# Bypassing memory safety mechanisms through speculative control flow hijacks

Andrea Mambretti

(mbr@ccs.neu.edu)

Alexandra Sandulescu, Alessandro Sorniotti and Anil Kurmus - IBM Research Europe, Zurich
William Robertson and Engin Kirda - Northeastern University

Northeastern University

IBM Research

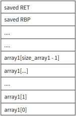
# Buffer overflow since 1996 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
  array1[i] = val;
}
return;
```

[1] http://phrack.org/issues/49/14.html

# Buffer overflow since 1996 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
   array1[i] = val;
}
return;
```



saved RET

saved RBP

....

....

array1[size_array1 - 1]

array1[...]

....

array1[1]

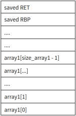
array1[0]

[1] http://phrack.org/issues/49/14.html

# Buffer overflow since 1996 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
   array1[i] = val;
}
return;
```

with **len** > **size_array1**



saved RET

saved RBP

....

....

array1[size_array1 - 1]

array1[...]

....

array1[1]

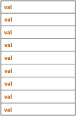
array1[0]

[1] http://phrack.org/issues/49/14.html

# Buffer overflow since 1996 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
  array1[i] = val;
}
return;
```

with **len** > **size_array1**

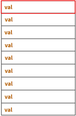


[1] http://phrack.org/issues/49/14.html

# Buffer overflow since 1996 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
  array1[i] = val;
}
return;
```

with **len** > **size_array1**



val
val
val
val
val
val
val
val
val

[1] http://phrack.org/issues/49/14.html

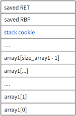
# Stack Smashing Protector (SSP) since 1998 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
   array1[i] = val;
}
check_stack_ovf();
return;
```

[1] https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

# Stack Smashing Protector (SSP) since 1998 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
  array1[i] = val;
}
check_stack_ovf();
return;
```



saved RET

saved RBP

stack cookie

....

array1[size_array1 - 1]

array1[...]

....

array1[1]

array1[0]

[1] https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

# Stack Smashing Protector (SSP) since 1998 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
  array1[i] = val;
}
check_stack_ovf();
return;
```

with **len** > **size_array1**

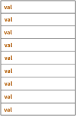
| |
|---|
| saved RET |
| saved RBP |
| stack cookie |
| .... |
| array1[size_array1 - 1] |
| array1[...] |
| .... |
| array1[1] |
| array1[0] |

[1] https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

# Stack Smashing Protector (SSP) since 1998 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
  array1[i] = val;
}
check_stack_ovf();
return;
```

with **len** > **size_array1**



**val**

**val**

**val**

**val**

**val**

**val**

**val**

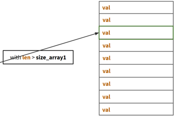
**val**

**val**

[1] https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

# Stack Smashing Protector (SSP) since 1998 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
  array1[i] = val;
}
check_stack_ovf();
return;
```

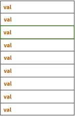


[1] https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

# Stack Smashing Protector (SSP) since 1998 [1]

```
int array1[size_array1];
...
for (int i=0; i < len; ++i) {
   array1[i] = val;
}
check_stack_ovf();
return;
```

with **len** > **size_array1**



**Fast forward 2018 - speculative execution attacks**

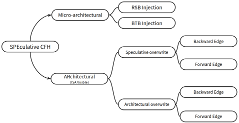
[1] https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

# Research Question

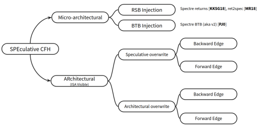
**Are current** memory corruption **mitigations** still **valid**

**in** the context of **speculative execution attacks**?

# SPEculative ARchitectural control flow hijacks (SPEAR)



# SPEculative ARchitectural control flow hijacks (SPEAR)



SPEculative CFH

Micro-architectural

RSB Injection

Spectre returns [**KKSG18**], ret2spec [**MR18**]

BTB Injection

Spectre BTB (aka v2) [**PJ0**]

ARchitectural
(ISA Visible)

Speculative overwrite

Backward Edge

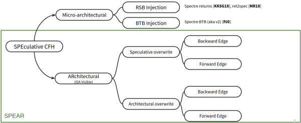
Forward Edge

Architectural overwrite

Backward Edge

Forward Edge

# SPEculative ARchitectural control flow hijacks (SPEAR)



SPEculative CFH

Micro-architectural

RSB Injection

Spectre returns [**KKSG18**], ret2spec [**MR18**]

BTB Injection

Spectre BTB (aka v2) [**PJ0**]

ARchitectural (ISA Visible)

Speculative overwrite

Backward Edge

Forward Edge

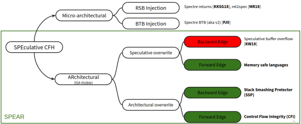
Architectural overwrite

Backward Edge

Forward Edge

SPEAR

# SPEculative ARchitectural control flow hijacks (SPEAR)



SPEculative CFH

Micro-architectural

RSB Injection

Spectre returns [**KKSG18**], ret2spec [**MR18**]

BTB Injection

Spectre BTB (aka v2) [**PJ0**]

ARchitectural
(ISA Visible)

Speculative overwrite

Backward Edge

Speculative buffer overflow [**KW18**]

Forward Edge

**Memory safe languages**

Architectural overwrite

Backward Edge

**Stack Smashing Protector (SSP)**

Forward Edge

**Control Flow Integrity (CFI)**

SPEAR

# Does SSP fully mitigate buffer overflows?

```
func:
    mov rbx, QWORD[fs:0x28]
    mov QWORD[stack_cookie], rbx
    .... /* buffer overflow */ ...
    mov rbx, QWORD[stack_cookie]
    xor QWORD[fs:0x28], rbx
    je exit
    call __stack_chk_fail
exit:
    ret
```

# Does SSP fully mitigate buffer overflows?

```
func:
    mov rbx, QWORD[fs:0x28]
    mov QWORD[stack_cookie], rbx
    .... /* buffer overflow */ ...
    mov rbx, QWORD[stack_cookie]
    xor QWORD[fs:0x28], rbx
    je exit
    call __stack_chk_fail
exit:
    ret
```

# Does SSP fully mitigate buffer overflows?

```
func:
    mov rbx, QWORD[fs:0x28]
    mov QWORD[stack_cookie], rbx
    .... /* buffer overflow */ ...
    mov rbx, QWORD[stack_cookie]
    xor QWORD[fs:0x28], rbx
    je exit
    call __stack_chk_fail
exit:
    ret
```

# Does SSP fully mitigate buffer overflows?

```
func:
    mov rbx, QWORD[fs:0x28]
    mov QWORD[stack_cookie], rbx
    .... /* buffer overflow */ ...
    mov rbx, QWORD[stack_cookie]
    xor QWORD[fs:0x28], rbx
    je exit
    call __stack_chk_fail
exit:
    ret
```

# Does SSP fully mitigate buffer overflows?

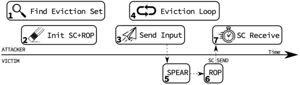
```
func:
    mov rbx, QWORD[fs:0x28]
    mov QWORD[stack_cookie], rbx
    .... /* buffer overflow */ ...
    mov rbx, QWORD[stack_cookie]
    xor QWORD[fs:0x28], rbx
    je exit
    call __stack_chk_fail
exit:
    ret
```

# Does SSP fully mitigate buffer overflows?

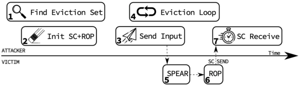
```
func:
    mov rbx, QWORD[fs:0x28]
    mov QWORD[stack_cookie], rbx
    .... /* buffer overflow */ ...
    mov rbx, QWORD[stack_cookie]
    xor QWORD[fs:0x28], rbx
    je exit
    call __stack_chk_fail
exit:
    ret
```

**We demonstrate SSP can be bypassed with a SPEAR**

# Bypassing SSP with SPEAR: CVE-2004-0597

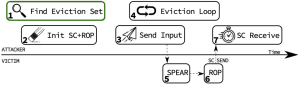


# Bypassing SSP with SPEAR: CVE-2004-0597



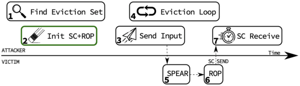
**Assumption:** SSP prevents traditional exploitation

# Bypassing SSP with SPEAR: CVE-2004-0597



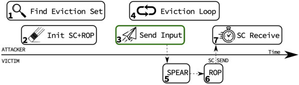
**Assumption:** SSP prevents traditional exploitation

# Bypassing SSP with SPEAR: CVE-2004-0597



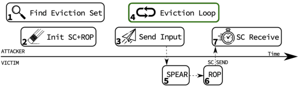
**Assumption:** SSP prevents traditional exploitation

# Bypassing SSP with SPEAR: CVE-2004-0597



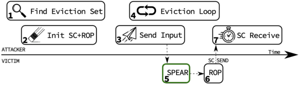
**Assumption:** SSP prevents traditional exploitation

# Bypassing SSP with SPEAR: CVE-2004-0597



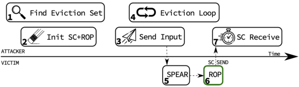
**Assumption:** SSP prevents traditional exploitation

# Bypassing SSP with SPEAR: CVE-2004-0597



**Assumption:** SSP prevents traditional exploitation

# Bypassing SSP with SPEAR: CVE-2004-0597



1 Find Eviction Set

2 Init SC+ROP

3 Send Input

4 Eviction Loop

7 SC Receive

ATTACKER

Time

VICTIM

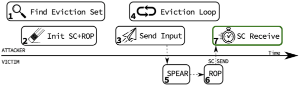
5 SPEAR

6 ROP

SC SEND

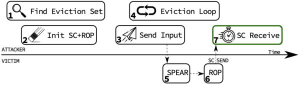
**Assumption:** SSP prevents traditional exploitation

# Bypassing SSP with SPEAR: CVE-2004-0597



**Assumption:** SSP prevents traditional exploitation

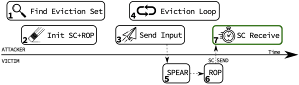
# Bypassing SSP with SPEAR: CVE-2004-0597



**Assumption:** SSP prevents traditional exploitation

**Impact**: arbitrary read from victim program

# Bypassing SSP with SPEAR: CVE-2004-0597



**Assumption:** SSP prevents traditional exploitation

Leakage 0.3 bytes x second

**Impact**: arbitrary read from victim program

# Other use cases

# Other use cases

Memory safe languages are also affected (Golang, Rust)

# Other use cases

Memory safe languages are also affected (Golang, Rust)

**=> our work motivated the introduction of -*spectre* flag in Go v1.15**

# Other use cases

Memory safe languages are also affected (Golang, Rust)

**=> our work motivated the introduction of -*spectre* flag in Go v1.15**

Control flow integrity (LLVM-CFI, GCC-VTV)

# Other use cases

Memory safe languages are also affected (Golang, Rust)

**=> our work motivated the introduction of *-spectre* flag in Go v1.15**

Control flow integrity (LLVM-CFI, GCC-VTV)

**=> LLVM-CFI NOT vulnerable due to design**

# Conclusion

# Conclusion

SPEAR attacks bypass mitigations and memory safety to leak confidential data

=> **new and old mitigations must be analyzed and possibly modified to withstand SPEAR attacks**

These attacks are complex but practical

=> **with new tools to aid building each attack stage, they could become more practical**

**Speculative ROP is possible and eases the task of finding a spectre v1-like side channel send gadget**

SEAs are a significant research and industry challenge for the next decade (tools, attacks and defences)