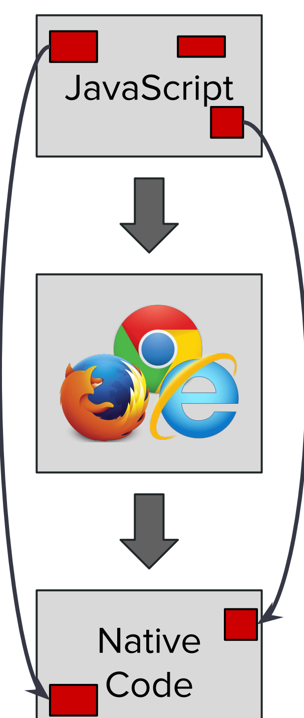
What Cannot be Read, Cannot be Leveraged?

Revisiting Assumptions of JIT-ROP Defenses

Giorgi Maisuradze, Michael Backes, Christian Rossow
CISPA, Saarland University, Germany

Overview

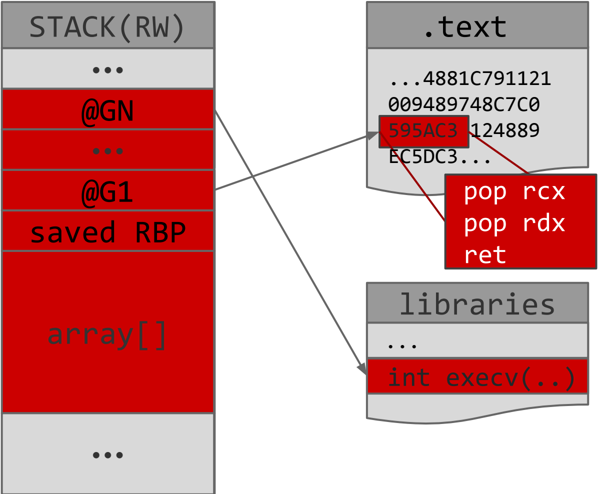
- Code Reuse Attacks/Defenses
- JavaScript JIT Compilation
- Existing JIT Attacks/Defenses
- *New* JIT Gadget Creation
- *New* JIT Defense



Revisiting Code Reuse

Code Reuse

- Find useful code fragments
- Chain gadgets and functions together
- Trigger execution of the gadget chain

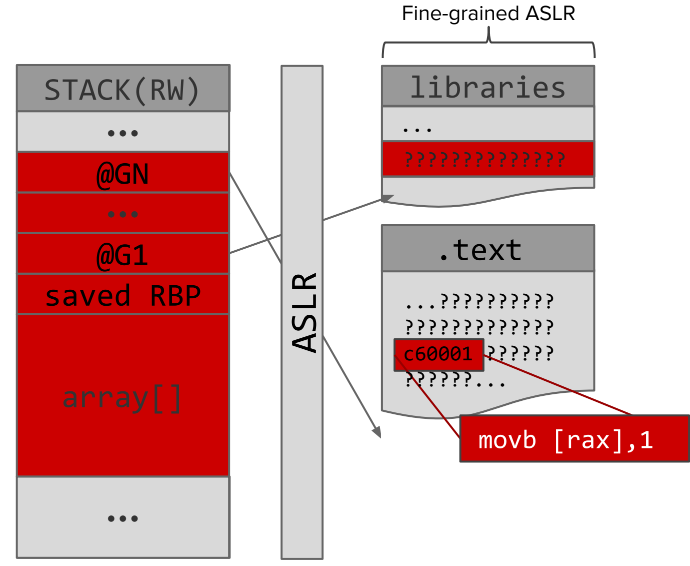


Code Reuse Defenses

- Find useful code fragments
- Chain gadgets and functions together
- Trigger execution of the gadget chain

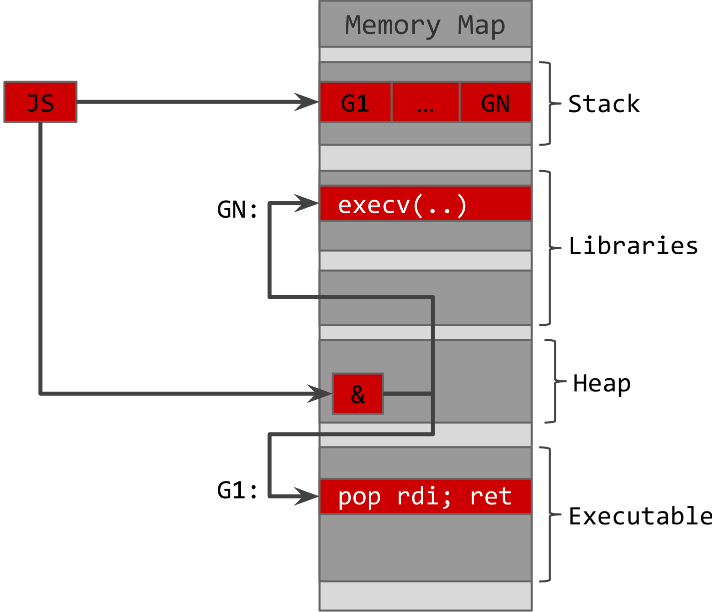
Defenses:

- Randomize memory segments (ASLR)
- Fine-grained code randomization



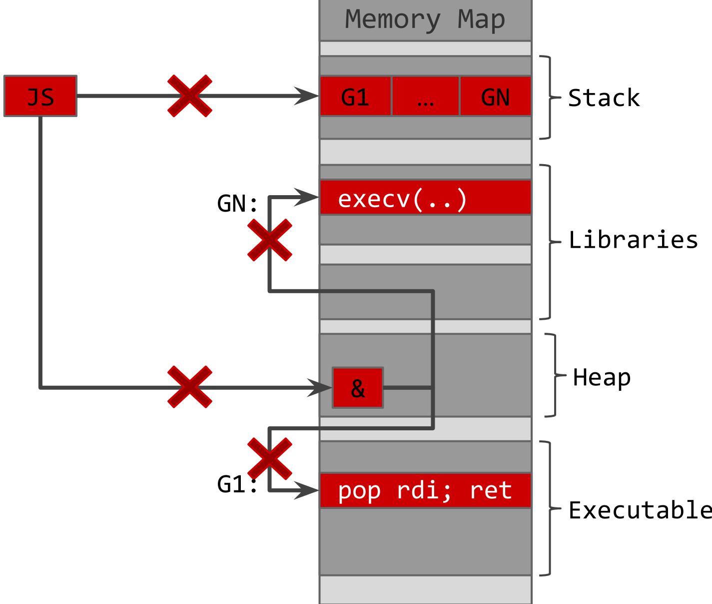
JIT-ROP [S&P'13]

- Leak and collect code pointers
- Read code pages to find gadgets and function pointers
- Chain gadgets and functions together
- Trigger execution of the gadget chain



Execute-no-Read (XnR) Schemes

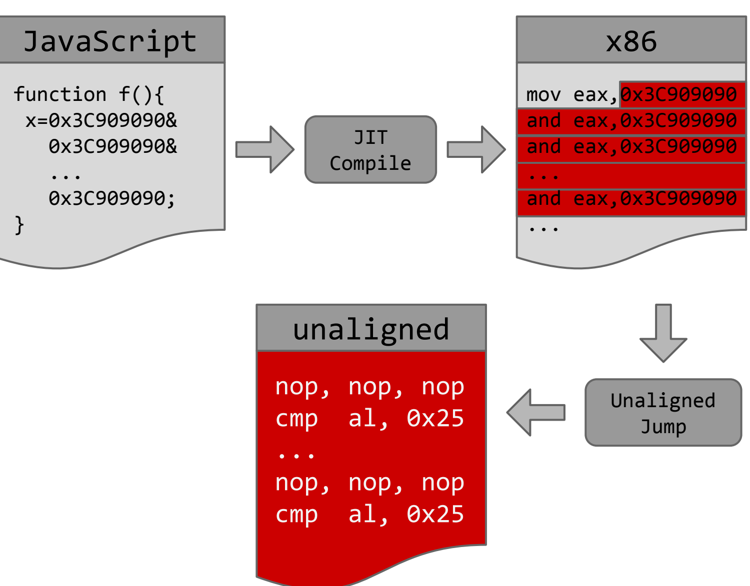
- Mark code pages non-readable
  [CCS’14, S&P’15, CCS’15, CODASPY’15, NDSS’16]
- Randomize code
- Hide code pointers



JIT code reuse without reading code?!

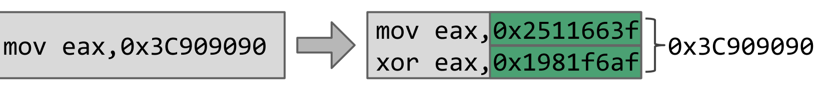
JIT Spraying [WOOT'10]

- Create JS functions containing operations on immediate values
- Trigger JIT-compilation of created functions
- Divert the CF in the middle of one of the constants

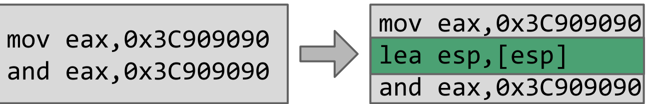


Defenses in Browsers

- Constant Blinding (IE + Chrome)
  - Store constant XORed with a random key to a register
  - XOR the register again with the key



- NOP Insertion (IE)
  - Randomly add random number of NOP instructions



- Only large constants (>2B) are blinded

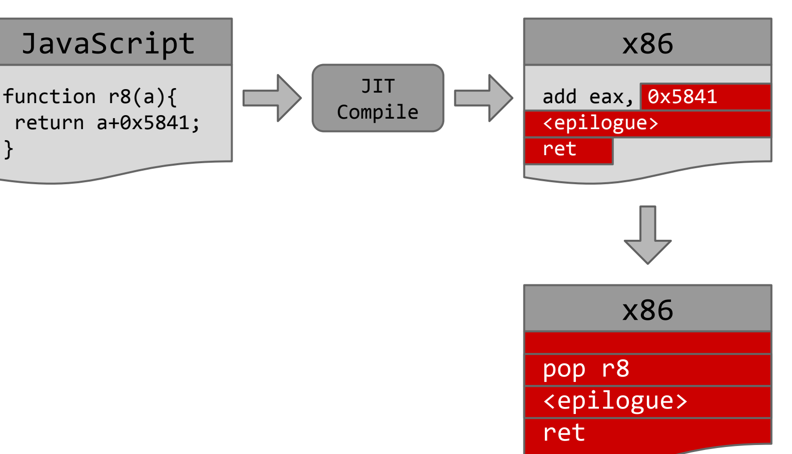
2B Gadgets

The Devil is in the Constants [NDSS'15]

- Encode OPs in 2B unblinded constants
- Use intended return instruction from JIT-compiled code
- Find and use emitted gadgets

Limitations:

- Requires readable code to find gadgets
- Aligned return instruction



Gadgets via Control Flow Instructions

Emitting Gadgets via Control Flow Instructions

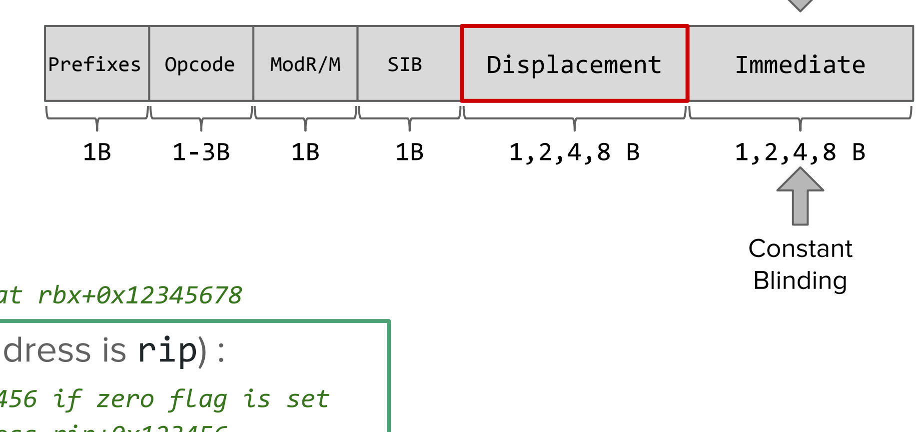
Displacement:

- Encodes the offset from a base address, e.g., from `rax` or `rip`
- Used in:
  - Instructions accessing the memory:

```
mov  rax, [rbx+0x12345678] ; read memory at rbx+0x12345678
```

  - Control flow instructions (implicit base address is `rip`) :

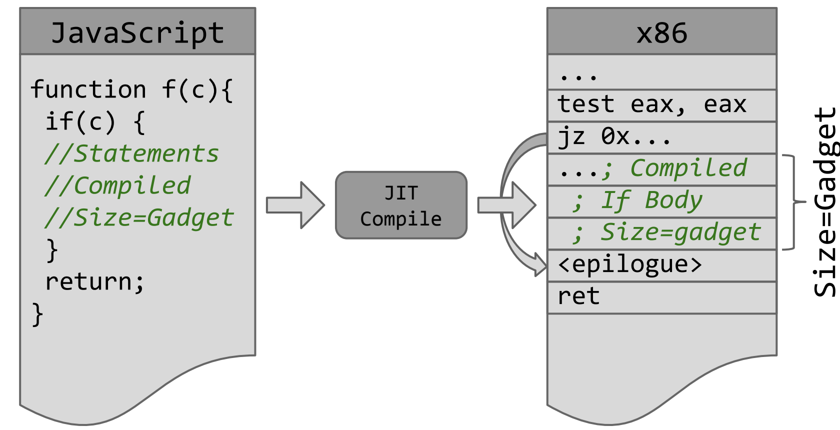
```
jz   0x123456 ; jump at address rip+0x123456 if zero flag is set
call 0x123456 ; call the function at address rip+0x123456
```



We will use JavaScript statements that are compiled to conditional jumps and direct calls to emit gadgets

Conditional Jumps in JS

- JavaScript `if` statement:
  - Emits conditional jumps after compilation
  - Displacement field denotes the size of the compiled `if` body
  - Change the size of `if` body ⇒ changes the value in the displacement field



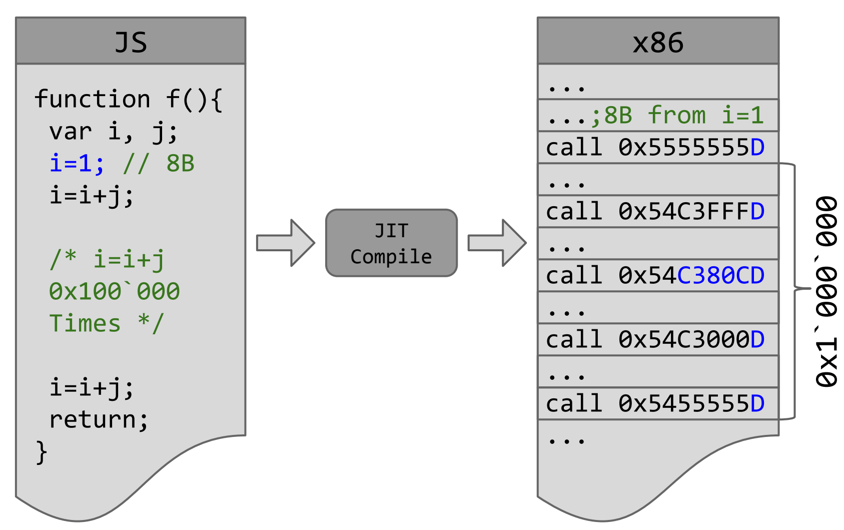
- Alternatives to `if` statement:

  `while/for/switch/break...`

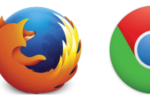
The size of the compiled `if` statement's body determines emitted Gadget

Direct Calls in JS

- Direct calls in JS:
  - Displacement: the distance between the caller and the callee
- Create a large JS function:
  - Use `i=i+j` (16B) 0x100`000 times
  - Emitted values [0x000005, 0xFFFFF5]
  - Prepend the function with `i=1` (8B) to change 5 into 13 (0xD)
- Find callee/caller address:
  - Compiler's heap
  - Return address from stack
  - JS objects (e.g., from `Math.random`)



Applying Concepts to Modern Browsers



Gadgets in Chrome 51 (64-bit)

Goal:

pop r8; ret	4158c3
pop r9; ret	4159c3
pop rcx; ret	59c3
pop rdx; ret	5ac3
int 0x80; ret	cd80c3

Direct calls (`syscall, pop rcx, pop rdx`):

- Align LSHB of displacements to 0xd
- Emit direct calls (`j++` 0x80`000 times)

Conditional jumps (`pop r8, pop r9`):

- Stack two if statements (for pop r8 and pop r9)
- Fill the inner (pop r8) with 0xc35841 bytes
- Add additional 0xed bytes to the outer `if` body
  - +0x13 bytes from `if` statement=0x100

Gadget generation time ≈ 1.3 seconds

JS

```
function syscall(){
 var j=0;
 // align displacements'
 // LSHB to 0xd
 j++;j++;j++;j++;
 // j++ 0x80`000 times
}
```

```
function popr8r9(r8,r9){
 var i=0, j=0;
 if(r9){
  // j=0x1010101 7 times
  // => 0xbd
  j++;j=i;j=i;
  // => 0xed
  if(r8){ // => 0x100
   // j=0x1010101 211 times
   // => 0x1641
   // j++ 399`888 times
   // => 0xC35841
  }
 // => 0xC35941
 }
}
```

Limitations in Internet Explorer

Challenges in IE:

1. Limited JIT-compiled function size
   - Maximum function size 1MB
   - No 3B displacements in conditional jumps
   - No huge functions with direct calls
2. NOP insertion
   - Non-predictable offsets inside the function

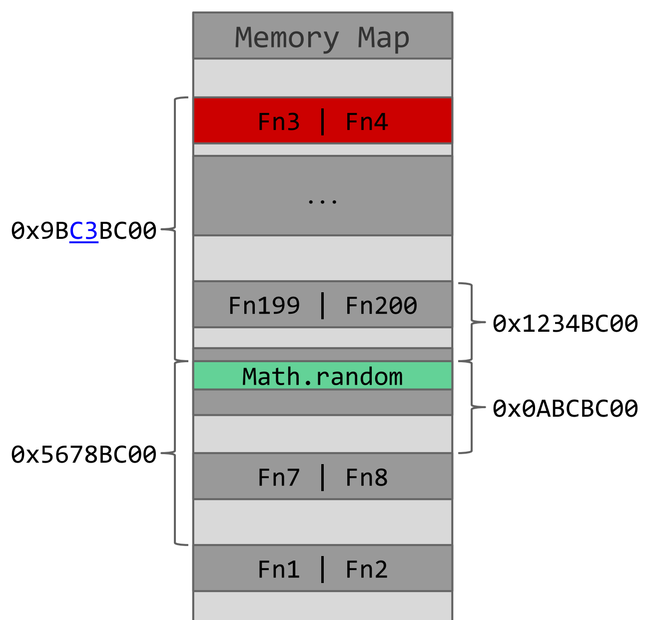
Steps to make direct calls work:

1. Get code page at the correct distance from the callee
2. Fill the page with gadget emitting functions
3. Emit the direct call at the correct address, or recompile

Allocating Code Page at a Correct Distance

IE code-page size is 0x20`000 bytes

- Compile 200 JavaScript functions of size 0x10`000 bytes each to allocate 100 code pages
- Choose the code page at the correct distance



Gadgets in IE ➔ Emitting Gadgets

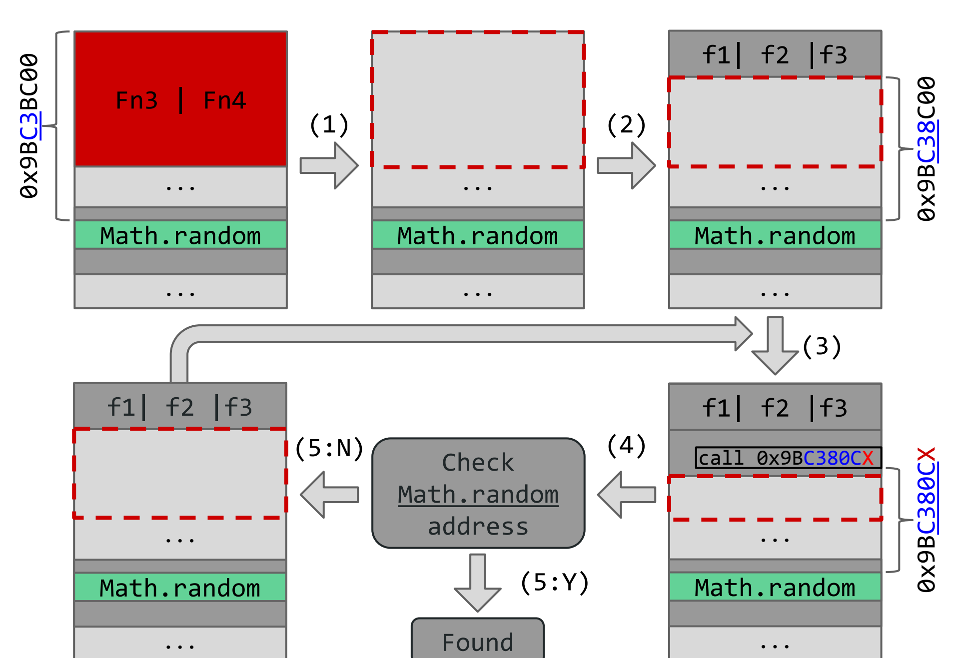
(1) Deallocate the function

(2) Fill the code page up to the correct distance from the callee

(3) Compile gadget-emitting JS function

(4) Check if the direct call was emitted at the correct place

(5) If yes, then gadget is found, else deallocate the function and goto (3)



Gadgets in IE 11 (32-bit)

Goal:

popa; ret	61c3
int 0x80; ret	cd80c3

Direct calls (`syscall, popa`):

- Use `i=Math.random()` 250 times (≈0x1`000 bytes)
- Check the address of emitted direct call

Emitting gadgets:

(1) Find and fill up the correct code page (≈*4 seconds*)
(2) Compile `syscall` and `popa` at correct places (≈*4 seconds*)
(3) Recompile `syscall` until the gadget is emitted (≈*2 seconds*)

Average gadget generation time ≈ 32 seconds

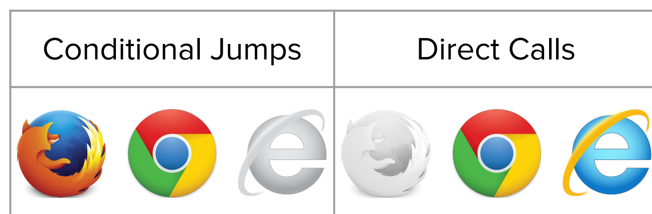
JS

```
function syscall(){
 var j=0;
 i=Math.random();
 //i=Math.random(); 240x
 check(Math.random());
 //i=Math.random(); 10x
}
```

```
function popa(){
 var j=0;
 i=Math.random();
 //i=Math.random(); 232x
 check(Math.random());
 //i=Math.random(); 18x
}
```

Gadgets in Browsers

- Conditional jumps
  - \+ Work in *Chrome* and *Firefox*
  - \- Only upto 2B gadgets possible in *IE*
- Direct calls
  - \+ Work in *Chrome* and *IE*
  - \- Do not work in *Firefox*



Removing the Gadgets...

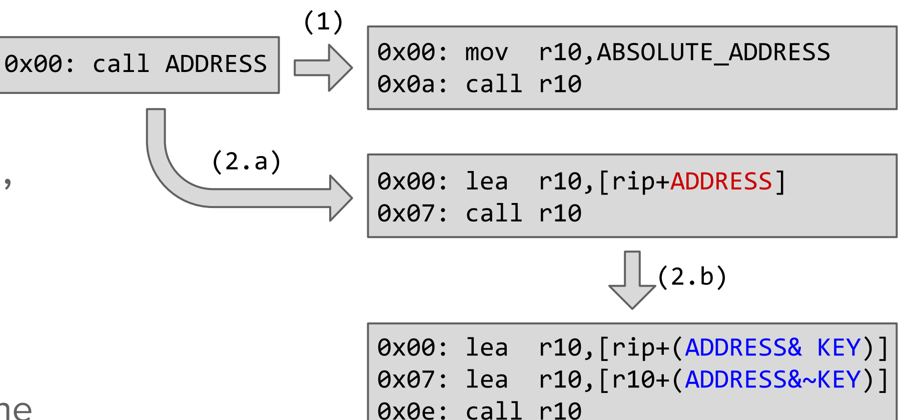
Defense ➜ Direct Calls

For each direct call:

(1) If the address of the callee is known, replace the call with the indirect call

(2) If the address is *not* known:

- (a) convert the direct call into the indirect one
- (b) blind the emitted relative address



Defense ➜ Conditional Jumps

For each conditional jump:

(1) Convert a direct (relative) jump into an indirect one

(2) Add the inverted conditional jump to jump into `<if_body>`

```
0x00: jz ADDRESS
0x06: <if_body>
...


XXXX: <else>
```

(1)

```
0x00: NOP ; 2-byte NOP
0x02: lea r10,[rip+(ADDRESS& KEY)]
0x09: lea r10,[r10+(ADDRESS&~KEY)]
0x10: jmp r10
0x13: <if_body>
...
YYYY: <else>
```

(2)

```
0x00: jnz 0x11
0x02: lea r10,[rip+(ADDRESS& KEY)]
0x09: lea r10,[r10+(ADDRESS&~KEY)]
0x10: jmp r10
0x13: <if_body>
...
YYYY: <else>
```

Defense ➔ Overhead

Cost of the defense in V8:

- V8's JS Benchmark Suite:
  - 2% performance overhead on average
- Microbenchmarks:
  - 14% and 10% overhead on average
- V8's JS Benchmark Suite (Code-size):
  - ≈26% overhead (1,123 kB ➔ 1,411 kB)

	Original	Modified	Overhead
Richards	36,263	35,555	1.95%
DeltaBlue	63,641	62,045	2.51%
Crypto	33,366	32,725	1.92%
RayTrace	77,198	75,488	2.21%
EarleyBoyer	44,900	43,700	2.67%
RegExp	6,525	6,414	1.71%
Splay	21,095	20,479	2.92%
NavierStokes	31,924	31,998	-0.23%
Total	32,255	31,662	1.96%

Summary

- Displacement fields of x86 instructions can be used to inject arbitrary 3 byte values in JIT-compiled code
- Predictable code output from JIT compilers allows adversaries to generate and reuse gadgets without reading the code
- Gadgets in displacements fields can be removed by converting direct jumps/calls into indirect ones

Thank you!