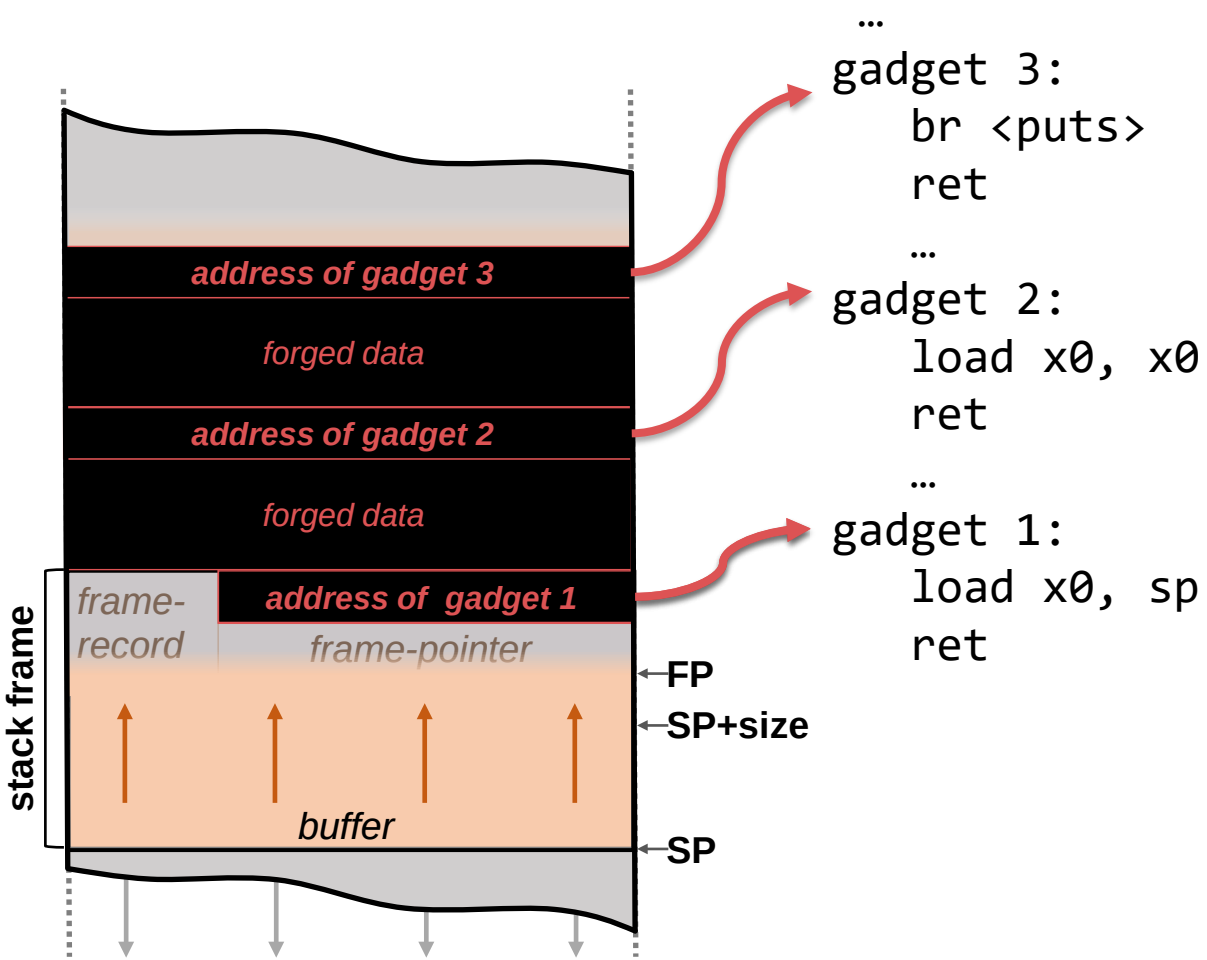
(ii) Code-reuse attacks

Exploit memory error without injecting code:

- Corrupt code pointer (usually return address) to redirect execution flow to **existing code:**
  - Library functions (return-into-libc)
  - Pre-existing instruction sequences (gadgets)

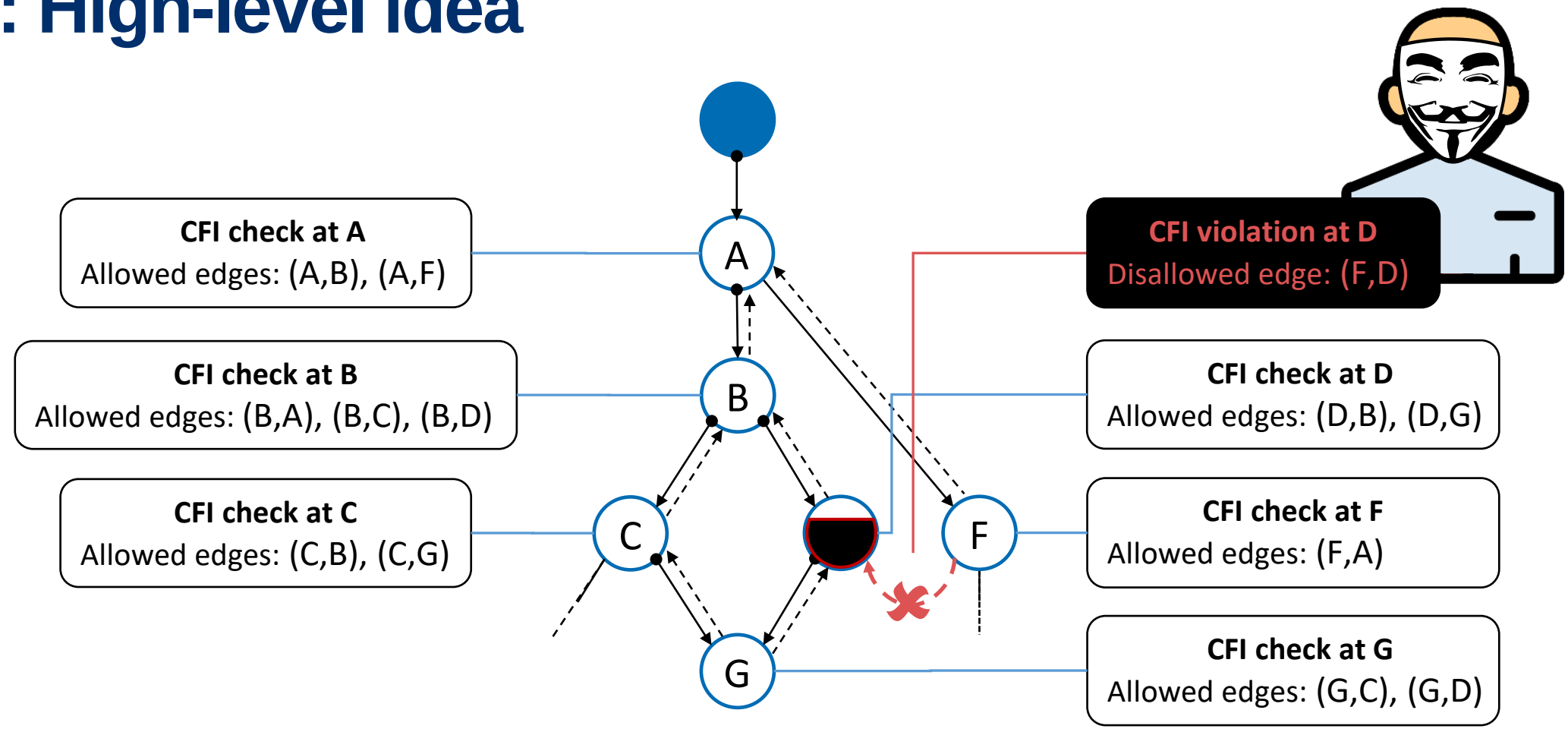
Countermeasures:

- **Control-flow Integrity (2005)**
  Detect control-flow transfers outside static control-flow graph or mismatched returns (shadow stack)
- **Address space randomization (2001)**
  Hide locations of useful gadgets in memory



A. Peslyak (as *Solar Designer*), Getting around non-executable stack (and fix), Bugtraq (1997)
H. Shacham, The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86), ACM CCS (2007)
T. Kornau, Return Oriented Programming for the ARM Architecture, MSc Thesis, RUB (2009)
M. Abadi, Control-flow integrity, ACM CCS (2005)

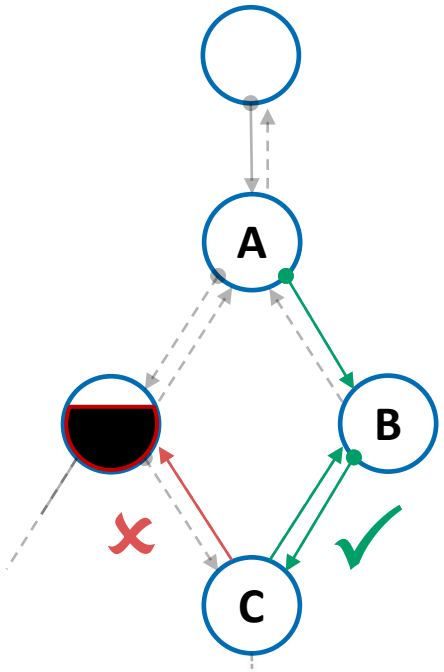
CFI: High-level idea

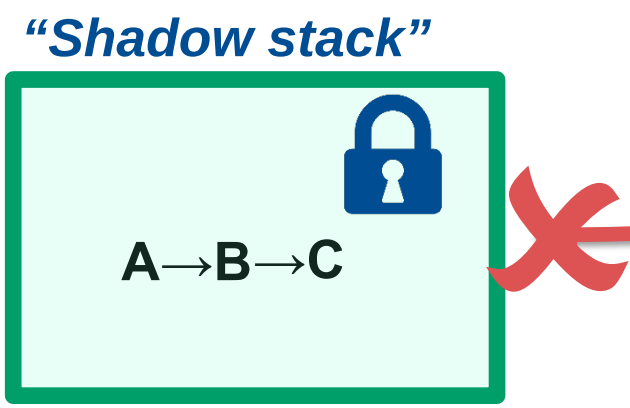


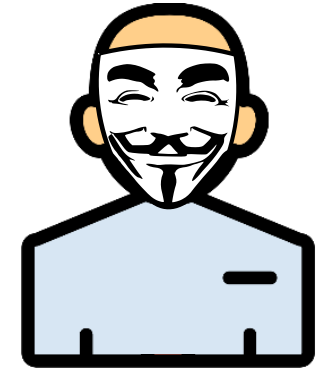
Legend:

- intended forward-edge in CFG
- intended backward-edge in CFG
- malicious edge not part of CFG
- initial node in CFG
- node in CFG

Shadow Stack: High-level idea







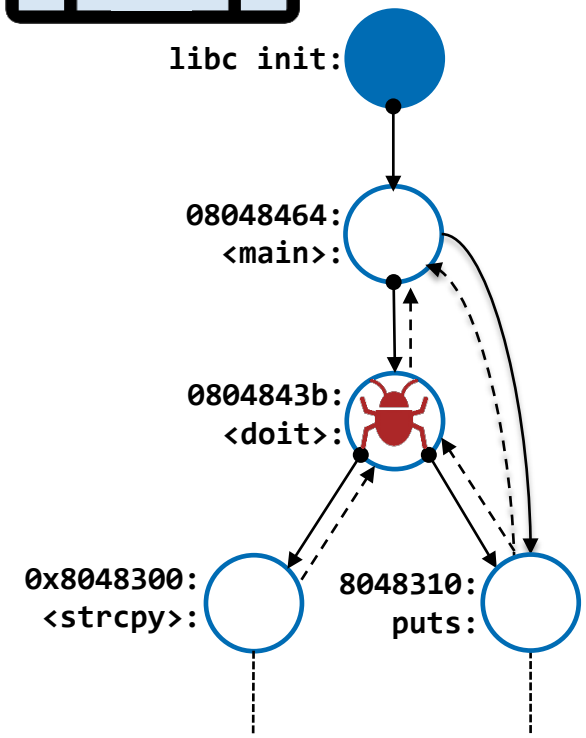
Adversary tampers with shadow stack

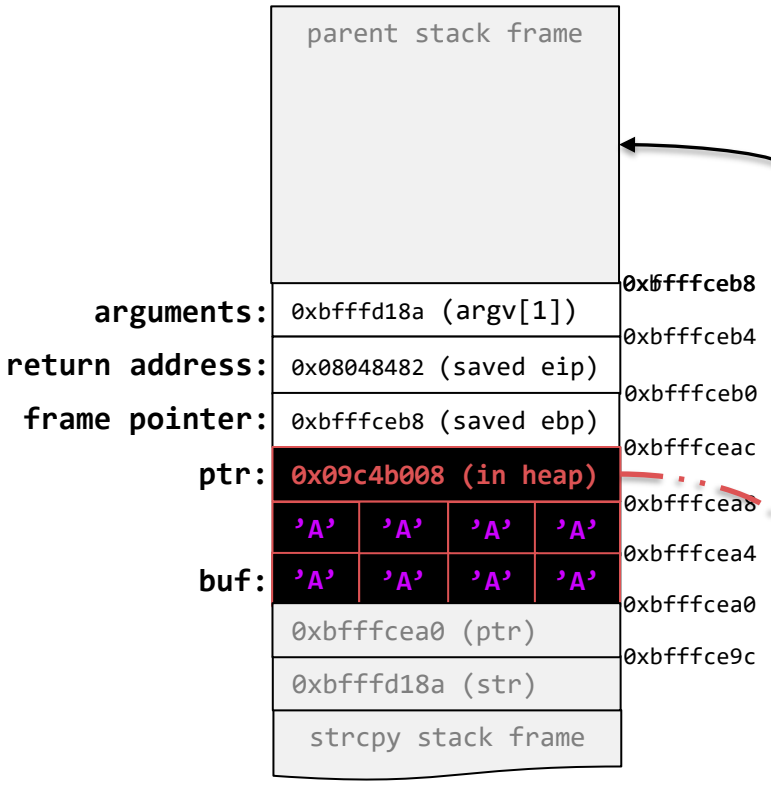
Non-control data attack

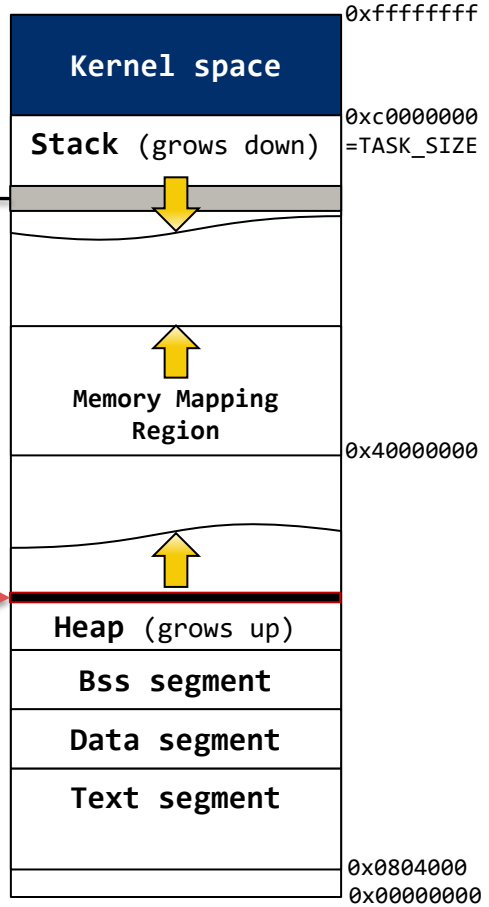
```
$ ./a.out $(perl -e 'print "A"x8 \
                ."\x08\xb0\xc4\x09" )
```

```
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```







Program logic that can be influenced as result of memory vulnerability constitute "*data-oriented gadgets*"

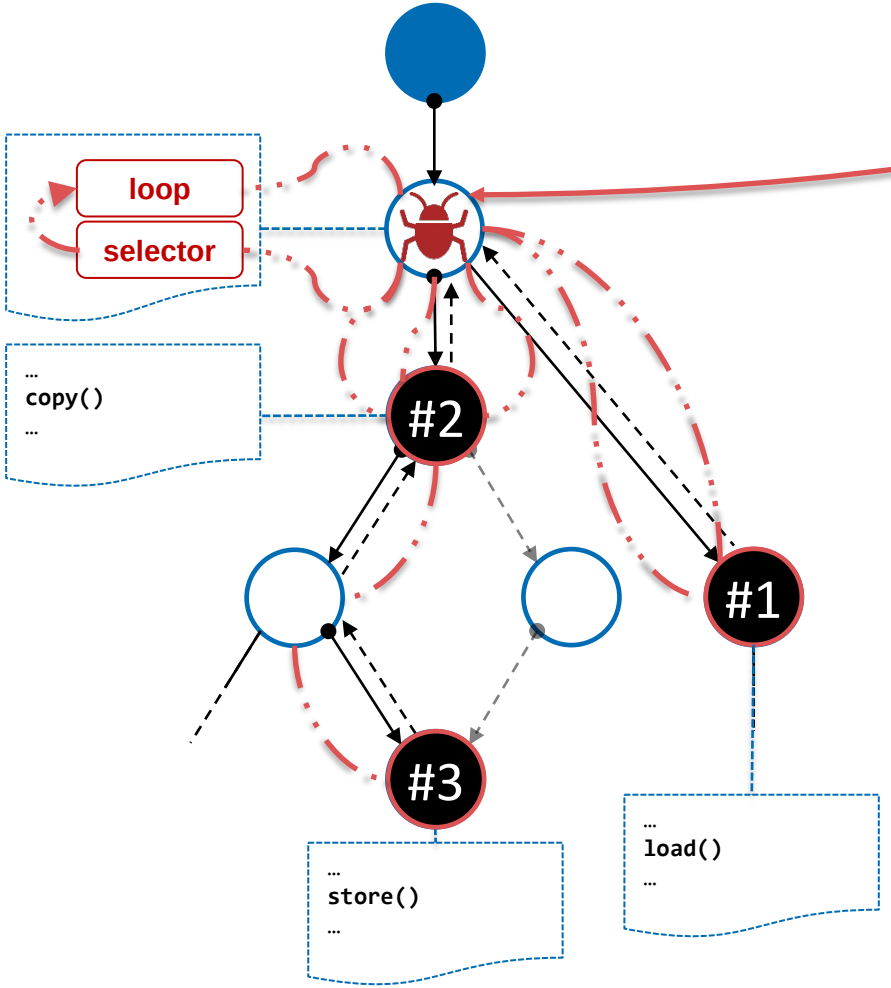
Attacker influences the behavior of benign program code without breaking control-flow integrity

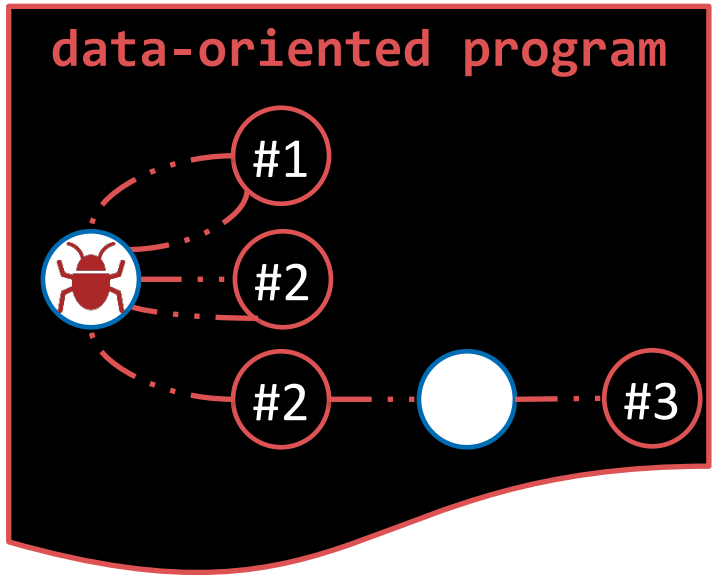
Data-oriented Programming

Enables expressive computation via use of "*data-oriented gadgets*" without diverging from program's benign control-flow

- **Requires a "gadget dispatch" that allows chaining together gadgets at will**

Adversary exploits bug

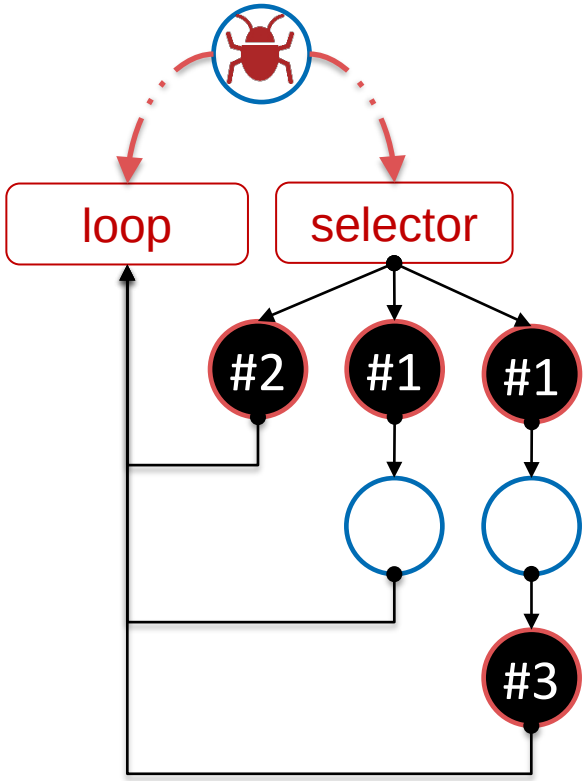


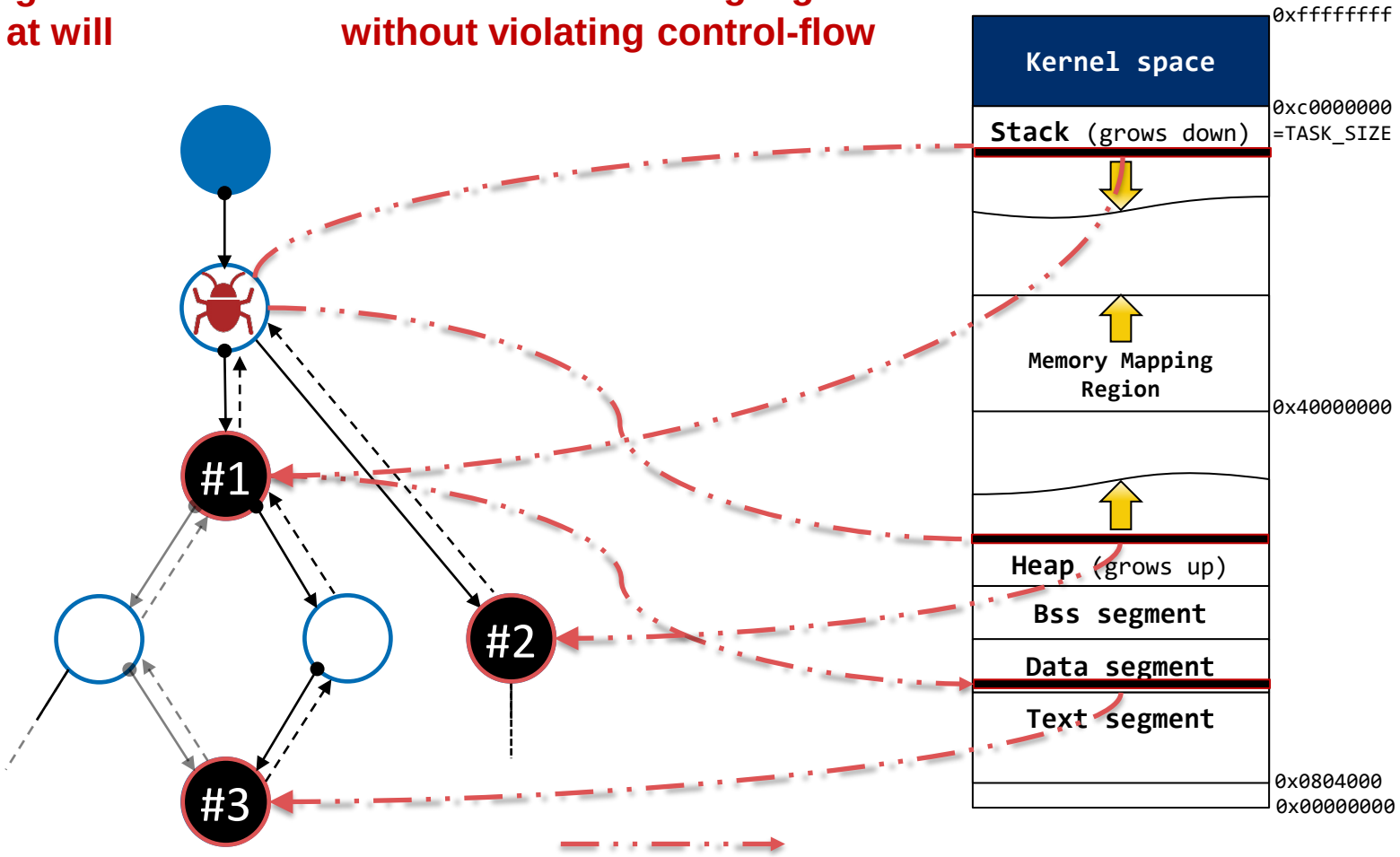


Data-oriented programming

Given a suitable gadget dispatch, an attacker can chain together data-oriented gadgets at will

Dispatch must be able to chain data-oriented gadgets without violating control-flow





corrupt data flow

Selected Research & Vulnerabilities

1988-99
- **ret2libc** — *Solar Designer (Phrack)*
- **Morris Worm:** RCE in `fingerd`
- **CVE-1999-1416:** DoS & RCE in Solaris Answerbook2
- **Format string vulnerabilities** — *Anders (Bugtraq. 1999)*

2001
- **Advanced ret2libc** — *Nergal (Phrack)*

2005
- **x86-64 borrowed code chunks exploitation** — *Krahmer*
- **Non-control-data attacks** — *Chen et al (SSYM. '05)*

2007
- **ROP on x86** — *Shacham (CCS'07)*

2008
- **ROP on ATMEL AVR** — *Francillon et al (CCS'08)*
- **ROP on SPARC** — *Buchanan et al (CCS'08)*

2009
- **ROP Rootkits** — *Hund et al (USENIX Sec. '09)*
- **ROP on PowerPC** — *FX Lindner (BlackHat USA)*
- **ROP on ARM / iOS** — *Miller et al (BlackHat Europe)*

2010
- **ROP w/o Returns** — *Checkoway et al (CCS'10)*
- **CVE-2010-3765**: *Nobel Peace Price website 0day*
- **CVE-2010-2883:** *RCE in Adobe Reader and Acrobat*

2011-12
- **CVE-2011-1938:** *RCE in PHP*
- **CVE-2012-0003:** *RCE in WMP MIDI library*
- **String-Oriented Programming** — *Payer (28C3. '11)*

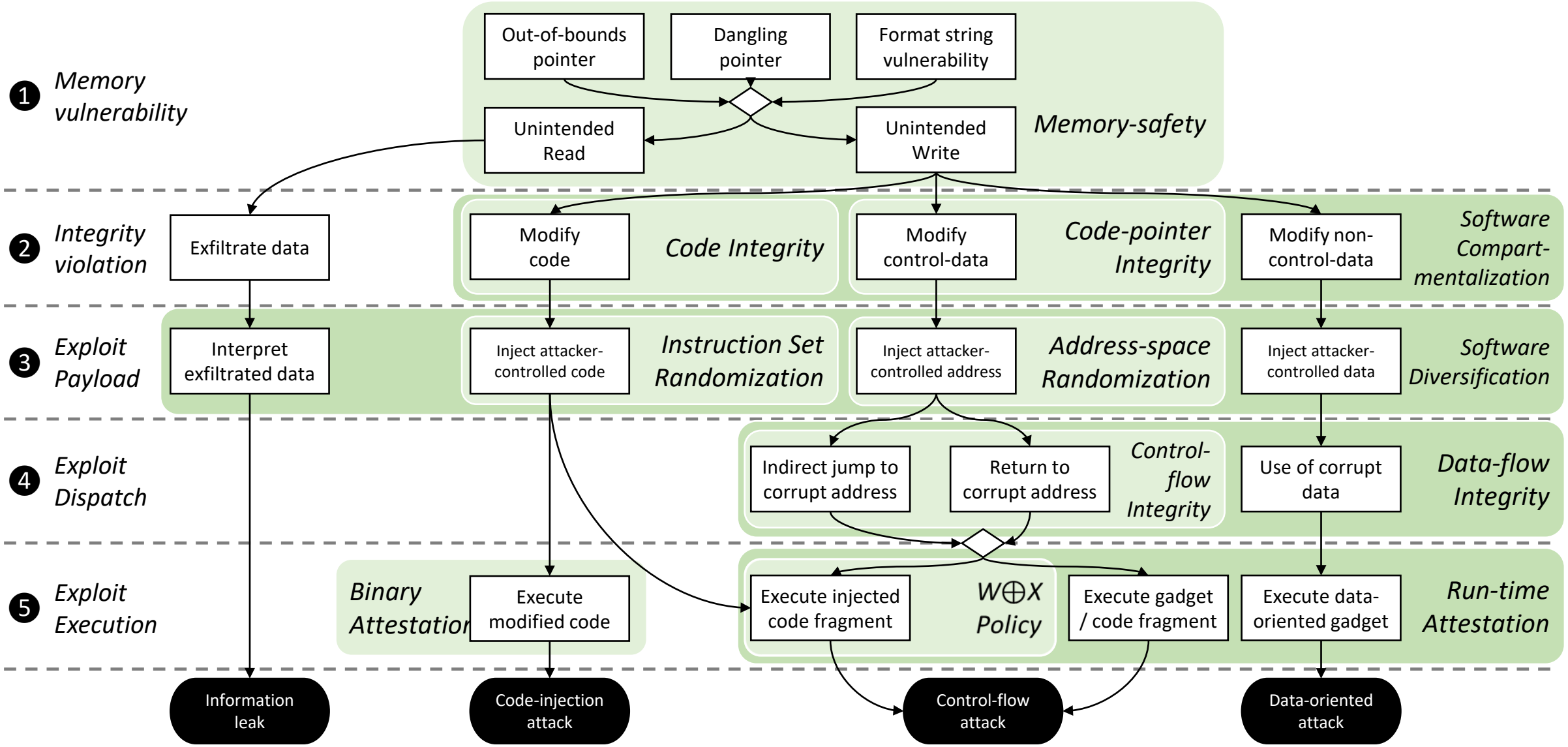
2013
- **JIT-ROP** — *Snow et al (IEEE S&P'13)*
- **CVE-2013-3893:** *RCE in Internet Explorer*
- **CVE-2014-9222:** *Misfortune cookie in RomPager*

2014
- **Blind ROP** — *Bittau et al (IEEE S&P'14)*
- **Stitching Gadgets** — *Davi et al (USENIX'14)*
- **CVE-2014-0160:** *Heartbleed vuln. in OpenSSL*
- **Write Once, Pwn Anywhere** — *Yu (BlackHat USA'14)*

2015
- **Out-of-Control** — *Göktas et al (IEEE S&P'14)*
- ***Gadget size Matters*** — *Göktas et al (USENIX'14)*
- **ROP is Still Dangerous** — *Carlini et al (USENIX'14)*
- **Data-Oriented Exploits** — *Hu et al (USENIX Sec.'15)*

2016
- **SROP** — *Bosman et al (IEEE S&P'14)*
- **Control-flow Bending** — *Carlini et al (USENIX Sec.'16)*
- **CVE-2016-0034:** *Angler RCE in Silverlight*
- **DOP** — *Hu et al (IEEE S&P '16)*

Taxonomy of Defenses



From **Thomas Nyman's doctoral dissertation, Towards Hardware-assisted Run-time Protection, 2020** (Figure Adapted from Szekeres et al., SoK: Eternal War in Memory, IEEE SP (2013))

Software, coarse-grained

Containers

chroot

Virtual machines

Software, fine-grained

Software CFI

Memory-safe languages

Privilege kernels

Part 2.

Harvard architecture

Memory Protection (MPU)

TEEs

Virtual memory (MMU)

Memory segments

W ⊕ X

Protection rings (Exception levels)

Enclaves

Hardware, coarse-grained

HW-assisted CFI

Tagged memory

Branch target indicators

HW shadow stack

Pointer Authentication

Fine-grained protection domains

HW-assisted bounds checks

Run-time scope enforcement

Hardware, fine-grained

Hardware-assisted defenses

How to thwart run-time attacks?

Run-time attacks are now routine

Software defenses incur security vs. cost tradeoffs

Hardware-assisted defenses are attractive

Protect against run-time attacks without incurring a significant performance penalty

Design new hardware-security mechanisms

Example: HardScope

Enforce variable visibility rules at run time

Mitigate effects of attacks that corrupt data-plane information

Digital design, FPGA realization, compiler instrumentation, extensive analysis

Deployment challenge:

- **Required the addition of 7 new instructions to the RISC-V ISA**

Nyman et al. HardScope: Hardening Embedded Systems Against Data-Oriented Attacks. DAC 2019

Hardware assisted defenses in CotS processors

ARMv8-A mechanisms

- Pointer Authentication (PA)
- Memory Tagging Extension (MTE)
- Branch Target Identification (BTI)

Intel x84_64 mechanisms

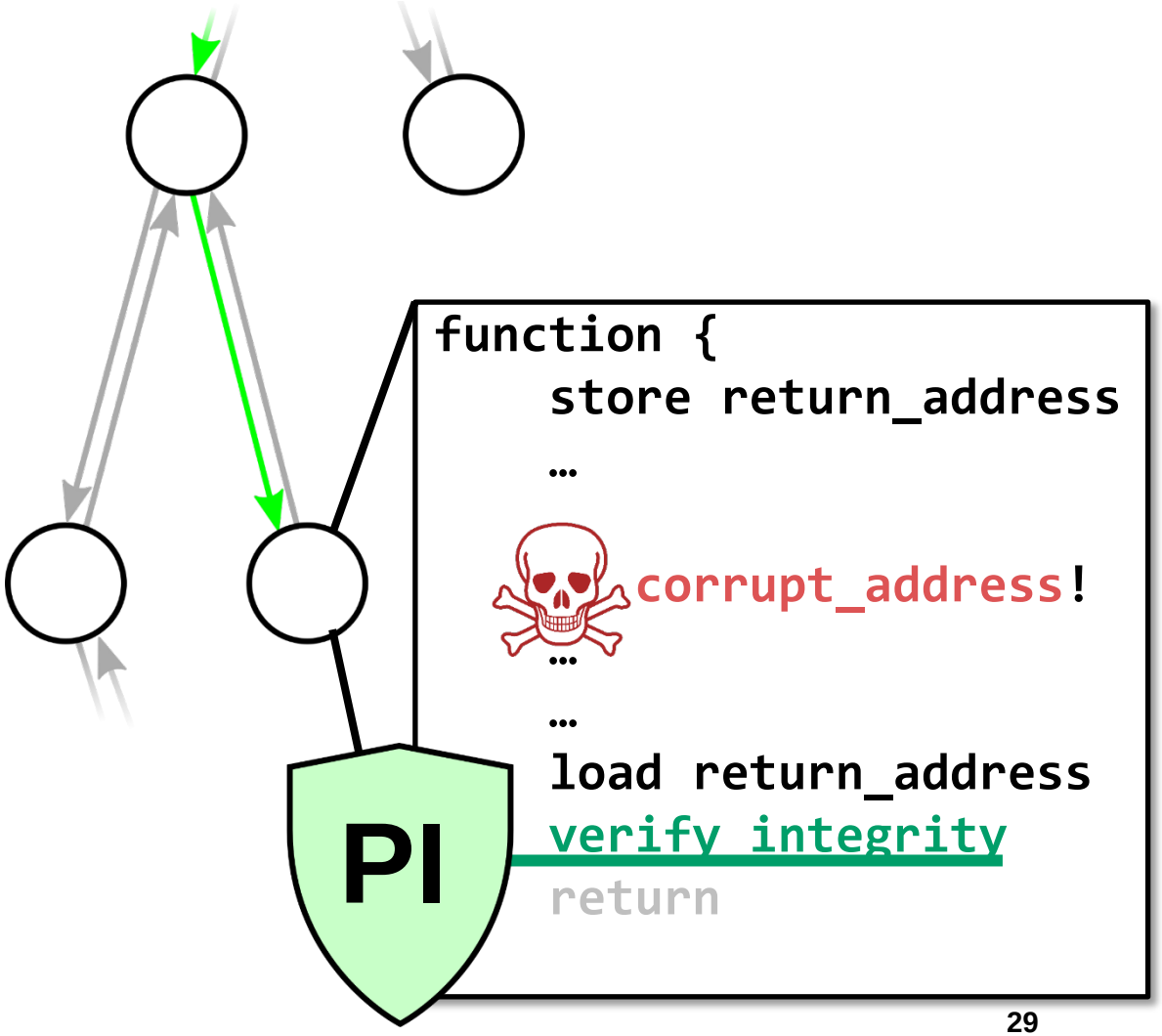
- Memory Protection eXtension (MPX)
- Memory Protection Keys (PKU)
- Control-flow Enforcement Technology (CET)

ARMv8-A mechanisms

Pointer Integrity: memory safety for pointers

Ensure pointers in memory remain unchanged

- **Code pointer integrity implies CFI**
  - Control-flow attacks manipulate code pointers
- **Data pointer integrity**
  - Reduces data-only attack surface



Kuznetsov et al. "Code-Pointer Integrity", USENIX OSDI 2014

ARMv8.3-A Pointer Authentication

General purpose hardware primitive approximating pointer integrity

- Ensure pointers in memory remain unchanged

Introduced in ARMv8.3-A specification (2016), **improved in ARMv8.6-A** (2020)

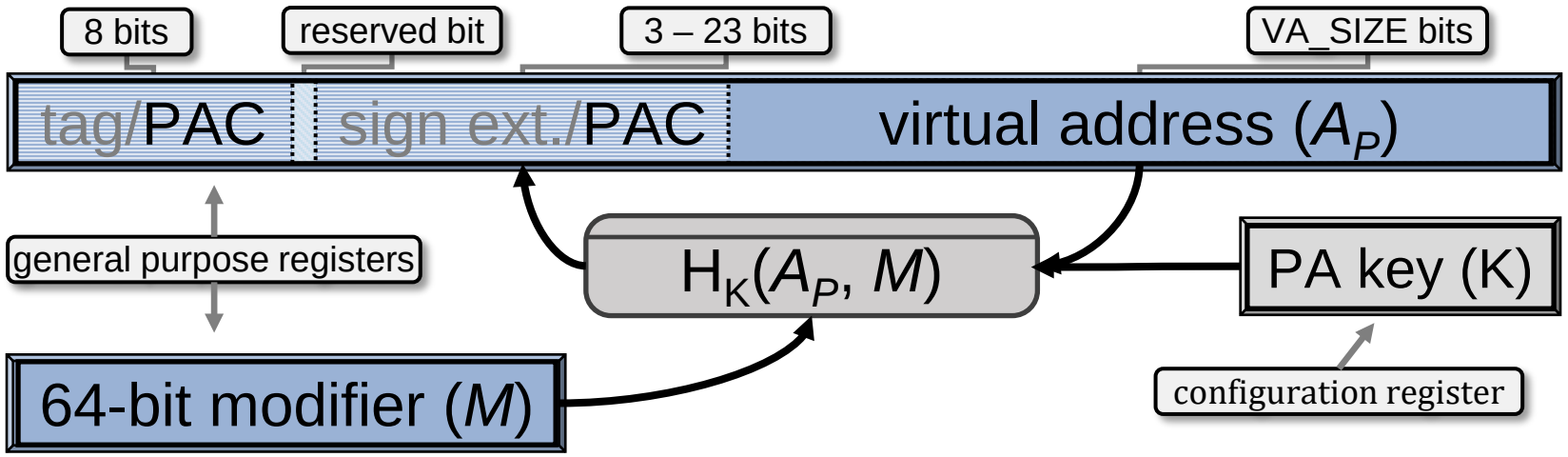
- First compatible processors 2018 (Apple A12 / iOS12)
- Userspace support in Linux 4.21, enhancements in 5.0, in-kernel support in 5.7
- Instrumentation support in GCC 7.0 (-msign-return address, deprecated in GCC 9.0 -mbranch-protection=pac-ret[+leaf] GCC 9.0 and newer)

ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019
ARM. Developments in the Arm A-Profile Architecture: Armv8.6-A. September 2019

ARMv8.3-A PA – PAC Generation

Adds Pointer Authentication Code (PAC) into unused bits of pointer

- Keyed, tweakable MAC from pointer address and 64-bit modifier
- PA keys protected by hardware, modifier decided where pointer created and used



ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019

ARMv8.3-A PA – Key management and instructions

Keys for PAC generation and verification

APIAKey_EL1	Key A for instruction address PACs
APIBKey_EL1	Key B for instruction address PACs
APDAKey_EL1	Key A for data address PACs
APDBKey_EL1	Key B for data address PACs
APGAKey_EL1	Key for generic authentication

PA Instructions

PAC<i|d><a|b> <Xd> <Xm>	Add PAC to address in *Xd* using modifier in *Xm*
AUT<i|d><a|b> <Xd> <Xm>	Authenticate address in *Xd* using modifier in *Xm*
PACGA <Xd> <Xn> <Xm>	Calculate generic PAC for data in *Xn* using modifier in *Xm*
XPAC<i|d> <Xd>	Strip PAC for address in *Xd*
BRA<a|b> <Xn> <Xm>	Branch to address in *Xn* after authenticating it with modifier in *Xm*
BLRA<a|b> <Xn> <Xm>	As BRA but perform branch with link
RETA<a|b>	Authenticate address in LR with SP as modifier and return
ERETA<a|b>	Authenticate address in ELR with SP as modifier and exception return
LDRA<a|b> <Xt> <Xn>	Authenticate address in *Xn* using modifier zero and load value to Xt

operate on instruction keys only

operate on data keys only

ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019

PA-based protection schemes

PA instructions are primitives, assembled to form protection schemes

Two main components:

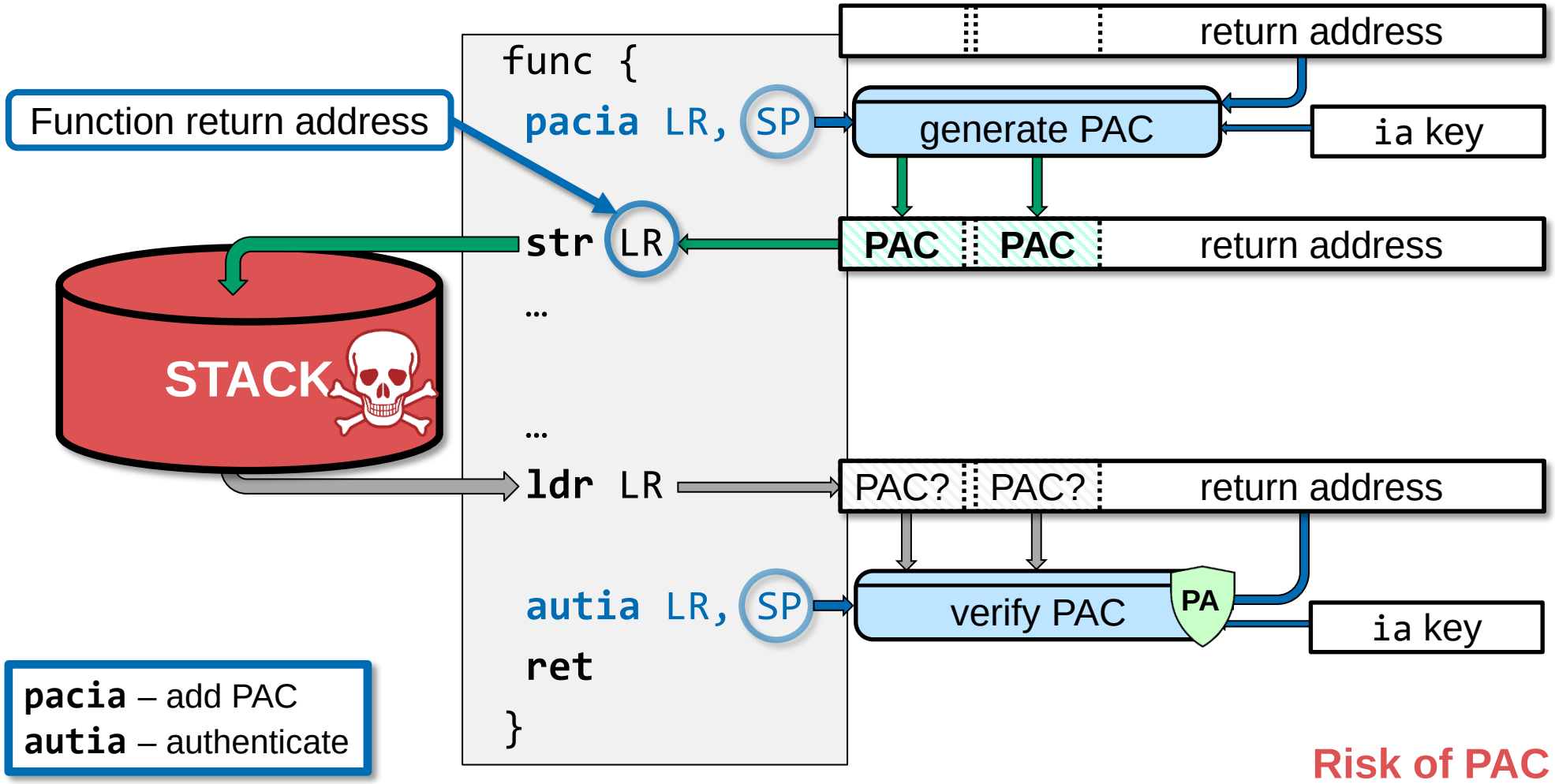
- When are pointers “PACed” and “unPACed”?
- Which modifier is used at a given point?

What should the modifier be for a given pointer?

- For security: using many different modifiers makes replay attacks harder
- For functionality: large numbers of modifiers are hard to keep track of

Example: `-msign-return-address`

Deployed in GCC 5.0 and LLVM/Clang 7.0



`pacia` – add PAC
`autia` – authenticate

Risk of PAC reuse!

Qualcomm "Pointer Authentication on ARMv8.3", whitepaper 2017

PA return address protection as a canary

The signed return address effectively is a canary:

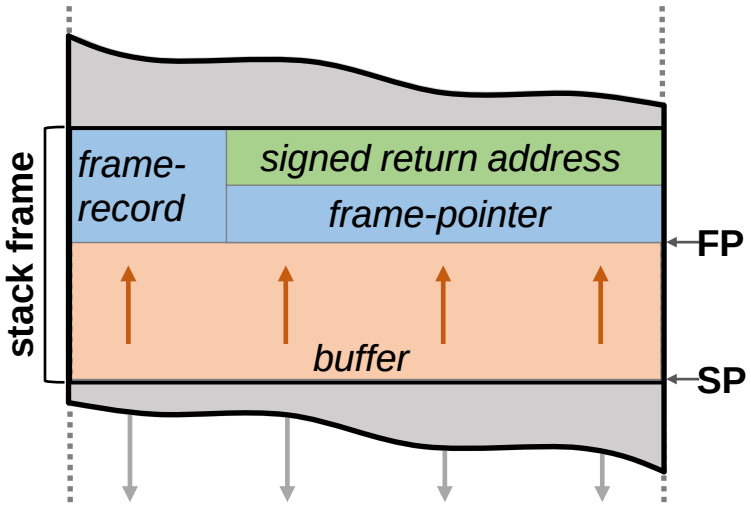
- Any overflow that corrupts the return address is detected

More powerful than `-stack-protector` canaries:

- Does not require reference value
- Can be bound to contextual information (e.g., the SP value)
- Protects return address against arbitrary writes

Also has similar weaknesses:

- Existing return addresses can be reused



Liljestrand et al, Protecting the stack with PACed canaries. SysTEX@SOSP 2019

PA only approximates fully-precise pointer integrity

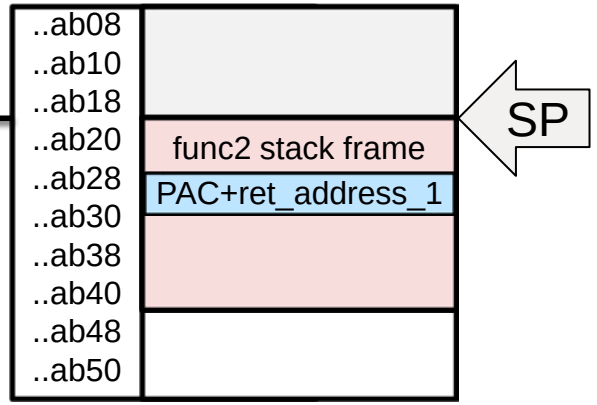
Adversary may reuse PACs

```
…
/* func1() */
brl %func1
…
/* func2() */
brl %func2
…
```

```
func1 {
 pacia LR, SP
 str LR
 …
```

```
func2 {
 pacia LR, SP
 str LR
 …
 ldr LR
 autia LR, SP
 ret
}
```

Not necessarily unique!



pacia – add PAC
autia – authenticate

PARTS

Modifier: based on pointer type

- Assigned at compile-time based on C type
- “this pointer really points to this type of data or function”

On-use or on-load authentication

- Branching with combined auth+branch instruction (**`blraa`**)
- Iterating an array uses only one authentication

```
// ptr = …
  …
  mov Xmod, #type_id
  pacia Xptr, Xmod
```

PACed only on pointer creation!

```
// *ptr
…
ldr Xptr, [Addr]
mov Xmod, #type_id
autda Xptr, Xmod
ldr Xd [Xptr]
```

Authenticated on load

```
// ptr();
  …
  …
  mov Xmod, #type_id
  blraa Xptr, Xmod
  …
```

Authenticated on use

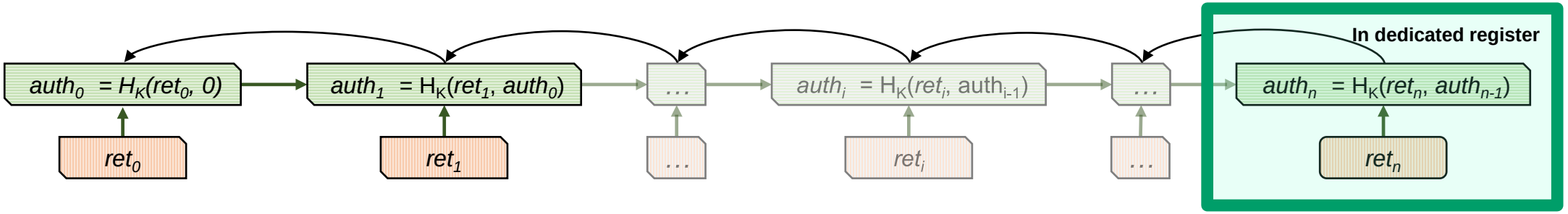
`pacda`	– add PAC with data A-key
`autda`	– authentic§ate
`pacia`	– add PAC with instr A-key
`blraa`	– authenticate and branch

Liljestrand et al. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication USENIX Security (2019)

Authenticated Call Stack: high-level idea

Chained MAC of authentications tokens cryptographically bound to return addresses

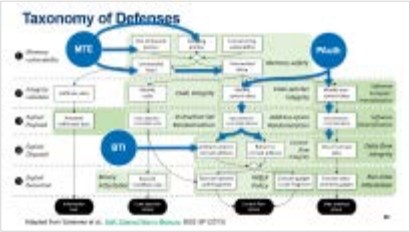
- Provides modifier (*auth*) bound to all previous return addresses on the call stack
- Statistically unique to control-flow path
  - prevents reuse
  - allows precise verification of returns



$auth_i$, $i \in [0, n-1]$ bound to corresponding return addresses, ret_i , $i \in [0, n]$, and $auth_n$

Liljestrand et al. PACStack: an Authenticated Call Stack. Usenix Security (2021)

Mitigation of hash-collisions: PAC masking



- **Challenge: PAC collisions occur on average after $1.253*2^{b/2}$ return addresses**
  - For b=16 n= 321 addresses

- **Solution: Prevent *recognizing* collisions by masking each *auth***
  - pseudo-random mask generated using `pacib(0x0,` $auth_{i-1}$)

Attack	w/o Masking	w/ Masking
Reuse previous auth collision	1	2^{-b}
Guess auth to existing call-site	2^{-b}	2^{-b}
Guess auth to arbitrary address	2^{-2b}	2^{-2b}

Maximum probability of success for different attacks

Liljestrand et al. PACStack: an Authenticated Call Stack. Usenix Security (2021)

ARMv8.5-A Memory Tagging Extension

Ensures memory accesses are safe by comparing tag in pointer with tag in memory

- Can prevent seqential buffer overflows, and (with high probability) other memory errors

Introduced in ARMv8.5-A specification (announced in 2018), no hardware currently

- Userspace support in Linux 5.10, to be enabled via PROT_MTE flag in mmap()
- Stack tagging in LLVM 9.0, heap tagging support planned
- Experimental support in Android 11 via LLVM's cudo memory allocator

ARM, Armv8.5-A Memory Tagging Extension, whitepaper 2019
ARM, Opensource support for Armv8.5-A Memory Tagging Extension, blog post 2019

ARMv8.5-A MTE

Address tags stored in top 4-bits of a pointer

- uses existing top-byte ignore (TBI) feature

Allocation tags stored transparently by hardware and cached

- 4-bit tag per 16-byte granule of memory

Mismatch between tags reported either:

- synchronously (precise check during testing), or
- asynchronously (imprecise checks after deployment)

ARM, Armv8.5-A Memory Tagging Extension, whitepaper 2019
ARM, Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile, Version F.c. July 2020

Example: Stack Tagging

- Choose random tag on function entry
- For each slot in stack frame choose tag at an offset to initial tag
- Accesses using immediate offset from SP are unchecked

```
int main(int argc, char *argv[])
{
            ...
    doit(argv[1]);
            ...

    return 0;
}
```

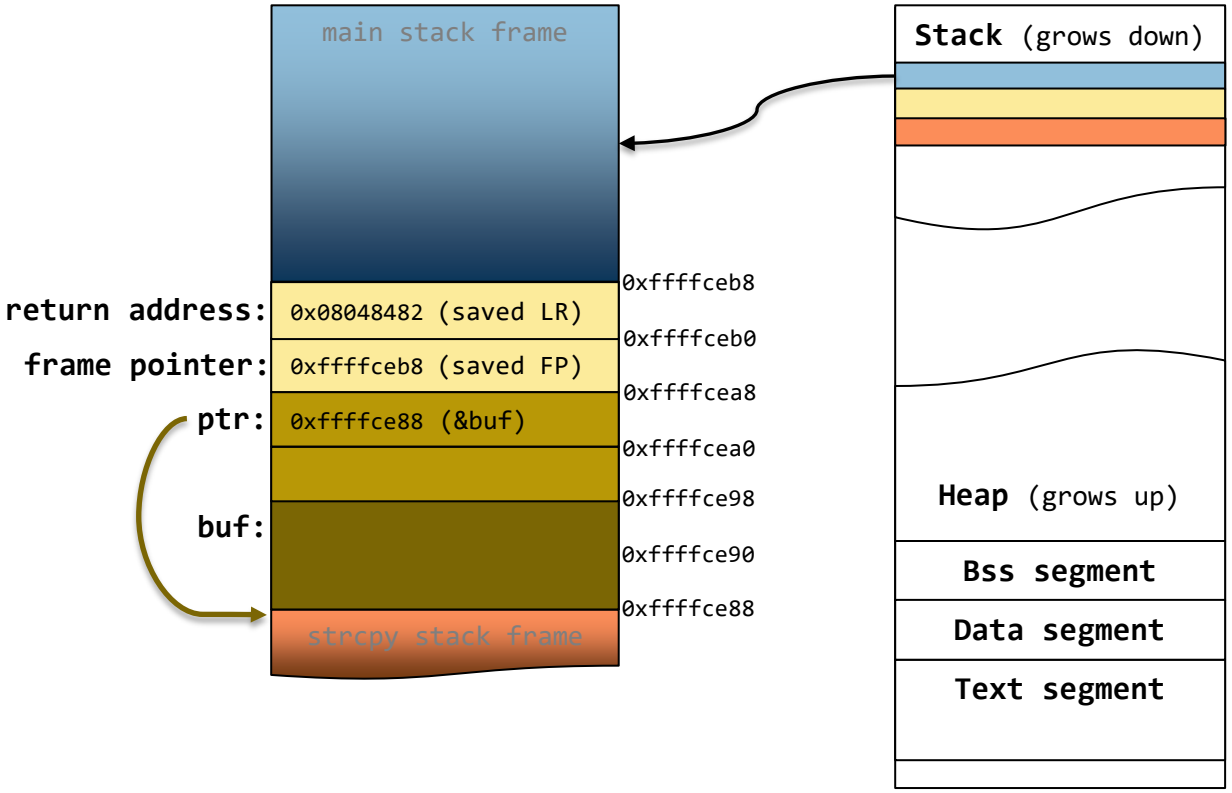
```
void doit(char *str)
{

    char buf[16];
    char ptr = buf;

    strcpy(ptr, str);
    puts(ptr);

}
```

```
void strcpy(char *str)
{
  ...
}
```



LLVM MemTagSanitizer

Random base tag for each stack frame

- Slots sequentially tagged to minimize tag book-keeping
- Uses Stack Safety Analysis to optimize instrumentation

Globals tagging requires loader support to assign initial tags

Heap tagging planned via the new secure Scudo allocator

Provides:

- Deterministic prevention of sequential overflows
- Probabilistic detection of use-after-free and non-sequential out-of-bounds
  - In the general case: $1-2^{-4} \approx 0.94$ chance of detection

E. Stepanov *et al.*, Memory tagging in LLVM and Android, LLVM Developers' Meeting (2020)
LLVM, MemTagSanitizer, online documentation
LLVM, Stack Safety Analysis, online documentation

LLVM Stack Safety Analysis

Introduced by Kuznetsov *et al*. but also used to optimize MTE instrumentation

Memory safety loosely defined as:

A memory object is safe, if all pointers derived from it are guaranteed to only access the memory object itself.

Does not preclude safe object corruption

- by unrelated unsafe memory accesses
- by within-allocation memory corruption

Algorithm finds access range of pointers

- **If range is within allocated memory, then allocation is provably safe**
- **Local analysis**
  - Determines local use ranges for allocations and function arguments
- **Global analysis**
  - Merges ranges from function arguments
  - Runs until fixed point reached

Pointers in memory assumed unsafe!

V. Kuznetsov *et al.*, Code-pointer integrity, OSDI (2014)
R. Gil *et al.*, There's a hole in the bottom of the C: on the effectiveness of allocation protection, SecDev (2018)

MTE (and MemTagSanitizer) challenges

Tags are corruptible

- Random tags prevent hard-coding
- Adversary can inject tagged pointers
  - Safe memory tags always known!
  - Guessing probability 2^{-4} with short 4-bit tags

Analysis is not MTE aware

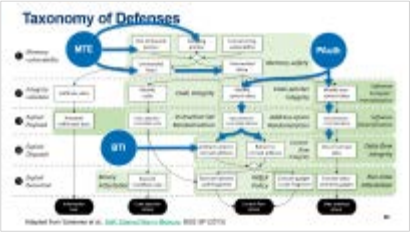
- Assume pointers in memory are unsafe

Unclear security properties

- Probabilistic, but sometimes not
- No hard guarantees with tag corruption

LLVM, MemTagSanitizer, online documentation

Our goal



Prevent tag forgery

- Enforce any pointer loaded from unsafe memory is recognized as unsafe

Leverage MTE-awareness in safety analysis

- Introduce MTE-specific protected domain (in addition to safe / unsafe domains)
- Prove/make a larger set of allocations as safe (better optimization)

Provide clearer security guarantees

- Allow programmer to indicate variables that must remain safe
- Provide concrete guarantees for variables designated as safe by the compiler

Preventing tag corruption

Tags can be enforced to achieve hard guarantees!

Always set one tag-bit on load from memory

- Prevents injection of “safe” tags (costs one tag-bit)

Alternatively use ARM Pointer Authentication

- Probabilistic but does not reserve one tag-bit

```
// Load ptr1 from ptr2
char *ptr1 = *ptr2;
// Apply ptr2 tag bits to ptr1
ptr1 = ptr1 | (tag & ptr2);
```

➔ pointers in safe memory can remain uncorrupted!

MTE-aware analysis

MTE can be used to prevent sequential overflows

- **Surround with different tag (either other allocation or dedicated memory guard)**

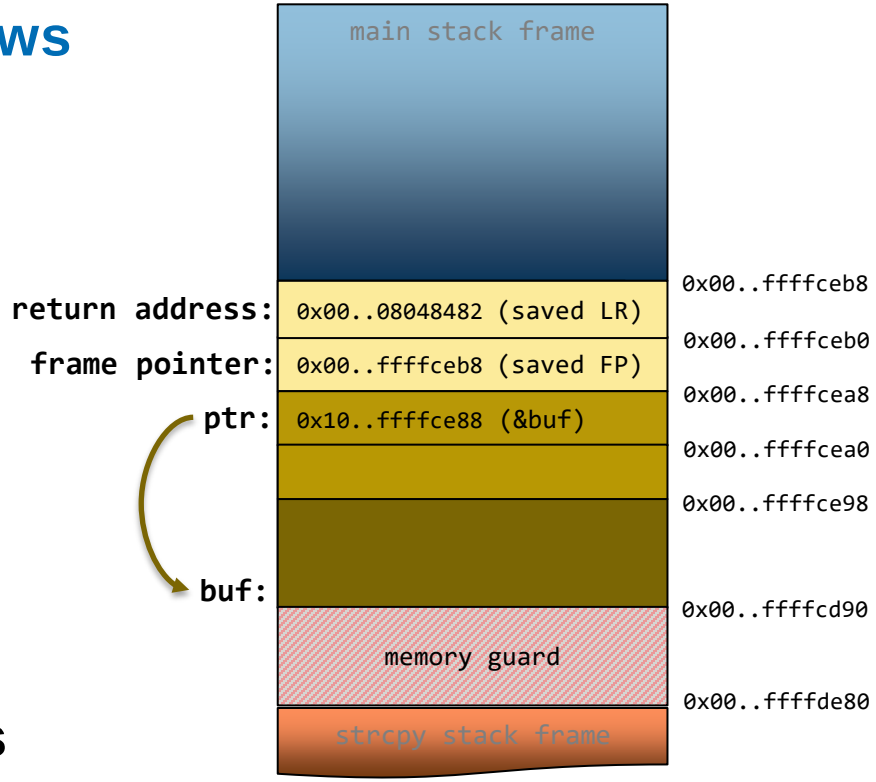
Can be treated as if memory safe:

New protected memory safe domain

- violation will lead to crash
- tagging of allocation itself can be omitted

Analysis checks that for any sequential access

- *start_range ⊆ allocated_memory*
- *max_step < memory_guard_size*



Analysis of in-memory pointers

Pointer safety requires storage location is safe

- **Must prove safety of pointer within storage**
  - In adiditon to allocation-based safety
- **Must find all subsequent loads of pointer**
  - Requires point-to analysis in general case
  - Non-linear data-flow through globals / heap

```
struct s { char buff[32]; char *ptr; };
// sizeof(struct s) = 40

struct s store;
struct s DATA;

store.ptr = &DATA;

for (i = 0; i < 40; ++i)
  store.buff[i] = get_char();

g_ptr = store.ptr;  // store in global
char *c = &store;
func(store.ptr);
```

store is safe

Is store pointer safe?

Where is &DATA used?

R. Gil *et al.*, There's a hole in the bottom of the C: on the effectiveness of allocation protection, SecDev (2018)

MTE-aware analysis with in-memory pointers

Conservative approximation of pointer-safety

- **Check in-allocation bounds based on type**
- **Assumes non-local stores unsafe**
- **Assumes non-typed use is unsafe**

Allows lightweight data-flow analysis

- Without dependence on full points-to analysis

```
struct s { char buff[32]; char *ptr; };
// sizeof(struct s) = 40

struct s store;
struct s DATA;

store.ptr = &DATA;

for (i = 0; i < 40; ++i)
  store.buff[i] = get_char();

g_ptr = store.ptr;  // store in global
char *c = &store;
func(store.ptr);
```

Is `store` pointer safe?

`&store.buff` out of bounds

`&DATA` leaked to global

type information lost

Need to merge function use with `DATA` use

ARMv8.5-A Branch Target Identification

Hardware-assisted CFI similar to Intel CET Indirect Branch Tracking

Introduced in ARMv8.5-A specification (2016)

- Support in Linux 5.8
- Instrumentation support in GCC 9.0 (-mbranch-protection=standard|bti)

ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019

ARMv8.5-A BTI

Indirect branches to guarded code regions require marker instructions

- compiler places marker potential indirect branch targets
- two classes of targets: calls and jumps (`RET` instructions not restricted by BTI)

Branch sources

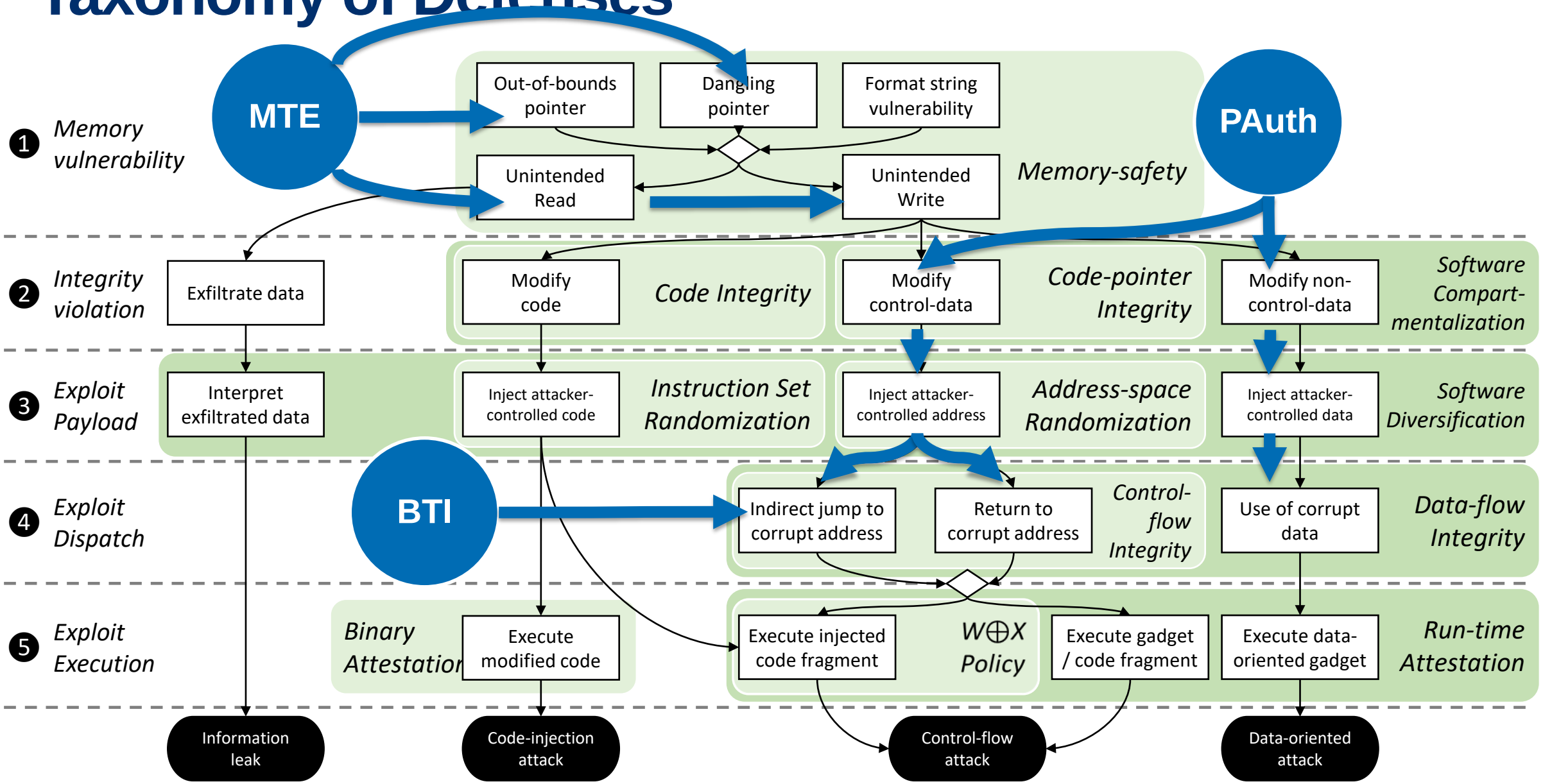
BTI call type branches	
`BLR …`	Indirect function calls
`BR <x16|x17>`	PLT entries and tail calls
BTI jump type branches	
`BR … (except x16|x17)`	Branches to jump tables

BTI Marker Instructions

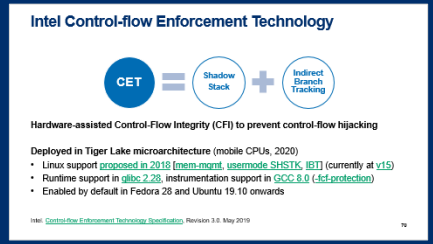
`BTI <c|j|cj>`	Branch Target Identification for c=calls, j=jumps, cj=calls or jumps
`BRK`	Breakpoint Instruction
`HLT`	Halting breakpoint
`PACIASP / PACIBSP`	Create PAC for Instruction address in LR using key A/B and SP as modifier

ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019

Taxonomy of Defenses



Adapted from Szekeres et al., SoK: Eternal War in Memory, IEEE SP (2013)



Intel x86_64 mechanisms

Skip to CET

Intel Memory Protection Extension

Run-time checks for memory accesses to detect pointer bounds violations

Deployed in SkyLake microarchitecture (2015)

- Support in Linux 3.9, removed in 5.6
- Instrumentation support in GCC 5.0, removed in 9.0 (-fcheck-pointer-bounds)

Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. Volume 1. Chapter 17. May 2019

Intel MPX – Example

```
struct obj { char buf[100];  int len }
obj* a[10];
1: for (i=0; i<M; i++) {
2:     total += a[i]->len;
3: }
```

```
1: obj* a[10]                    // Array of pointers to objs

2: total = 0
3: for (i=0; i<M; i++):
4:     ai = a + i                // Pointer arithmetic on a
5:     objptr   = load ai        // Pointer to obj at a[i]
6:     lenptr   = objptr + 100   // Pointer to obj.len
7:     len   = load lenptr
8:     total += len              // Total length of all objs
```

Intel MPX – Bound Check Instructions

MPX Registers

BND00 – BND03	Bound Registers storing 64-bit LowerBound (LB) and 64-bit UpperBound (UB)

MPX Instructions

BNDMK	<*Reg*>	<*Addr*> <*offset*>	Create LB (*Addr*) and UB (*Addr + offset*) in *Reg*
			Bound Check Instructions
BNDCL	<*Reg*>	<*Addr*>	Check *Addr* against LB in *Reg* .
BNDCU	<*Reg*>	<*Addr*>	Check *Addr* against UB in *Reg* in 1's compliment form
BNDCN	<*Reg*>	<*Addr*>	Check *Addr* against UB in *Reg* not in 1's compliment form
			Bound Management Instructions
BNDMV	<*Reg*>	<*Reg*|*Addr*>	Copy LB and UB from *Reg* *or* *Addr* *to* *Reg*
BNDMV	<*Addr*>	<*Reg*>	Store LB and UB from *Reg* *to* *Addr*
BNDLDX	<*Reg*>	<*SIB*>	Load LB and UB from bound directory
BNDSTX	<*SIB*>	<*Reg*>	Store LB and UB to bound directory

Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. Volume 1. Chapter 17.4. May 2019

Intel MPX – Example

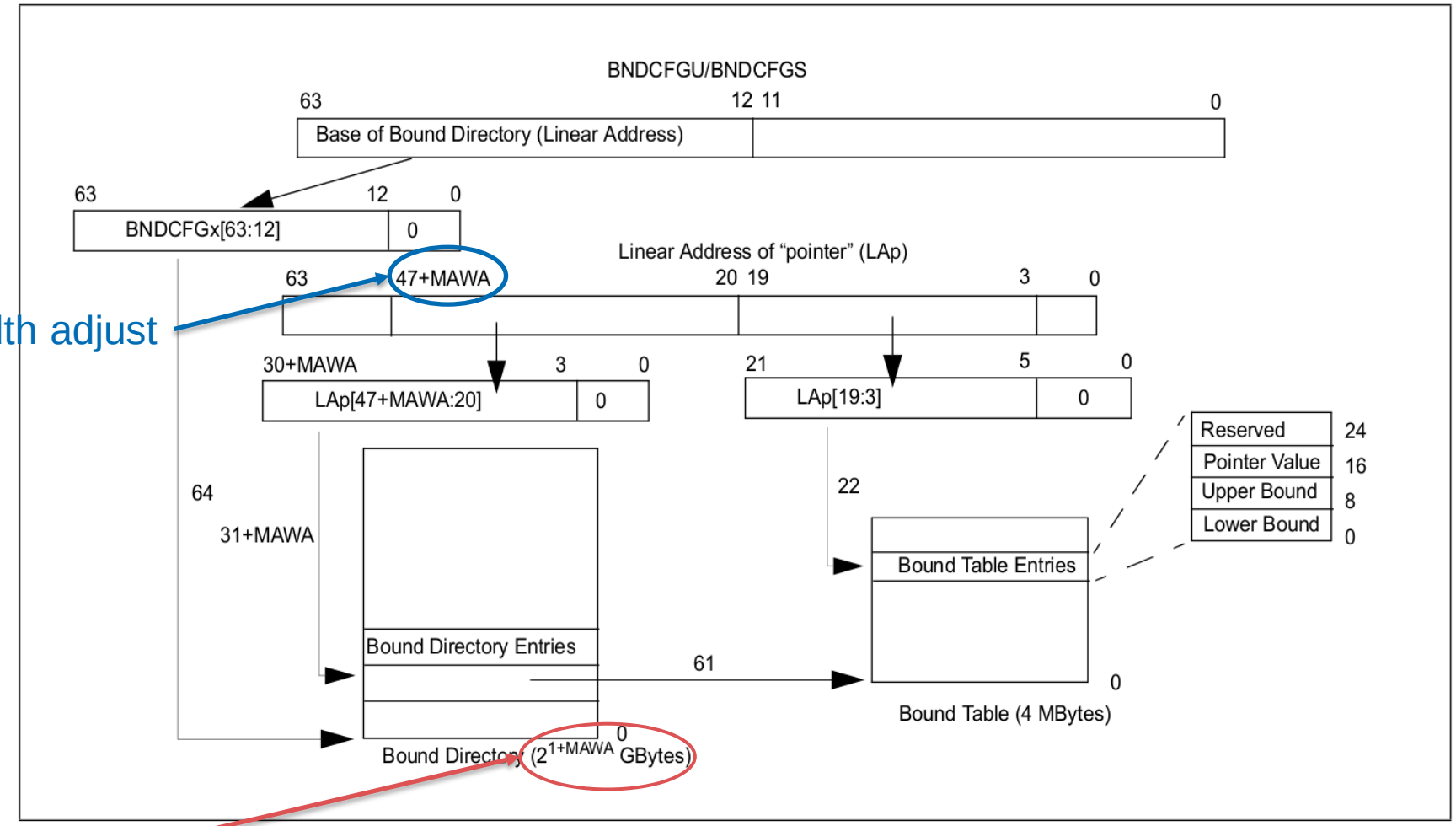
```
struct obj { char buf[100];  int len }
obj* a[10]
for (i=0; i<M; i++) {
    total += a[i]->len;
}
```

```
obj* a[10]                      // Array of pointers to objs
a_b = bndmk a, a+79             // Make bounds [a, a+79]
total = 0
for (i=0; i<M; i++):
    ai = a + i                  // Pointer arithmetic on a
    bndcl a_b, ai               // LowerBound check of a[i]
    bndcu a_b, ai+7             // UpperBound check of a[i]
    objptr   = load ai          // Pointer to obj at a[i]
    objptr_b = bndldx ai        // Bounds for pointer at a[i]
    lenptr   = objptr + 100     // Pointer to obj.len
    bndcl objptr_b, lenptr      // Check LowerBound of obj.len
    bndcu objptr_b, lenptr+3    // Check UpperBound of obj.len
    len   = load lenptr
    total += len                // Total length of all objs
```


Oleksenko et al. Design of Intel MPX. Web. 2018.
Oleksenko et al. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. SIGMETRICS ’18

Intel MPX – Bound Directory

MPX address-width adjust

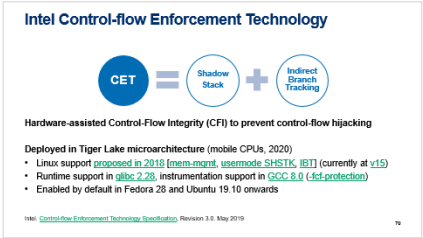


2^{1+MAWA} GBytes

Figure 17-4. Bound Paging Structure and Address Translation in 64-Bit Mode

Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. Volume 1. Chapter 17.4.3.1 May 2019

Intel MPX – Limitations



Study by Oleksenko et al. identified the following limitations:

- overhead comparable to software-based (up to 4x slowdown, ~50% overhead on average)
- no protection against temporal memory safety errors (e.g. use-after-free)
- no support for multithreading, can lead to unsafe data races between threads
- no support for common C/C++ idioms due to memory layout restrictions
- conflicts with other ISA extensions (Intel TSX, SGX)
- instrumentation incurs significant performance penalty (> 15%) even if MPX not available

Susceptible to Bounds Check Bypass due to Meltdown speculative execution attack

- exploits lazy handling of raised bound range (#BR) exception

Oleksenko et al. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. SIGMETRICS ’18
Canella et al. A Systematic Evaluation of Transient Execution Attacks and Defenses. USENIX Sec ‘19

Skip to Intel CET

Adapting MPX for kernel code

Bound Directory cannot be used in kernel code

- kernel cannot handle page faults at arbitrary points within its own execution
- pre-allocating the bound directory and bound tables not feasible due to memory overhead
  bound directory for 64-bit kernel is 2^{28} 64-bit entries = 2 Gbytes
  each bound table 2^{17} 32-byte entries = 4 Mbytes

Solution: dynamically determine pointer bounds using existing kernel metadata

https://ssg.aalto.fi/research/projects/kernel-hardening/

Reshetova et al. Towards Linux Kernel Memory safety. Software: Practice and Experience, Volume 48, Issue12. 2018

Intel Protection Keys for Userspace

PKU

User-level memory access-control mechanism at page granularity

- Associates each memory page with a 4-bit protection key kept in page table entry
- Access control rules for protection keys maintained by userspace code in PKRU register

Deployed in SkyLake microarchitecture (server configuration, 2015)

- Support in Linux 4.6 (known as MPK)
- Userspace support in GNU C Library (glibc) 2.27, GCC 5.3 (pkey_mprotect)

Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. Volume 1. Chapter 2.7., 4.6.2 May 2019

Intel PKU – Instructions

PKU Registers

PKRU	Protection Key Right Register

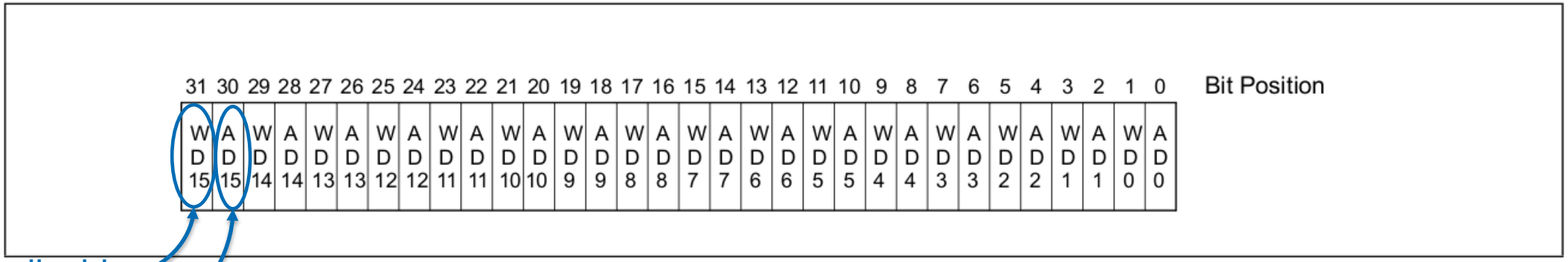


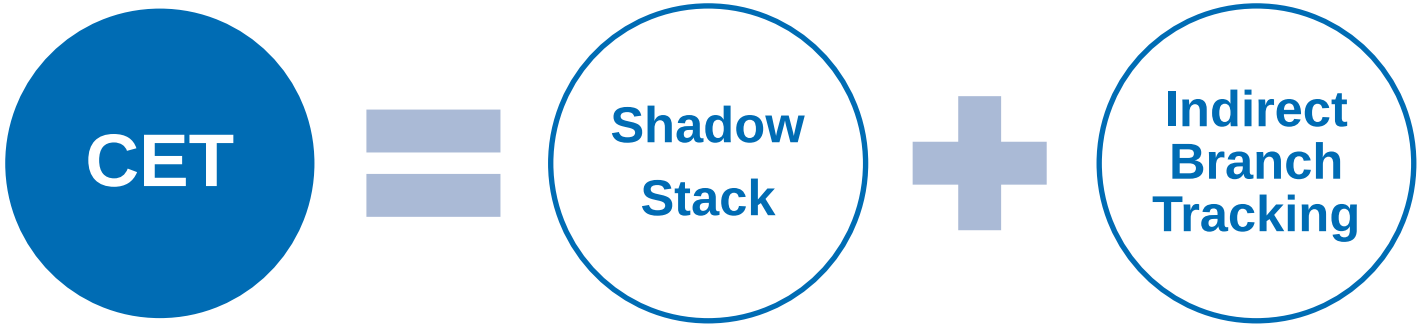
Figure 2-9. Protection Key Rights Register for User Pages (PKRU)

PKU Instructions

RDPKRU	Read PKRU value to EAX
WRPKRU	Write EAX value to PKRU

Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. Volume 1. Chapter 2.7., 4.6.2 May 2019

Intel Control-flow Enforcement Technology



Hardware-assisted Control-Flow Integrity (CFI) to prevent control-flow hijacking

Deployed in Tiger Lake microarchitecture (mobile CPUs, 2020)

- Linux support proposed in 2018 [mem-mgmt, usermode SHSTK, IBT] (currently at v15)
- Runtime support in glibc 2.28, instrumentation support in GCC 8.0 (-fcf-protection)
- Enabled by default in Fedora 28 and Ubuntu 19.10 onwards

Intel. Control-flow Enforcement Technology Specification, Revision 3.0. May 2019

Intel CET – Shadow Stack

Mechanism for protecting return address stored on the call stack

- introduces second stack used exclusively for copies of return addresses
- return address popped from both stacks on return and compared

Writes to shadow stack restricted to control-flow and management instructions

- shadow stack pages protected by page table protections (additional “shadow stack” attr)
- page protection also prevents overflow and underflow of shadow stack

New architectural register: Shadow Stack Pointer

- Cannot be directly encoded as source, destination or memory operand by instructions

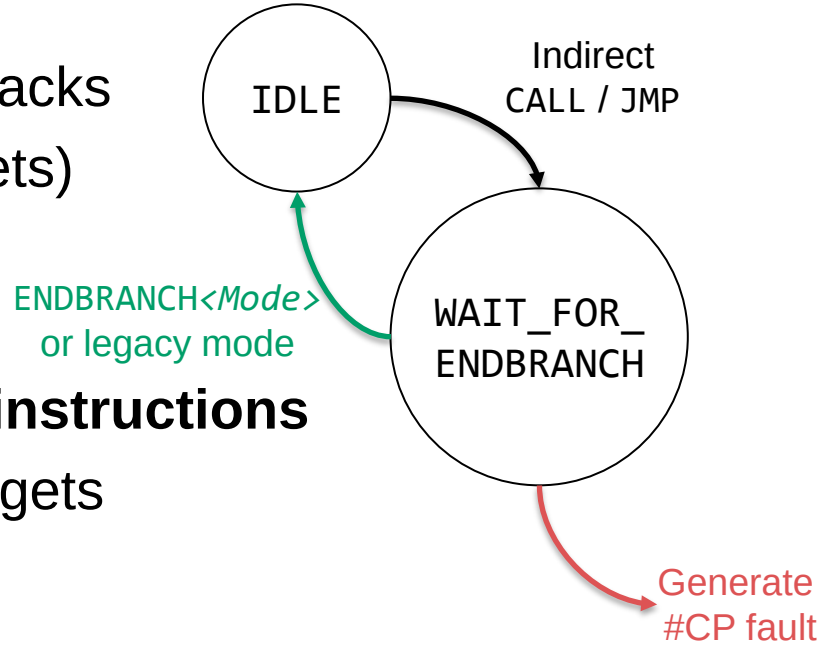
Intel. Control-flow Enforcement Technology Specification, Revision 3.0. Chapter 2. May 2019

Skip to taxonomy

Intel CET – Indirect Branch Tracking

Prevents diverting indirect CALL/JMP to invalid targets

- typical attack vector in call/jmp-oriented programming attacks
- achieves only weak CFI guarantees (single class of targets)



Requires indirect JMP / CALL to target specific marker instructions

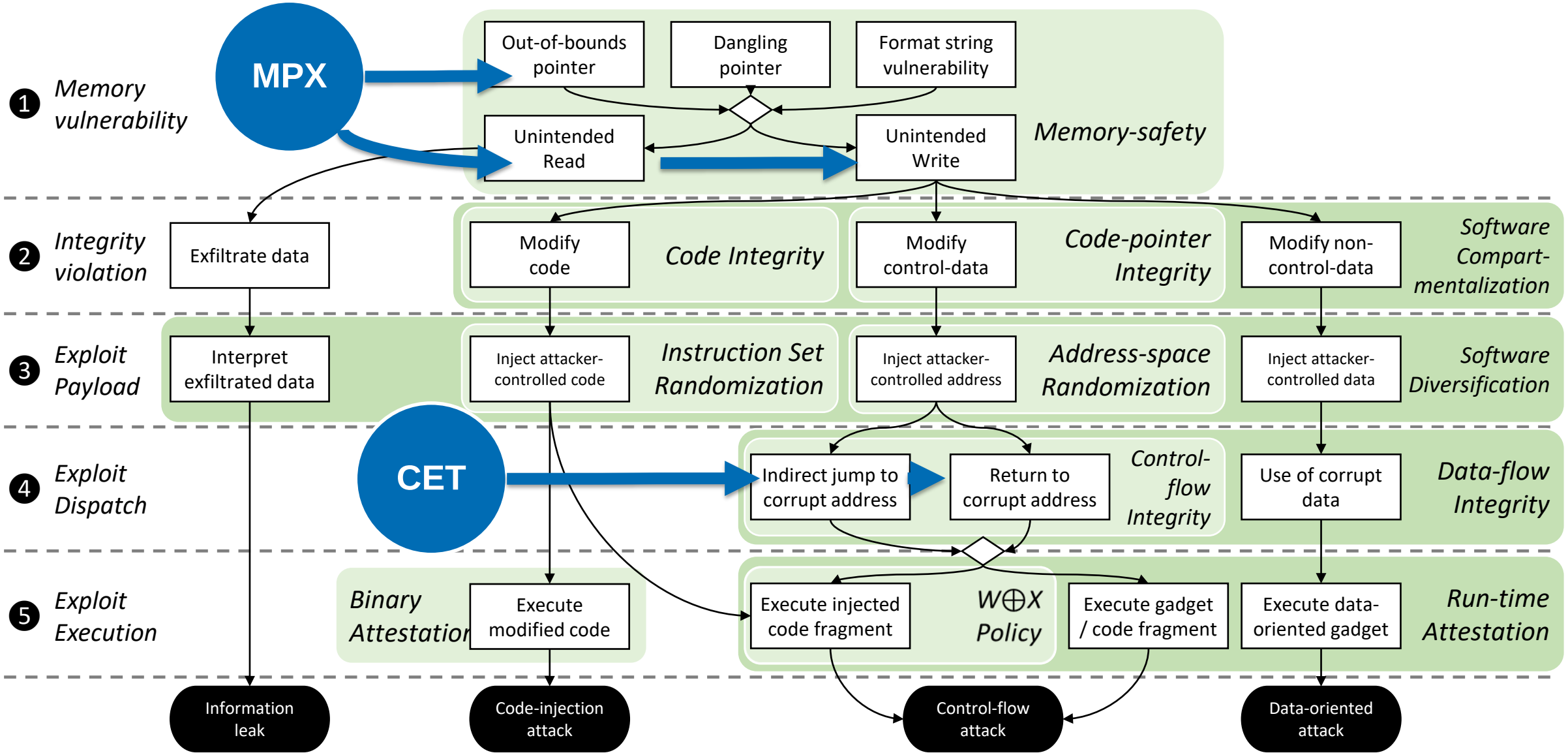
- compiler places marker at all potential indirect branch targets
- new control-protection exception (#CP) raised otherwise

IBT Marker Instructions

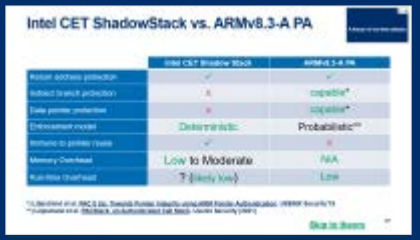
ENDBRANCH32	Marker instruction in 32-bit mode
ENDBRANCH64	Marker instruction in 64-bit mode

Intel. Control-flow Enforcement Technology Specification, Revision 3.0. Chapter 3. May 2019

Taxonomy of Defenses



Adapted from Szekeres et al., SoK: Eternal War in Memory, IEEE SP (2013)



Intel CET ShadowStack vs. ARMv8.3-A PA

	Intel CET Shadow Stack	ARMv8.3-A PA
Return address protection	✓	✓
Indirect branch protection	x	capable*
Data pointer protection	x	capable*
Enforcement model	Deterministic	Probabilistic**
Returns-to-pointer-reuse	✓	x
Memory Overhead	Low to Moderate	NA
Run-time Overhead	? (likely low)	Low

*Is identified in ref. PAC It Up: Towards Pointer Integrity using ARM Pointer Authentication, USENIX Security '19
**Is explained in ref. Mitigating Jump-Oriented Programming Attacks, Vienna Security (VSS)

Skip to Beyond

Comparison

Skip to PA vs CET

Intel MPX vs. ARMv8.5-A MTE

	Intel MPX	ARMv8.5-A MTE
Spatial error protection	✓	✓
Temporal error protection	✗	✓
Enforcement model	Deterministic	Probabilistic (16 classes)
Memory Overhead	High	?
Run-time Overhead	Moderate to High	?

Intel CET ShadowStack vs. ARMv8.3-A PA

	Intel CET Shadow Stack	ARMv8.3-A PA
Return address protection	✓	✓
Indirect branch protection	✗	capable*
Data pointer protection	✗	capable*
Enforcement model	Deterministic	Probabilistic**
Immune to pointer reuse	✓	✗
Memory Overhead	Low to Moderate	N/A
Run-time Overhead	? (likely low)	Low

*) Liljestrand et al. PAC It Up: Towards Pointer Integrity using ARM Pointer Authentication. USENIX Security'19
**) Liljestrand et al. PACStack: an Authenticated Call Stack. Usenix Security (2021)

Skip to theory

Intel IBT vs. ARMv8.3-A BTI

	Intel IBT	ARMv8.3-A BTI
Indirect branch protection	✓ (one class)	✓ (two classes)
Enforcement model	Deterministic	Deterministic
Memory Overhead	N/A	N/A
Run-time Overhead	? (likely low)	? (likely low)

A theory of run-time attacks

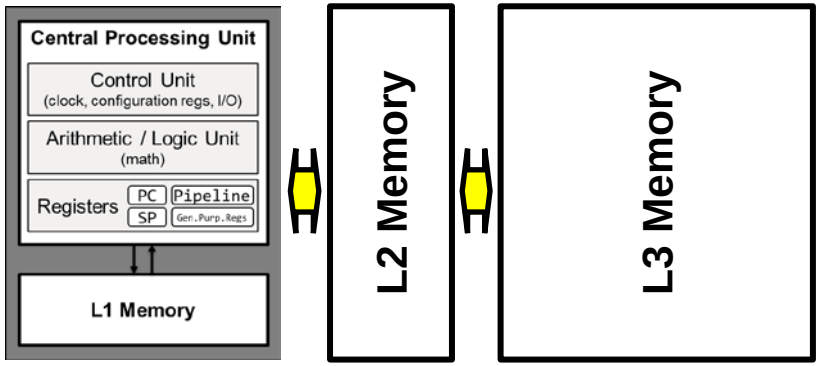
von Neumann architecture

Architecture for a stored-program computer

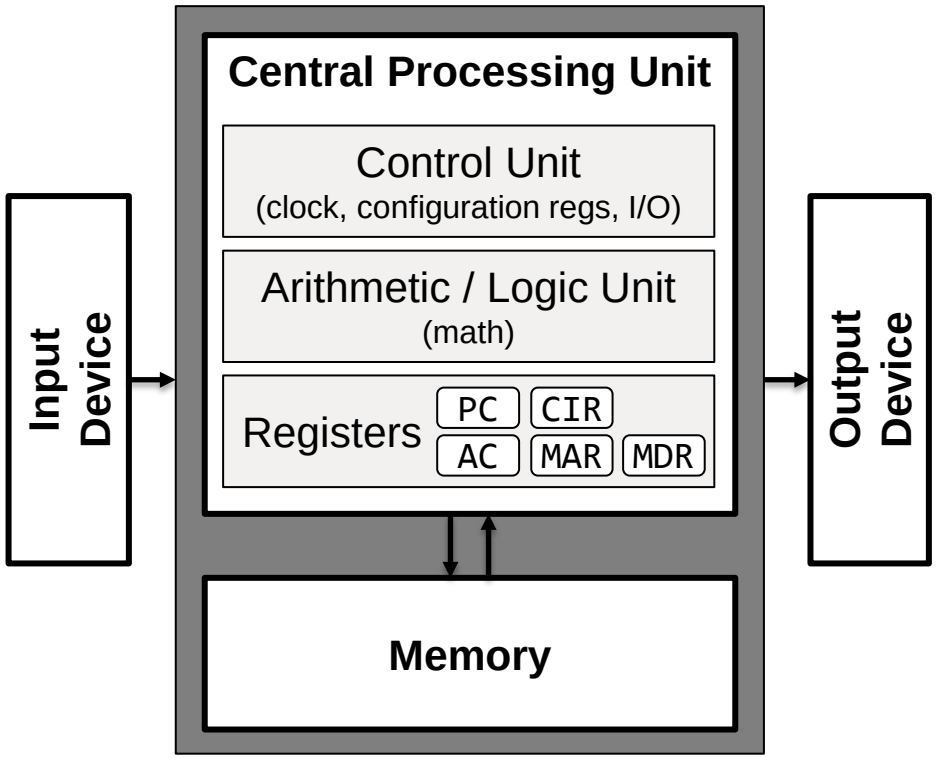
- Realizes (theoretical) concept of universal Turing machine
- Instructions and data stored in memory
- Operates by changing internal state,
  i.e., instructions read and modify some data.

Computer (circa 2020)

With a large addressable memory, different memory types (e.g. SRAM, DRAM flash etc.) and I/O map onto single memory space



Computer (circa 1945)

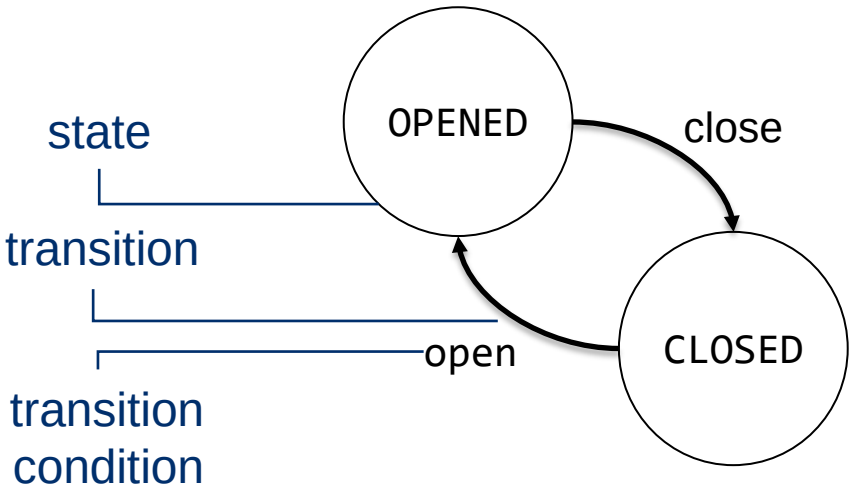


Programs as intended finite state machines

Design of program p can be modeled as (potentially very large) finite state machine[†,‡]

- The **intended finite state machine (IFSM)** describes the intended function of p
- To execute the IFSM on real-world computers, p is realized as a software emulator for the IFSM

$$\theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma)^{\S}$$



The IFSM represents a bug-free version of p

p is a (potentially faulty) emulator for the IFSM

p runs on a processor cpu

†) or a finite state transducer if output is possible

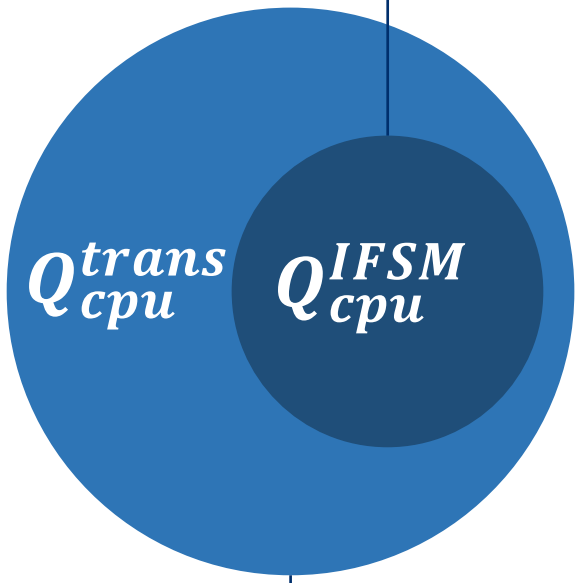
§) Q = set of states, i = initial state
F = final state, Σ, Δ = input and output alphabets
state transition function $\delta: Q \times \Sigma \rightarrow Q$,
output function $\sigma: Q \times \Sigma \rightarrow \Delta$

‡) non-equivalence of FSM/FST to a Turing machine does not matter as any real-world computing device has finite memory

cpu states

$$Q_{cpu}^{p} = Q_{cpu}^{IFSM} \cup Q_{cpu}^{trans}$$

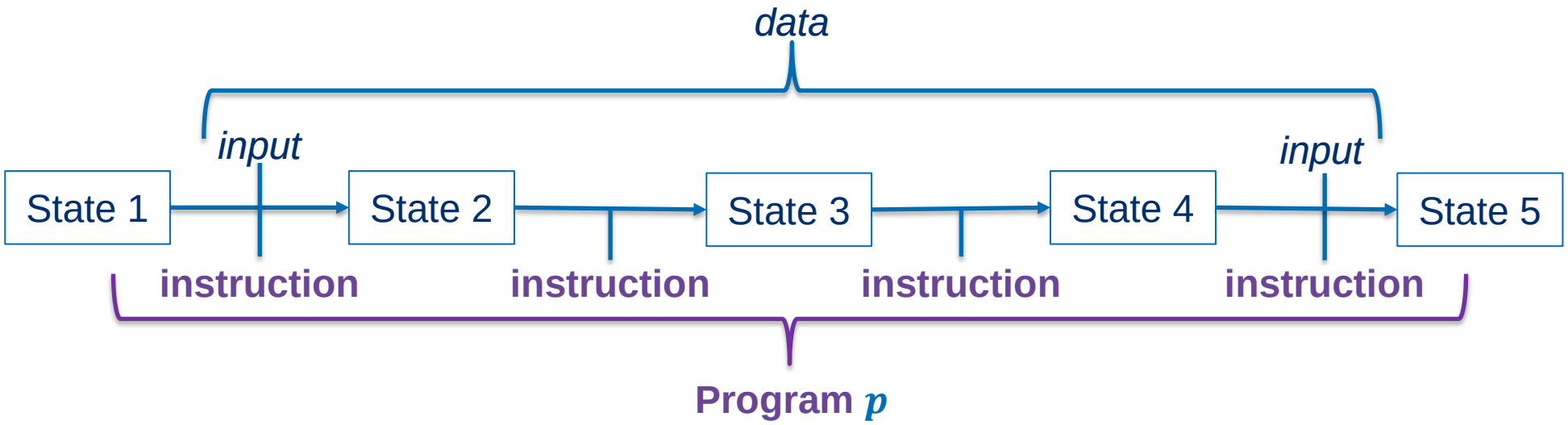
Q_{cpu}^{IFSM}: concrete states of target machine that map to a state in the IFSM

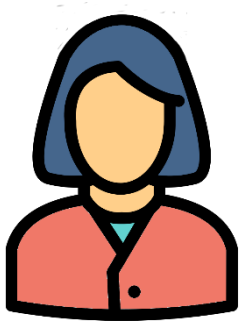


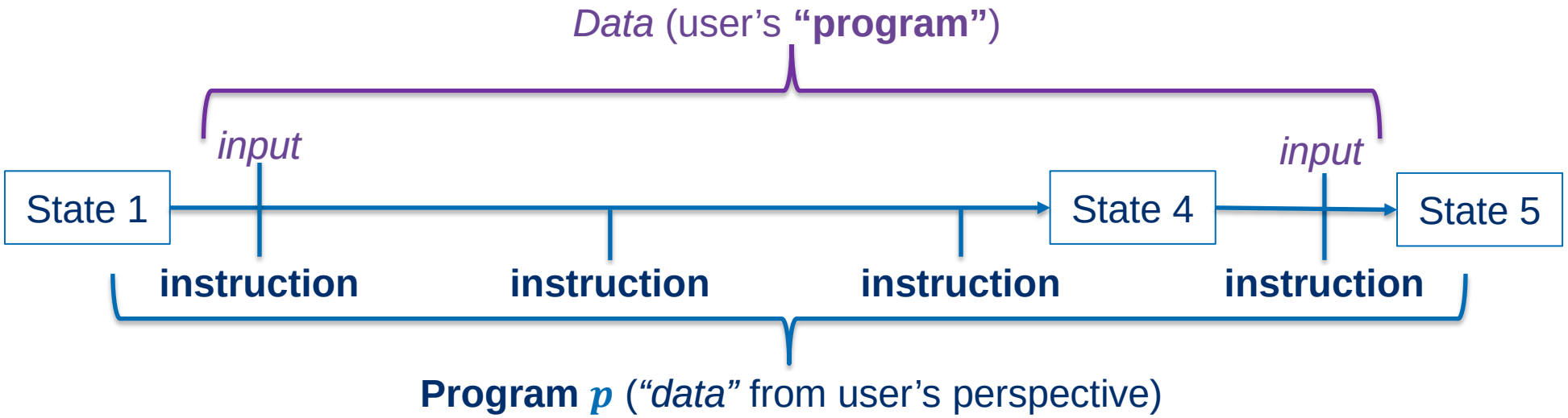
Q_{cpu}^{trans} : benign transitory states that occur during emulation of an edge in the IFSM; part of intended transitions

T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

Two different perspectives of θ

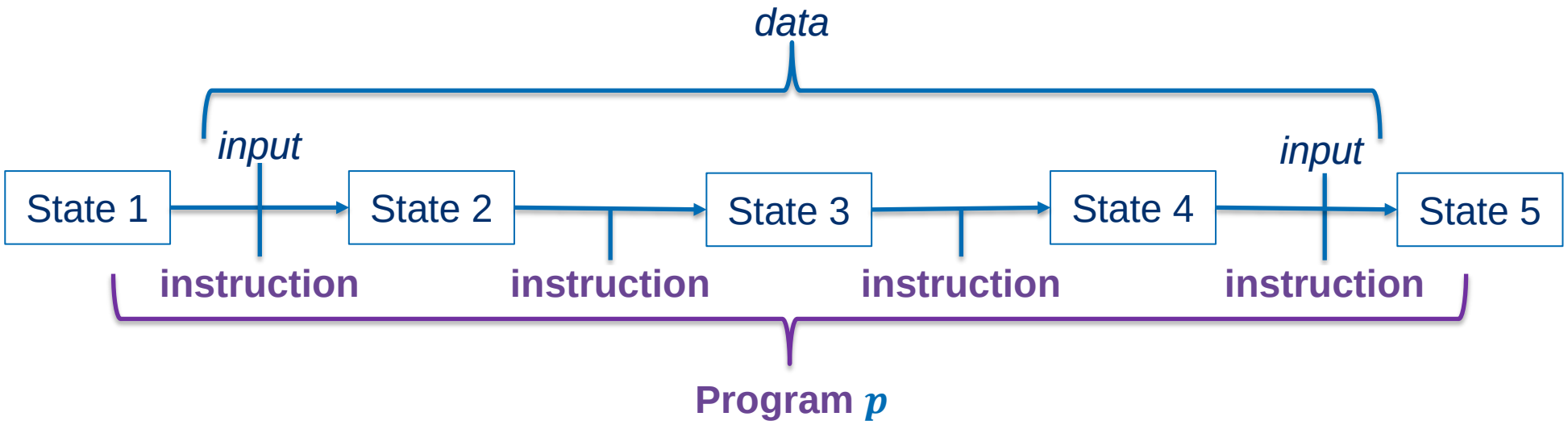


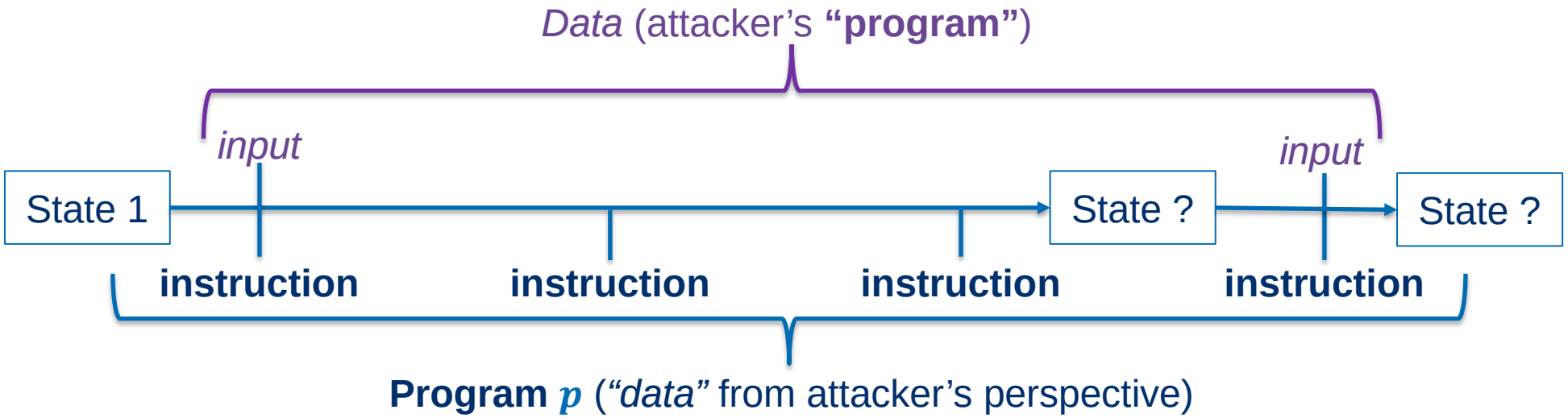




T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

Two different perspectives of θ



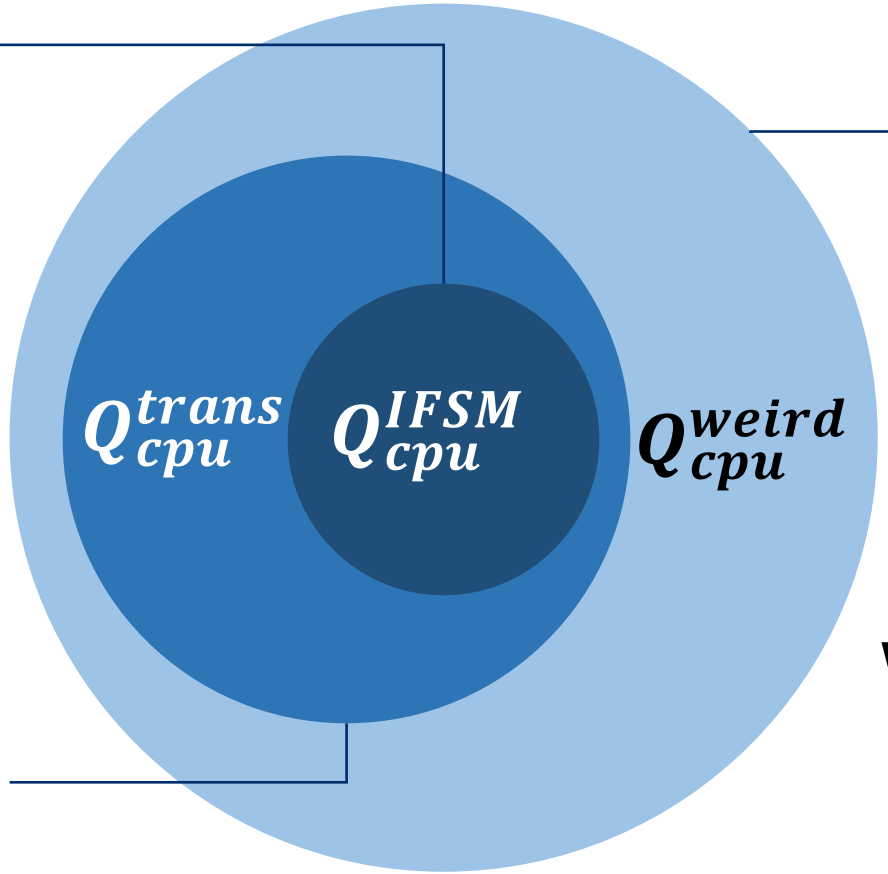


T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

What is a “weird state”?

$$Q_{cpu} = Q_{cpu}^{IFSM} \cup Q_{cpu}^{trans} \cup Q_{cpu}^{weird}$$

Q_{cpu}^{IFSM}: concrete states of target machine that map to a state in the IFSM



Q_{cpu}^{weird}: set of stats in Q_{cpu} not in Q_{cpu}^{IFSM} nor Q_{cpu}^{trans}

Q_{cpu}^{trans} : benign transitory states that occur during emulation of an edge in the IFSM; part of intended transitions

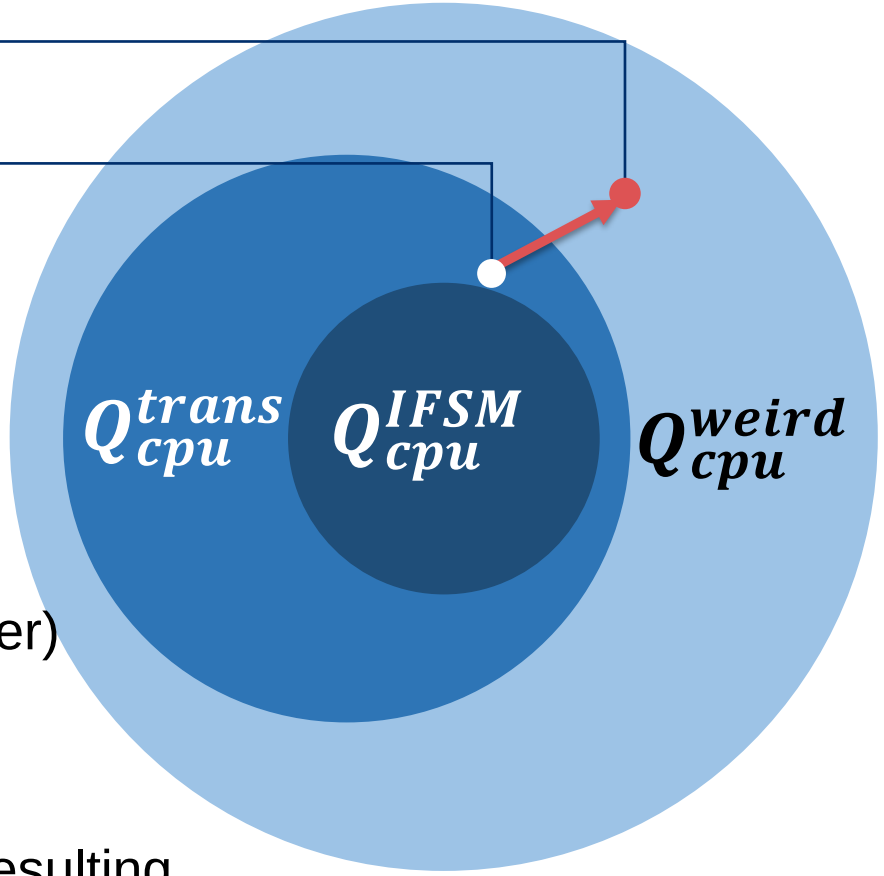
Weird states arise unintentionally and have no meaningful interpretation in the IFSM

T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

Reaching a weird state

$q_{init} \in Q_{cpu}^{weird}$

$q_i \in Q_{cpu}^{IFSM} \cup Q_{cpu}^{trans}$



Intuitively: a bug has occurred when cpu enters a weird state

Vulnerability

- method of moving p to a weird state (accessible to attacker)

Exploitation; run-time attack

- process of choosing q_i, entering q_{init} and programming resulting "weird machine" in order to violate security properties of the IFSM

T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

Weird machines

Recall: $\boldsymbol{\theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma)}$

$\boldsymbol{Q}$ = set of states, $\boldsymbol{i}$ = initial state
$\boldsymbol{F}$ = final state, $\boldsymbol{\Sigma}$, $\boldsymbol{\Delta}$ = input and output alphabets
state transition function $\boldsymbol{\delta}: \boldsymbol{Q} \times \boldsymbol{\Sigma} \rightarrow \boldsymbol{Q}$, output function $\boldsymbol{\sigma}: \boldsymbol{Q} \times \boldsymbol{\Sigma} \rightarrow \boldsymbol{\Delta}$

A weird machine is a computational device where IFSM transitions operate on weird states

$$\boldsymbol{\theta_{weird} = \left(Q_{cpu}^{weird}, q_{init}, Q_{cpu}^{IFSM} \cup Q_{cpu}^{trans}, \Sigma', \Delta', \delta', \sigma'\right)}$$

Instruction stream depends on input

- weird machine programmed through carefully crafted input to p once q_{init} has been entered

Emergent instruction set

- attacker (programmer of the weird machine) must discover the (often unwieldly) semantics of instructions

Unknown state space

- depends heavily on p and q_{init}

Unknown computational power

- greater complexity of the IFSM may yield greater number of instructions, but whether or not the instructions are usable is difficult to predict

T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

Possible sources of weird states

Human error when program p is developed

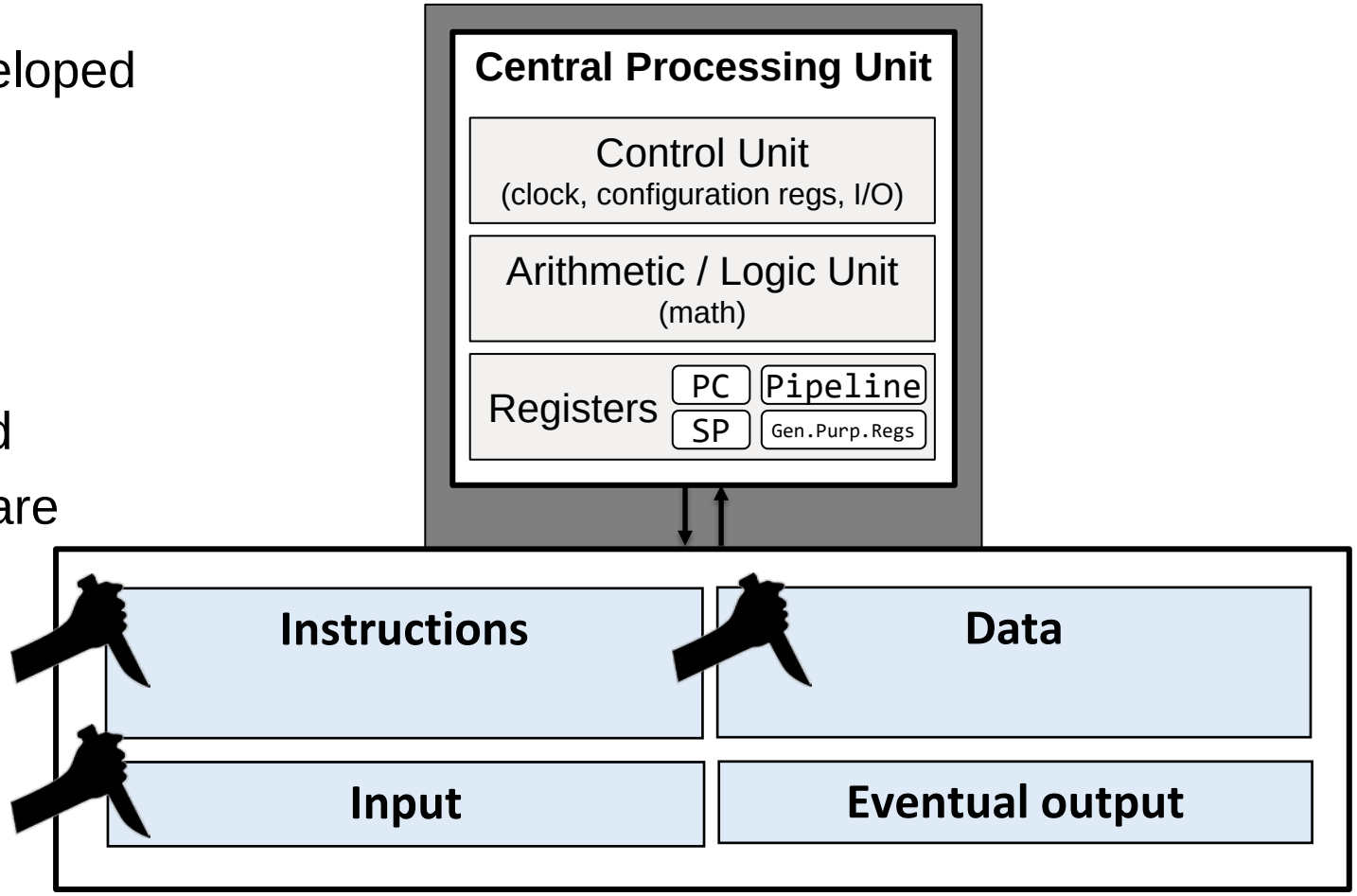
- Memory-related errors, e.g.,
  - spatial errors (buffer overflows)
  - temporal errors (use-after-free)
- Logic errors, e.g., integer overflow

Hardware faults when p is executed

- Probabilistically deterministic hardware
- Fault injection, e.g., Rowhammer

Transcription errors when p is transmitted over error-prone medium

- Hardware failure, e.g., hard drive



T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)
Y. Kim et al., Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors, ISCA (2014)

Modelling the attacker

Arbitrary program-point, chosen-bitflip

can stop p anywhere, flip any one bit in memory, and continue

Arbitrary program-point, chosen-bitflip, registers

same as above, but cannot modify/access registers

Fixed program-point, chosen-bitflip, registers

Fixed program-point, sequential memory rewriting, registers

classical buffer overflow

…

Arbitrary program-point, arbitrary memory-rewriting, registers

most powerful adversary

T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

Defining security

Depends on the desired security goal of θ and p *:* e.g., not disclose sensitive information *s*

Attacker defines $\theta_{exploit}$ to (adapatively) interact with θ_{weird}

Attacker wins if *s* is in the output of θ_{weird} with a higher probability than random

T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

Takeaways

New hardware-assisted defenses are emerging and are (going to be) widely available

How to utilize available primitives effectively?

- **Towards pointer integrity with PA (Usenix SEC ’19)**

How to deal with downsides?

e.g. optimally minimize scope for PA reuse attacks?

- **For return addresses: PACStack (Usenix SEC ‘21)**
- **For other types of pointers?**

https://ssg.aalto.fi/research/projects/harp/

How do different hardware primitives compare?

We have open postdoc and graduate student positions. Talk to me!

Acknowledgments

Icons on slides 3, 4, 5, 15, 16, 17, 19, 21, 22, 23 and 24 made by Good Ware from www.flaticon.com licensed by CC 3.0 BY

The PHP logo on slide 25 made by Colin Viebrock licensed by CC BY-SA 4.0

The BSD daemon on slide 25 is copyright of Marshall Kirk McKusick

Slide 11 (Return-oriented programming (high-level idea) is by Luca Davi.