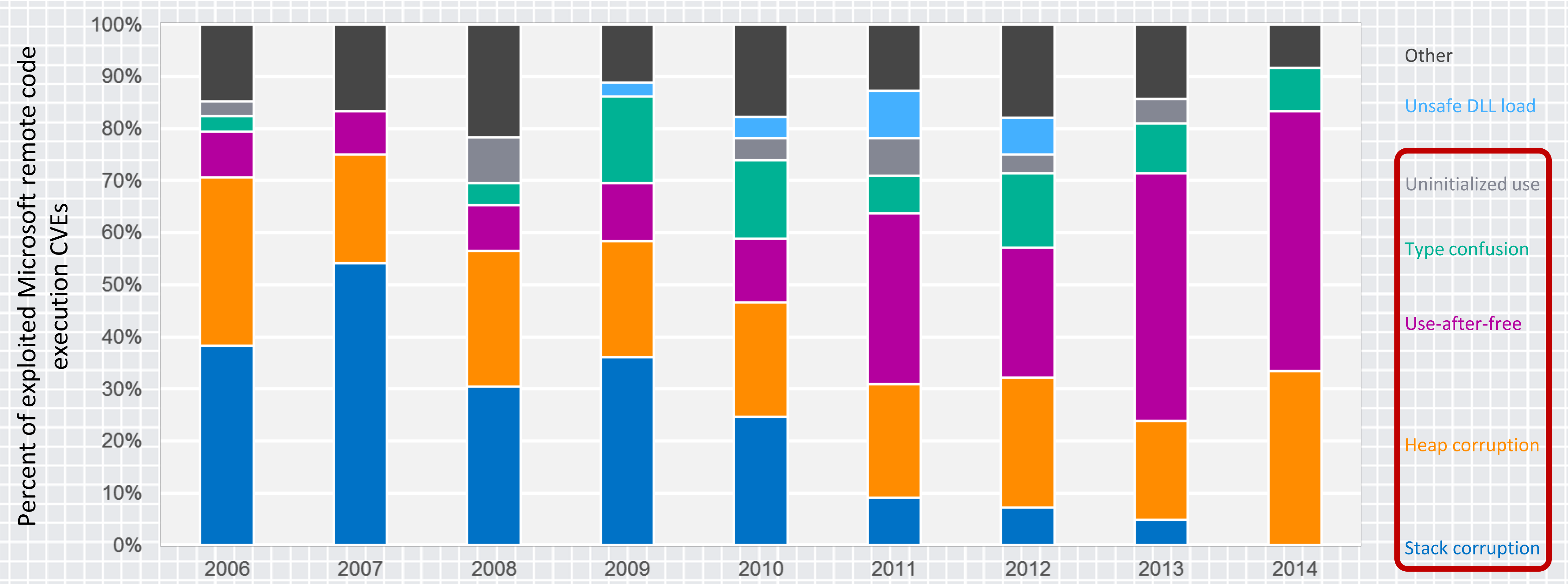
HDFI: Hardware-Assisted Data-flow Isolation

Chengyu Song[1], Hyungon Moon[2], Monjur Alam[1], Insu Yun[1], Byoungyoung Lee[1], Taesoo Kim[1], Wenke Lee[1], Yunheung Paek[2]

[1]**Georgia Institute of Technology**

[2]Seoul National University

Memory corruption vulnerability

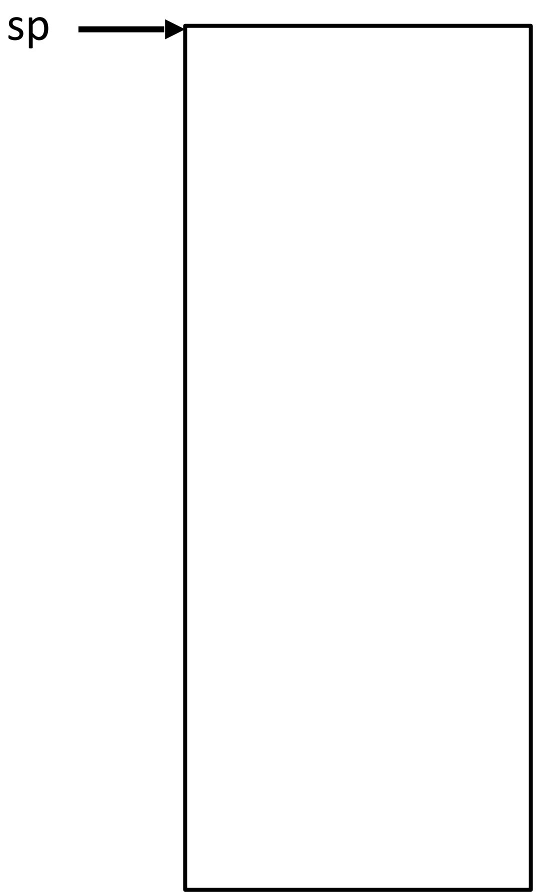


Exploitation Trends: From Potential Risk to Actual Risk, RSA 2015

A simple stack overflow

```c
int main(int argc, const char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

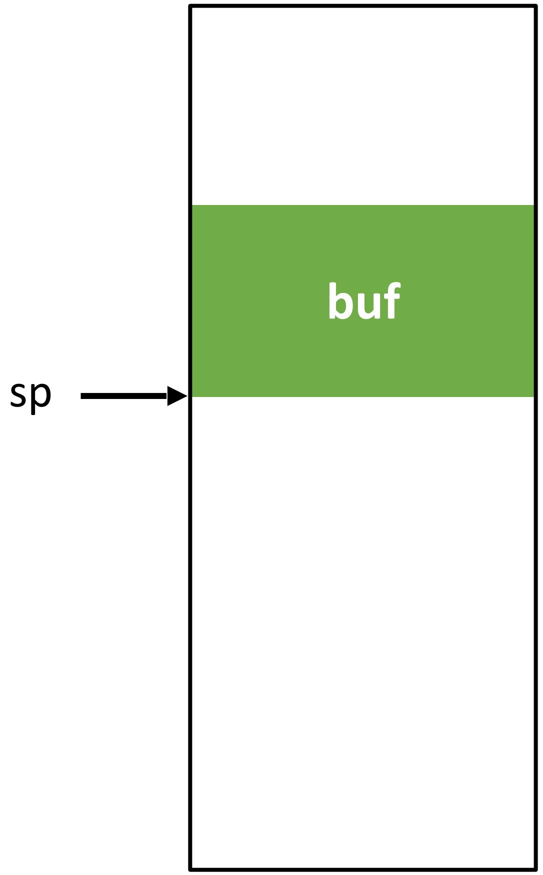
```asm
main:
  add     sp,sp,-32
  sd      ra,24(sp)
  ld      a1,8(a1)        ; argv[1]
  mv      a0,sp           ; char buff[16]
  call    strcpy          ; strcpy(buff, argv[1])
  li      a0,0
  ld      ra,24(sp)
  add     sp,sp,32
  jr      ra              ; return
```



A simple stack overflow

```c
int main(int argc, const char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

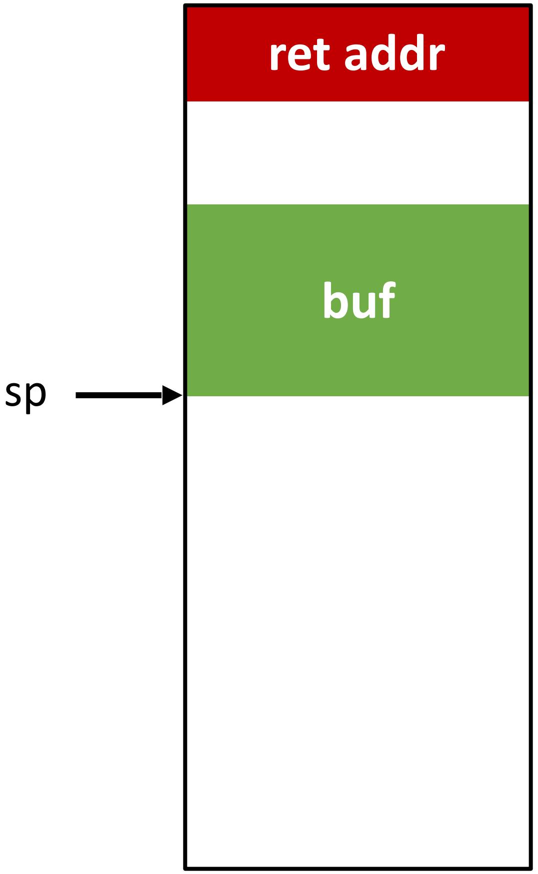
```asm
main:
    add     sp,sp,-32
    sd      ra,24(sp)
    ld      a1,8(a1)        ; argv[1]
    mv      a0,sp           ; char buff[16]
    call    strcpy          ; strcpy(buff, argv[1])
    li      a0,0
    ld      ra,24(sp)
    add     sp,sp,32
    jr      ra              ; return
```



A simple stack overflow

```c
int main(int argc, const char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

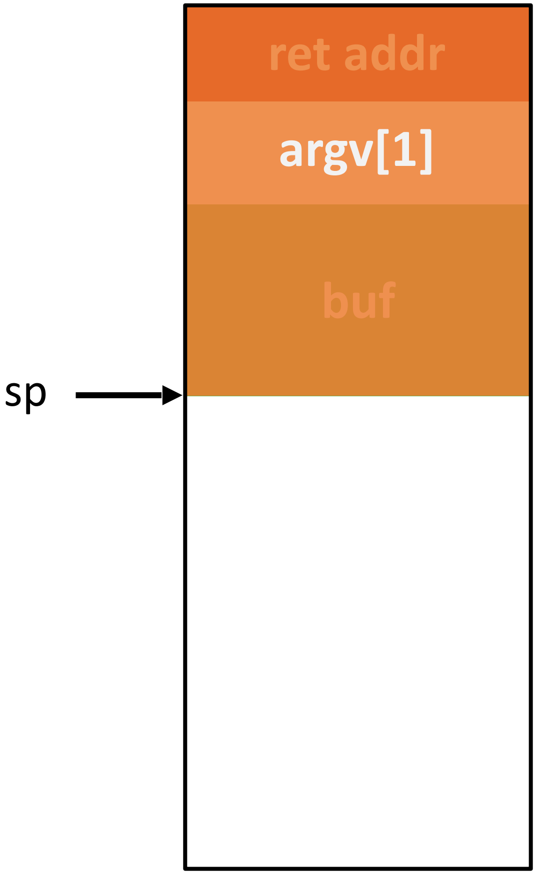
```
main:
  add     sp,sp,-32
  sd      ra,24(sp)
  ld      a1,8(a1)       ; argv[1]
  mv      a0,sp          ; char buff[16]
  call    strcpy         ; strcpy(buff, argv[1])
  li      a0,0
  ld      ra,24(sp)
  add     sp,sp,32
  jr      ra             ; return
```



A simple stack overflow

```c
int main(int argc, const char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

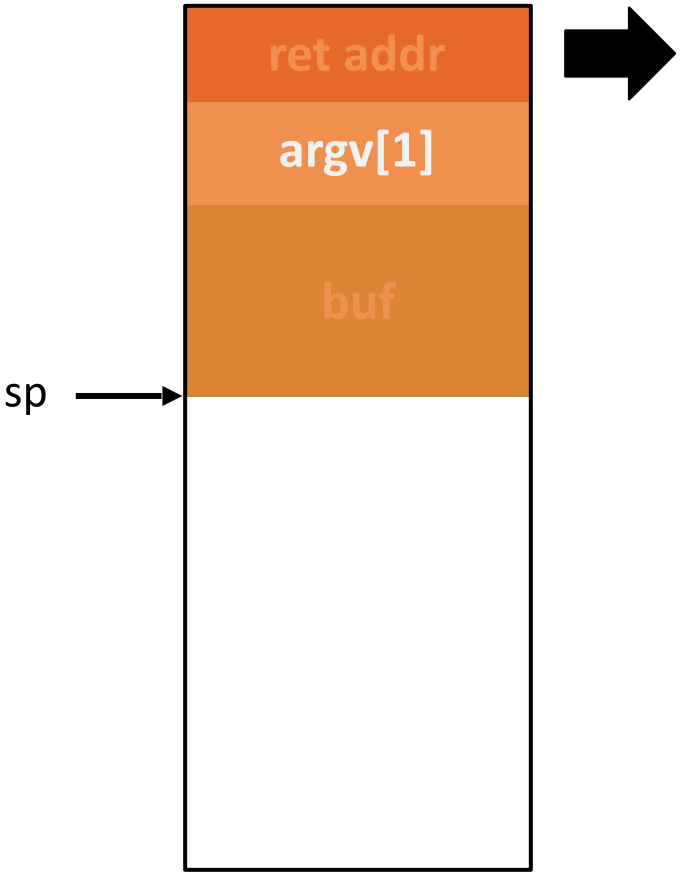
```
main:
  add    sp,sp,-32
  sd     ra,24(sp)
  ld     a1,8(a1)         ; argv[1]
  mv     a0,sp            ; char buff[16]
  call   strcpy           ; strcpy(buff, argv[1])
  li     a0,0
  ld     ra,24(sp)
  add    sp,sp,32
  jr     ra               ; return
```



A simple stack overflow

```c
int main(int argc, const char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

```asm
main:
  add     sp,sp,-32
  sd      ra,24(sp)
  ld      a1,8(a1)        ; argv[1]
  mv      a0,sp           ; char buff[16]
  call    strcpy          ; strcpy(buff, argv[1])
  li      a0,0
  ld      ra,24(sp)
  add     sp,sp,32
  jr      ra              ; return
```



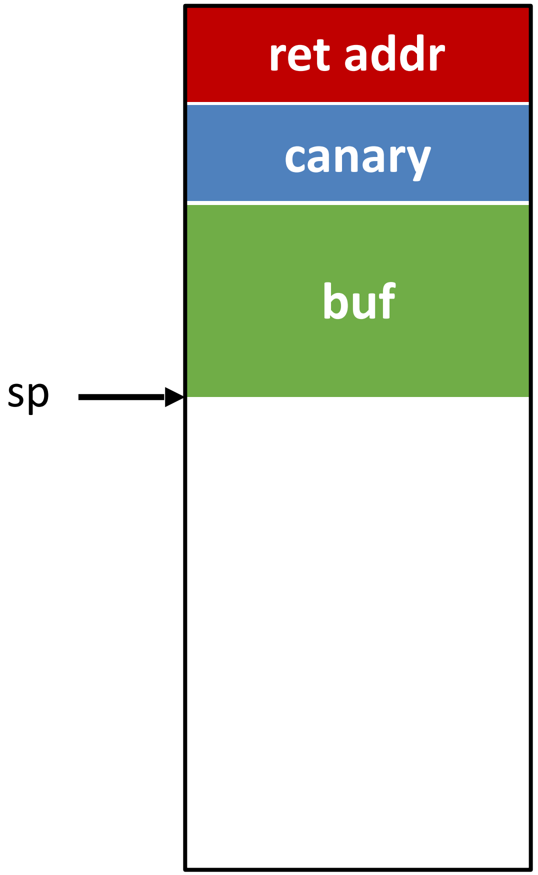
Code Injection

ROP

Defense mechanisms

```c
int main(int argc, const char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

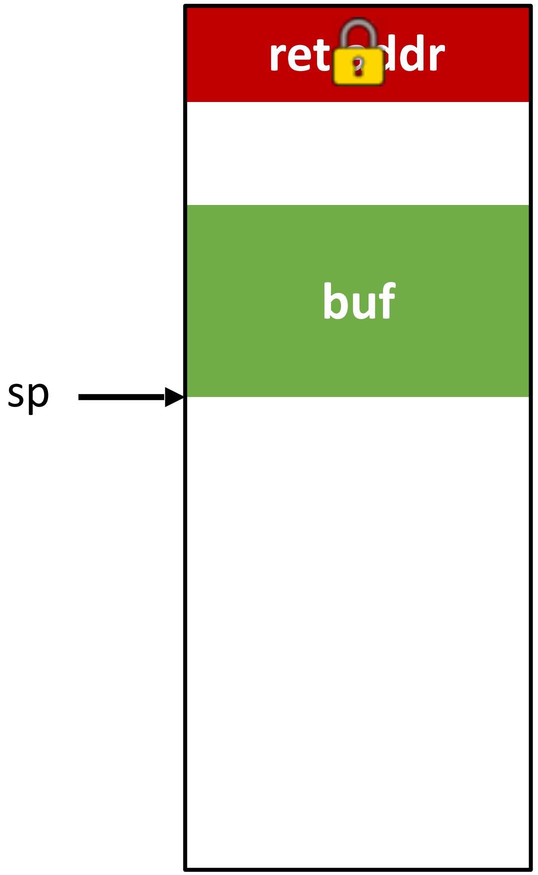
```asm
main:
  add     sp,sp,-32
  sd      ra,24(sp)
  ld      a1,8(a1)          ; argv[1]
  mv      a0,sp             ; char buff[16]
  call    strcpy            ; strcpy(buff, argv[1])
  li      a0,0
  ld      ra,24(sp)
  add     sp,sp,32
  jr      ra                ; return
```



Defense mechanisms

```c
int main(int argc, const char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

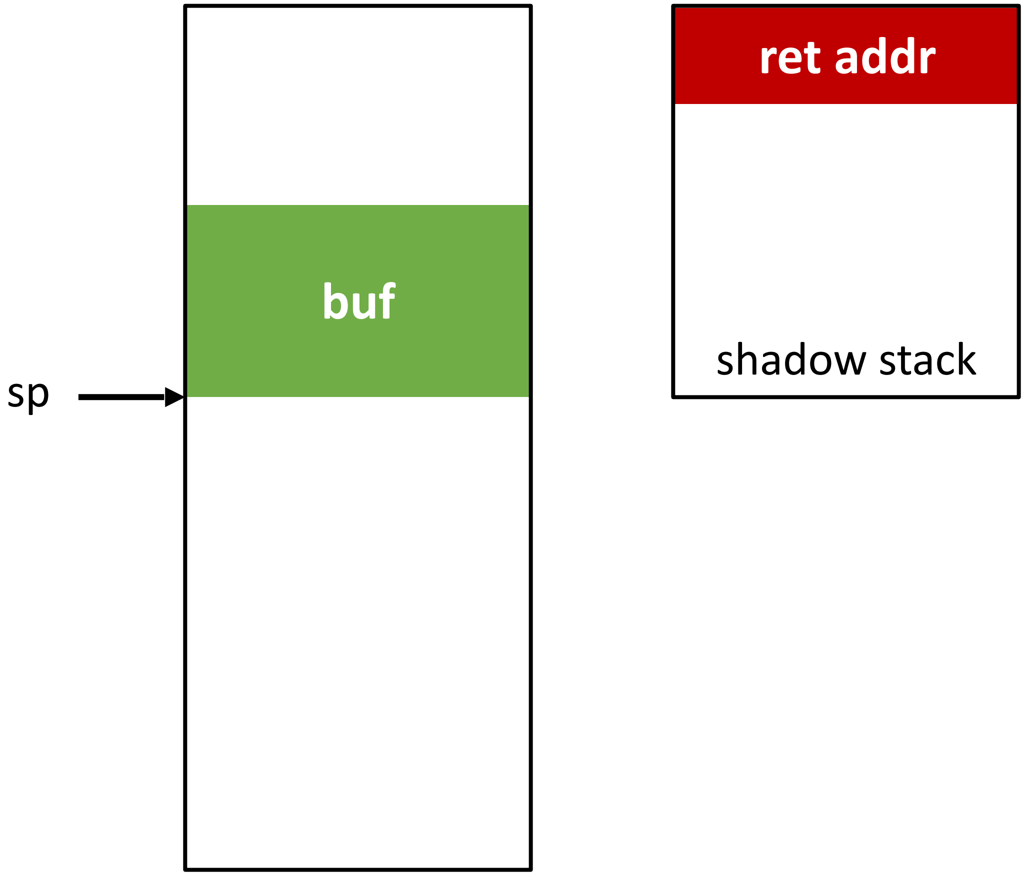
```asm
main:
    add     sp,sp,-32
    sd      ra,24(sp)
    ld      a1,8(a1)        ; argv[1]
    mv      a0,sp           ; char buff[16]
    call    strcpy          ; strcpy(buff, argv[1])
    li      a0,0
    ld      ra,24(sp)
    add     sp,sp,32
    jr      ra              ; return
```



Defense mechanisms

```c
int main(int argc, const char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

```asm
main:
  add     sp,sp,-32
  sd      ra,24(sp)
  ld      a1,8(a1)        ; argv[1]
  mv      a0,sp           ; char buff[16]
  call    strcpy          ; strcpy(buff, argv[1])
  li      a0,0
  ld      ra,24(sp)
  add     sp,sp,32
  jr      ra              ; return
```



Limitations

- Software: lacks good isolation mechanisms in 64-bit world
  - SFI and virtual address space: secure but expensive
  - Address randomization: efficient but insecure
- Hardware: lacks flexibility
  - Context saving/restoring (setjmp/longjmp), deep recursion, kernel stack, etc.
  - Other data: code pointers, non-control data
- Data shadowing: adds overheads
  - Breaks data locality, needs additional step to look up or reserved register(s)
  - Occupies additional memory

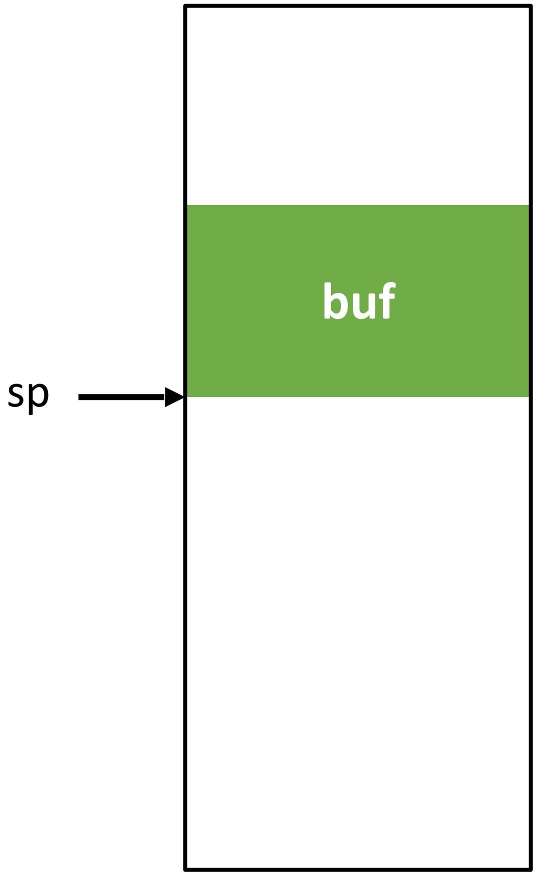
Hardware-assisted data-flow isolation

- Secure and efficient
  - Low performance overhead and strong security guarantees
- Flexible
  - Capable of supporting different security model/mechanisms
- Fine-grained
  - No more data-shadowing
- Practical
  - Minimized hardware changes

Data-flow Integrity [OSDI’06]

Runtime data-flow should not deviate from static data-flow graph

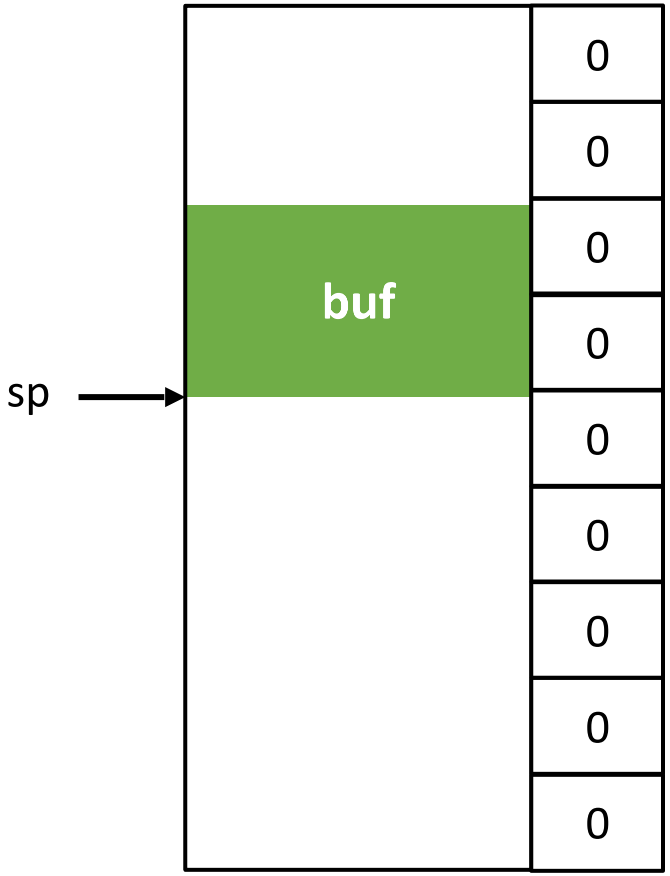
```
main:
  add     sp,sp,-32
  sd      ra,24(sp)
  ld      a1,8(a1)          ; argv[1]
  mv      a0,sp             ; char buff[16]
  call    strcpy            ; strcpy(buff, argv[1])
  li      a0,0
  ld      ra,24(sp)
  add     sp,sp,32
  jr      ra                ; return
```



Data-flow Integrity [OSDI'06]

Runtime data-flow should not deviate from static data-flow graph

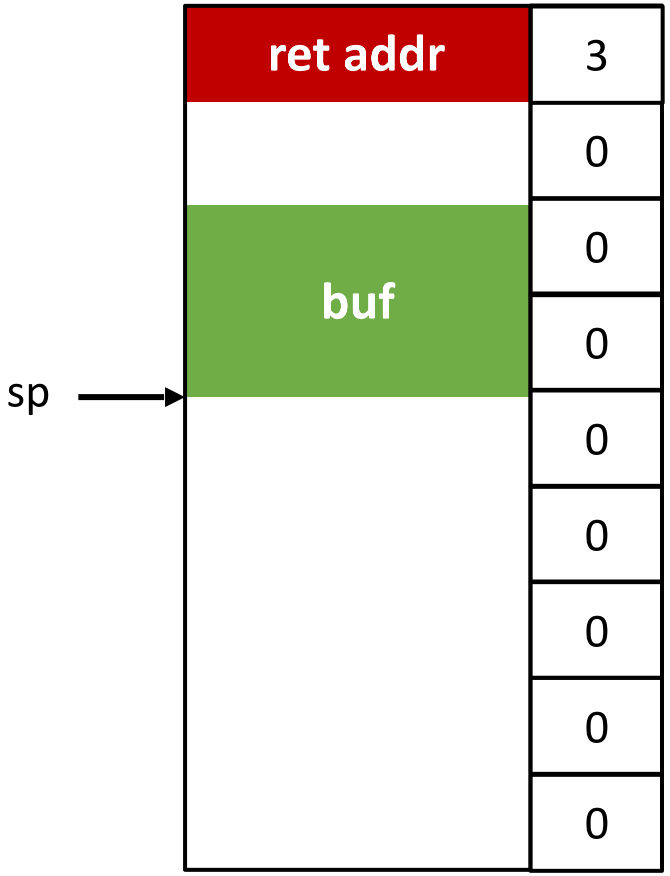
```
main:
  add     sp,sp,-32
  sd      ra,24(sp)
  ld      a1,8(a1)          ; argv[1]
  mv      a0,sp             ; char buff[16]
  call    strcpy            ; strcpy(buff, argv[1])
  li      a0,0
  ld      ra,24(sp)
  add     sp,sp,32
  jr      ra                ; return
```



Data-flow Integrity [OSDI'06]

Runtime data-flow should not deviate from static data-flow graph

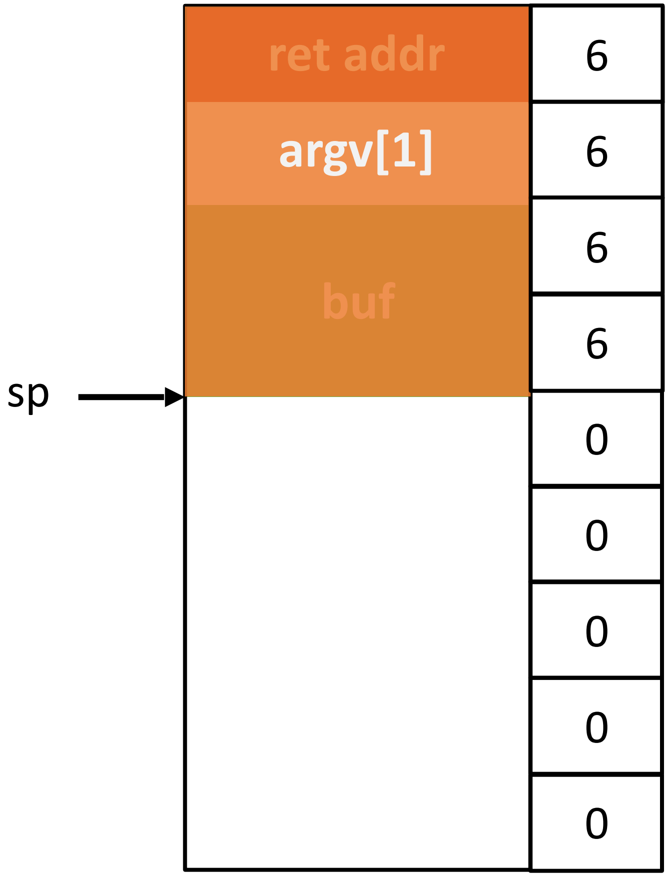
```
main:
  add     sp,sp,-32
  sd      ra,24(sp)
  ld      a1,8(a1)        ; argv[1]
  mv      a0,sp           ; char buff[16]
  call    strcpy          ; strcpy(buff, argv[1])
  li      a0,0
  ld      ra,24(sp)
  add     sp,sp,32
  jr      ra              ; return
```



Data-flow Integrity [OSDI'06]

Runtime data-flow should not deviate from static data-flow graph

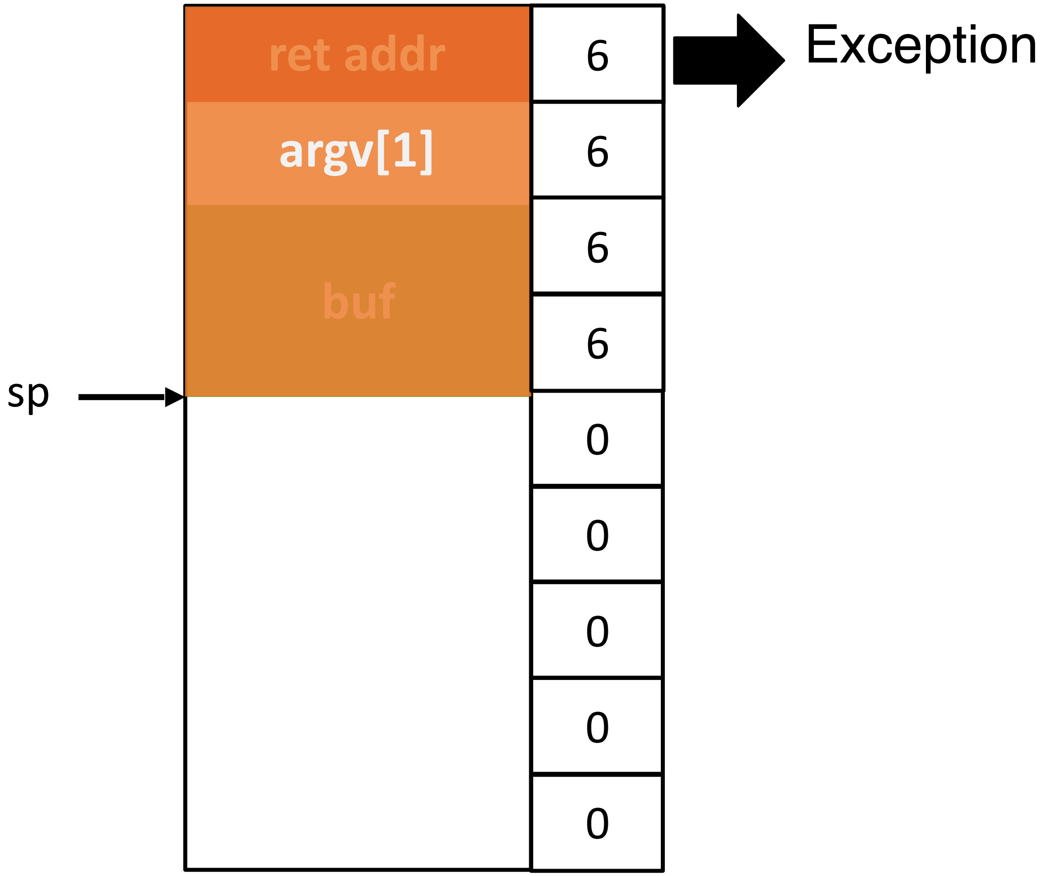
```
main:
  add    sp,sp,-32
  sd     ra,24(sp)
  ld     a1,8(a1)        ; argv[1]
  mv     a0,sp           ; char buff[16]
  call   strcpy          ; strcpy(buff, argv[1])
  li     a0,0
  ld     ra,24(sp)
  add    sp,sp,32
  jr     ra              ; return
```



Data-flow Integrity [OSDI'06]

Runtime data-flow should not deviate from static data-flow graph

```
main:
  add     sp,sp,-32
  sd      ra,24(sp)
  ld      a1,8(a1)        ; argv[1]
  mv      a0,sp           ; char buff[16]
  call    strcpy          ; strcpy(buff, argv[1])
  li      a0,0
  ld      ra,24(sp)
  add     sp,sp,32
  jr      ra              ; return
```

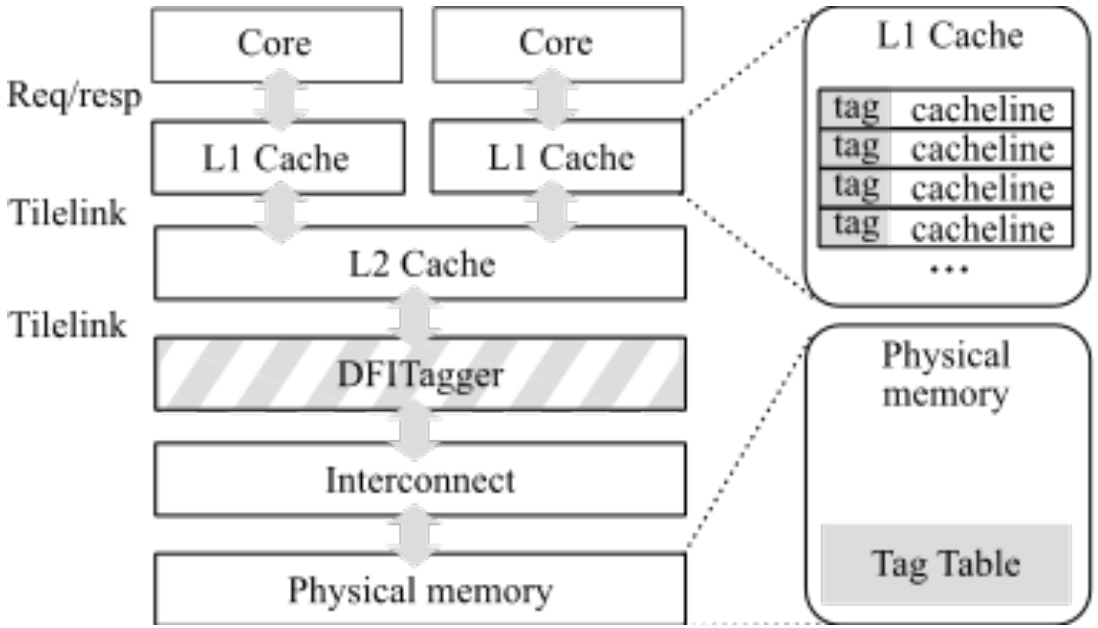


ISA extension

- Tagged memory
  - Machine word granularity
  - Fixed tag size → currently only *1 bit* (sensitive or not)
- Three new *atomic* instructions to enable DFI-style checks
  - **`sdset1, ldchk0, ldchk1`**
- New semantic of old instructions (backward compatible)
  - **`sd`** : **`sdset0`**
  - **`ld`** : now tag check

Hardware extension

- Cache extension
  - Extra bits in the cache line for storing the tag (reusing existing cache coherence interconnect)
- Memory Tagger
  - Emulating tagged memory without physically extending the main memory



Optimizations

- Memory Tagger introduces additional performance overhead
  - Naive implementation: 2x memory accesses, 1 for data, 1 for tag
- Three optimization techniques
  - Tag cache
  - Tag valid bits (TVB)
  - Meta tag table (MTT)

Return address protection

- Policy: return address should always have tag 1
- Benefits: secure and supports context saving/restoring, deep recursion, modified return address, kernel stack

```
main:
  add      sp,sp,-32
 ⋆sdset1   ra,24(sp)
  ld       a1,8(a1)        ; argv[1]
  mv       a0,sp           ; char buff[16]
  call     strcpy          ; strcpy(buff, argv[1])
  li       a0,0
 ⋆ldchk1   ra,24(sp)
  add      sp,sp,32
  jr       ra              ; return
```

Various applications

	Application	Security Policy (invariants)
Integrity Protection	Shadow Stack	return address and register spills should has tag 1 (push / pop)
	`vptr` Protection	`vptr` should has tag 1 (constructor / virtual function call)
	Code Pointer Separation	code pointer should has tag 1 (CPI [OSDI'14])
	C Library Enhancement	important data/pointer should has tag 1 (manual modification)
	Kernel Protection	sensitive kernel data should has tag 1 (Kenali [NDSS'16])
Leak Detection	Heartbleed Prevention	crypto keys should has tag 1
		output buffer should has tag 0

Implementations

- Hardware
  - RISC-V RocketCore generator: 2198 LoC
  - Instantiated on Xilinx Zynq ZC706 FPGA board
- Software (RISC-V toolchain)
  - Assembler gas: 16 LoC
  - Kernel modifications: 60 LoC
  - Security applications: 170 LoC

Effectiveness of optimizations

- Memory bandwidth and latency

Benchmark	Tag Cache	+TVB	+MTT	+TVB+MTT
L1 hit	0%	0%	0%	0%
L1 miss	14.47%	5.26%	14.47%	5.26%
Copy	13.14%	4.44%	11.84%	4.26%
Scale	10.62%	4.79%	9.45%	4.67%
Add	4.37%	1.26%	4.13%	1.2%
Triad	9.66%	1.96%	8.8%	1.83%

- SPEC CINT2000

Benchmark	Tag Cache	+TVB	+MTT	+TVB+MTT
164.gzip	16.09%	2.18%	6.85%	1.87%
175.vpr	29.51%	3.26%	7.71%	1.43%
181.mcf	36.89%	3.08%	13.66%	-0.11%
197.parser	16.11%	2.27%	7.61%	1.53%
254.gap	12.19%	1.04%	6.53%	0.71%
256.bzip2	14.52%	2.65%	3.63%	0.84%
300.twolf	26.71%	2.97%	7.37%	0.36%

Security experiments

- With synthesized attacks

Mechanism	Attacks	Result
Shadow stack	RIPE	✓
Heap metadata protection	Heap exploit	✓
VTable protection	VTable hijacking	✓
Code pointer separation (CPS)	RIPE	✓
Code pointer separation (CPS)	Format string exploit	✓
Kernel protection	Privilege escalation	✓
Private key leak prevention	Heartbleed	✓

Impacts on security solutions

- Security
  - Hardware-enforced isolation
- Simplicity
- No data shadowing
- Usability
- Implementation/port is very easy

Application	Language	LoC
Shadow Stack	C++ (LLVM 3.3)	4
VTable Protection	C++ (LLVM 3.3)	40
CPS	C++ (LLVM 3.3)	41
Kernel Protection	C (Linux 3.14.41)	70
Library Protection	C (glibc 2.22)	10
Heartbleed Prevention	C (OpenSSL 1.0.1a)	2

Impacts on security solutions (cont.)

- Efficiency
  - GCC (-O2)
  - Clang (-O0)

Benchmark	Shadow stack (GCC)	SS+CPS (Clang)
164.gzip	1.12%	2.42%
181.mcf	1.76%	3.54%
254.gap	3.34%	13.23%
256.bzip2	3.05%	4.61%

Security analysis

- Attack surface
  - Inaccuracy of data-flow analysis
  - Deputy attacks
- Best practice
  - CFI is necessary (e.g., CPS + shadow stack)
  - Recursive protection of pointers
  - Guarantee the trustworthiness of the written value
  - Use runtime memory safety technique to compensate inaccuracy of static analysis

Q & A

Thank you!