black hat USA 2020

AUGUST 5-6, 2020

BRIEFINGS

NoJITsu: Locking Down JavaScript Engines

Taemin Park[*], Karel Dhondt[†], David Gens[*], Yeoul Na[*], Stijn Volckaert[†], Michael Franz[*]

[*]Department of Computer Science, University of California, Irvine

[†]Department of Computer Science, imec-DistriNet, KU Leuven

Web browser and JavaScript

- Web browsers become essential parts of our daily lives.
- JavaScript fosters rich interaction between browsers and web pages.

SpiderMonkey

V8

JavaScriptCore

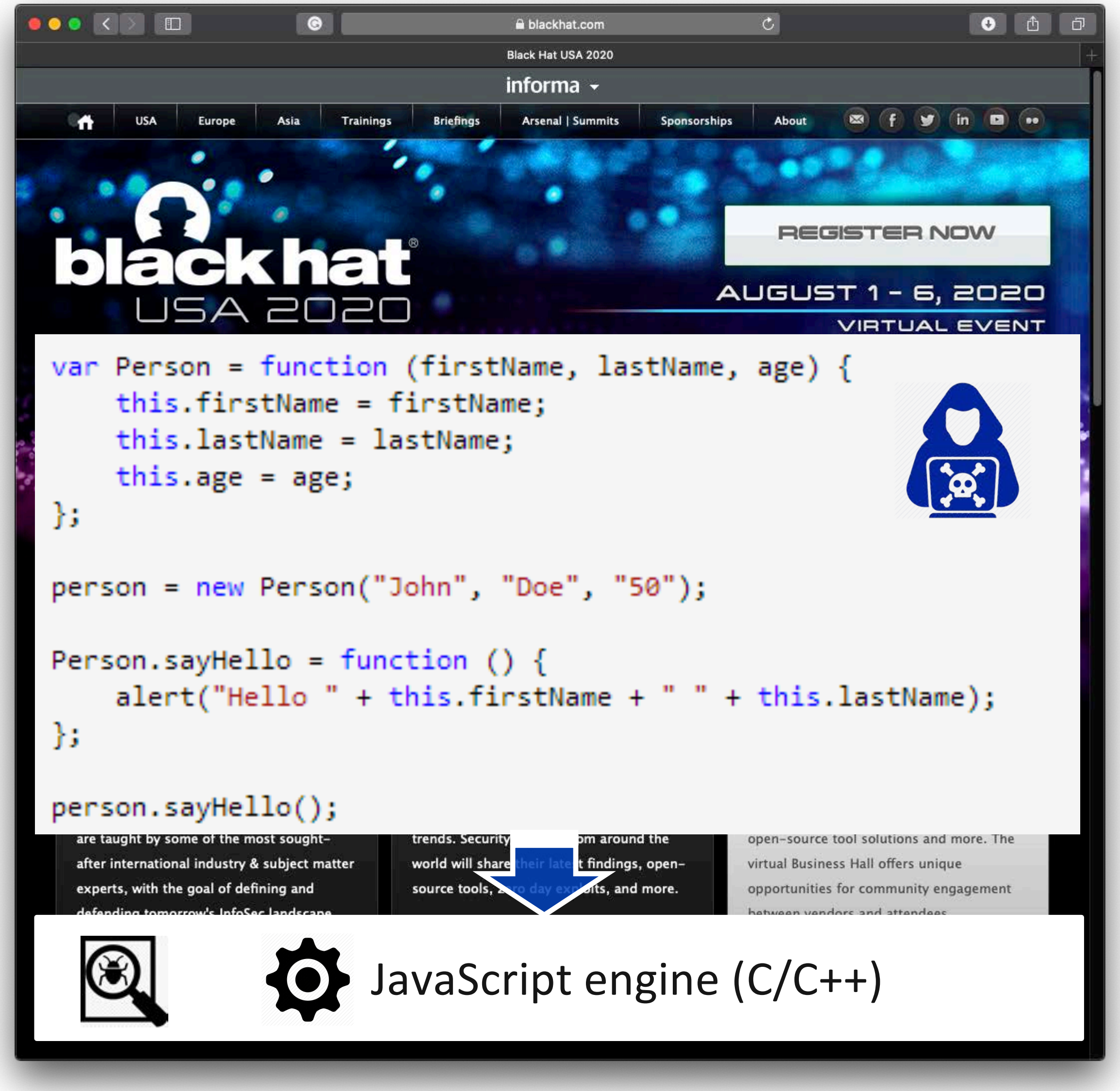
Chakra

Problems in JavaScript Engines

- JavaScript engines are written in an unsafe language such as C/C++.

- JavaScript engines automatically run any script embedded in a webpage.

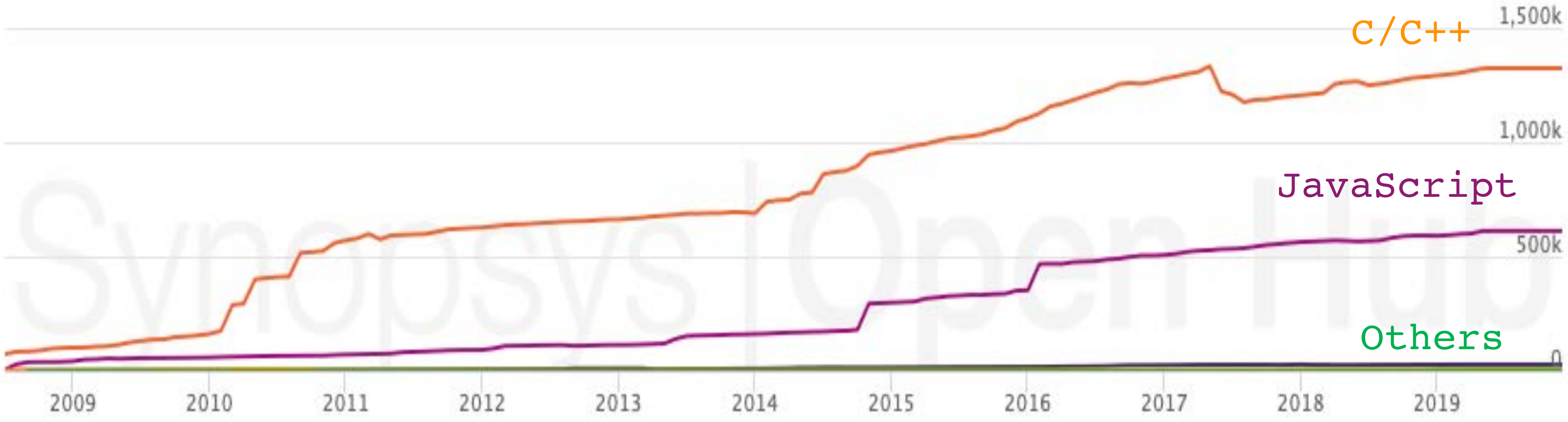
- Attackers trigger a vulnerability to exploit a victim's machine.


blackhat.com
Black Hat USA 2020
informa
USA
Europe
Asia
Trainings
Briefings
Arsenal | Summits
Sponsorships
About
black hat USA 2020
REGISTER NOW
AUGUST 1 – 6, 2020
VIRTUAL EVENT
are taught by some of the most sought-after international industry & subject matter experts, with the goal of defining and defending tomorrow's InfoSec landscape.
trends. Security ... om around the world will share their latest findings, open-source tools, zero day exploits, and more.
open-source tool solutions and more. The virtual Business Hall offers unique opportunities for community engagement between vendors and attendees.
JavaScript engine (C/C++)

```
var Person = function (firstName, lastName, age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
};

person = new Person("John", "Doe", "50");

Person.sayHello = function () {
    alert("Hello " + this.firstName + " " + this.lastName);
};

person.sayHello();
```

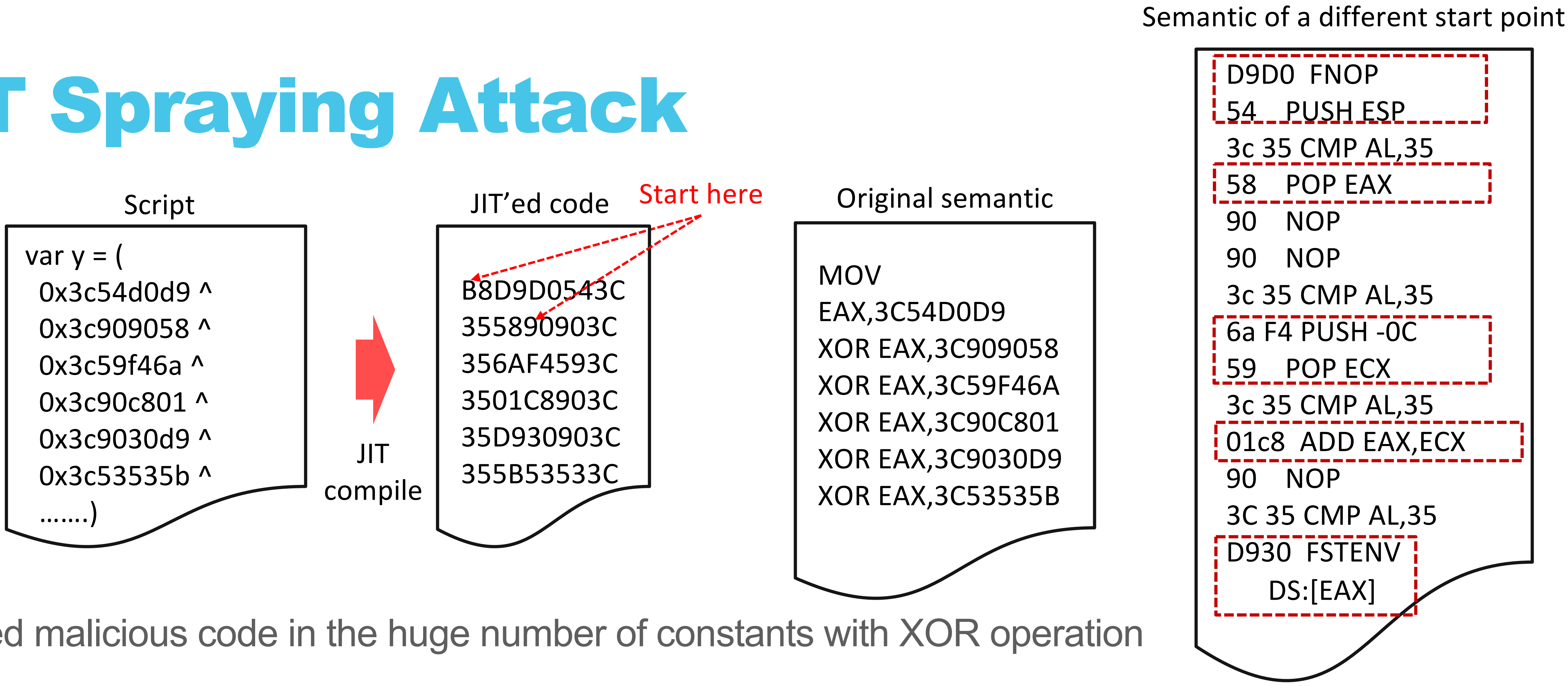
Vulnerable JavaScript Engines

- JavaScript engines are getting bigger
- Hundred of vulnerabilities are found every year



Line of code in V8

JIT Spraying Attack



Script

```
var y = (
  0x3c54d0d9 ^
  0x3c909058 ^
  0x3c59f46a ^
  0x3c90c801 ^
  0x3c9030d9 ^
  0x3c53535b ^
  …….)
```

JIT'ed code

```
B8D9D0543C
355890903C
356AF4593C
3501C8903C
35D930903C
355B53533C
```

Original semantic

```
MOV
EAX,3C54D0D9
XOR EAX,3C909058
XOR EAX,3C59F46A
XOR EAX,3C90C801
XOR EAX,3C9030D9
XOR EAX,3C53535B
```

Semantic of a different start point

```
D9D0  FNOP
54    PUSH ESP
3c 35 CMP AL,35
58    POP EAX
90    NOP
90    NOP
3c 35 CMP AL,35
6a F4 PUSH -0C
59    POP ECX
3c 35 CMP AL,35
01c8  ADD EAX,ECX
90    NOP
3C 35 CMP AL,35
D930  FSTENV
        DS:[EAX]
```

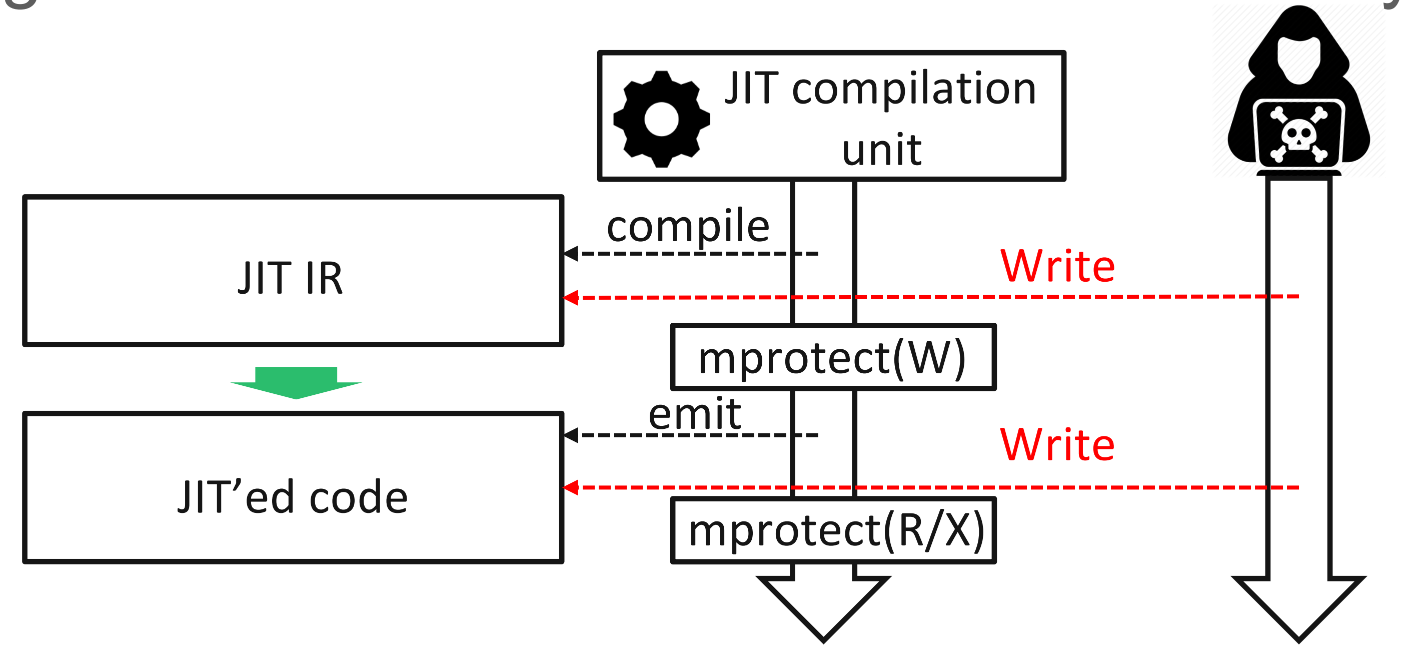
- Embed malicious code in the huge number of constants with XOR operation
- Trigger a vulnerability to jump to the middle of code

Writing JIT-Spray Shellcode for fun and profit, Alexey Sintsov

Athanasakis, M., Athanasopoulos, E., Polychronakis, M., Portokalidis, G., & Ioannidis, S. (2015). The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. Presented at the Proceedings 2015 Network and Distributed System Security Symposium.

Advanced Attacks and Defenses on JIT’ed code

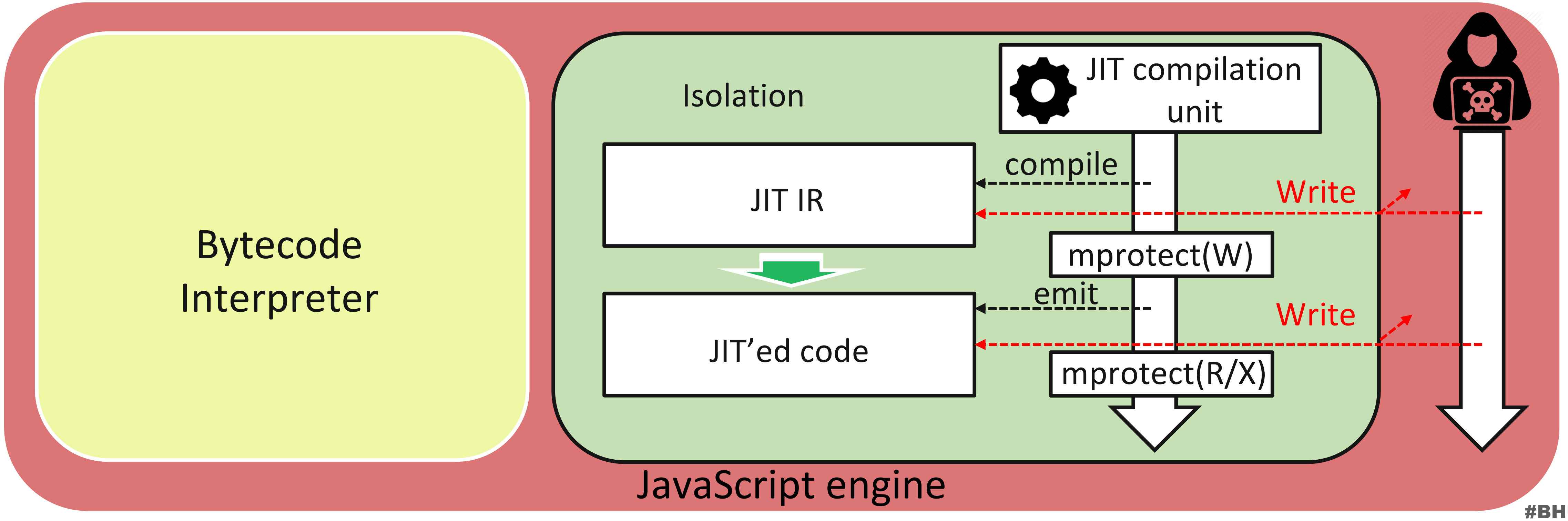
- Attack vectors from multi-threading environment
  - Corrupt JIT IR when it is being compiled
  - Write on JIT’ed region when JIT’ed code is emitted to memory



Song, C., Zhang, C., Wang, T., Lee, W., & Melski, D. (2015). Exploiting and Protecting Dynamic Code Generation. Presented at the Proceedings 2015 Network and Distributed System Security Symposium.
Frassetto, T., Gens, D., Liebchen, C., & Sadeghi, A.-R. (2017). JITGuard: Hardening Just-in-time Compilers with SGX (pp. 2405–2419). New York, New York, USA: ACM

Advanced Attacks and Defenses on JIT’ed code

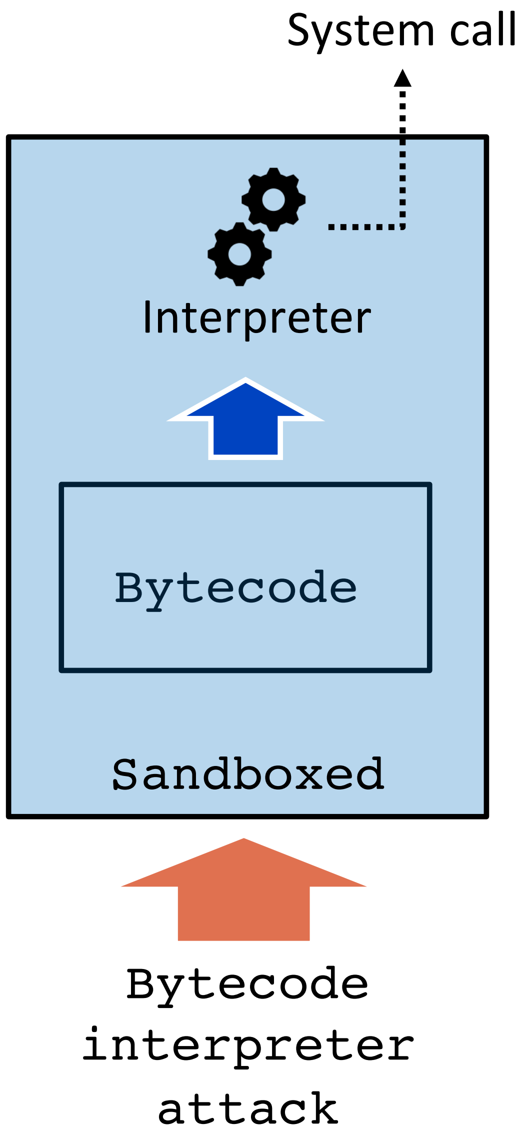
- Putting JIT compilation into a separate process or trusted execution environment



Song, C., Zhang, C., Wang, T., Lee, W., & Melski, D. (2015). Exploiting and Protecting Dynamic Code Generation. Presented at the Proceedings 2015 Network and Distributed System Security Symposium.
Frassetto, T., Gens, D., Liebchen, C., & Sadeghi, A.-R. (2017). JITGuard: Hardening Just-in-time Compilers with SGX (pp. 2405–2419). New York, New York, USA: ACM

Contribution

- Bytecode interpreter attack
  - Corrupt the bytecode interpreter to execute arbitrary systems calls
- Defense mechanisms to protect JavaScript engines
  - The bytecode interpreter attack
  - Code-injection, code-reuse attacks



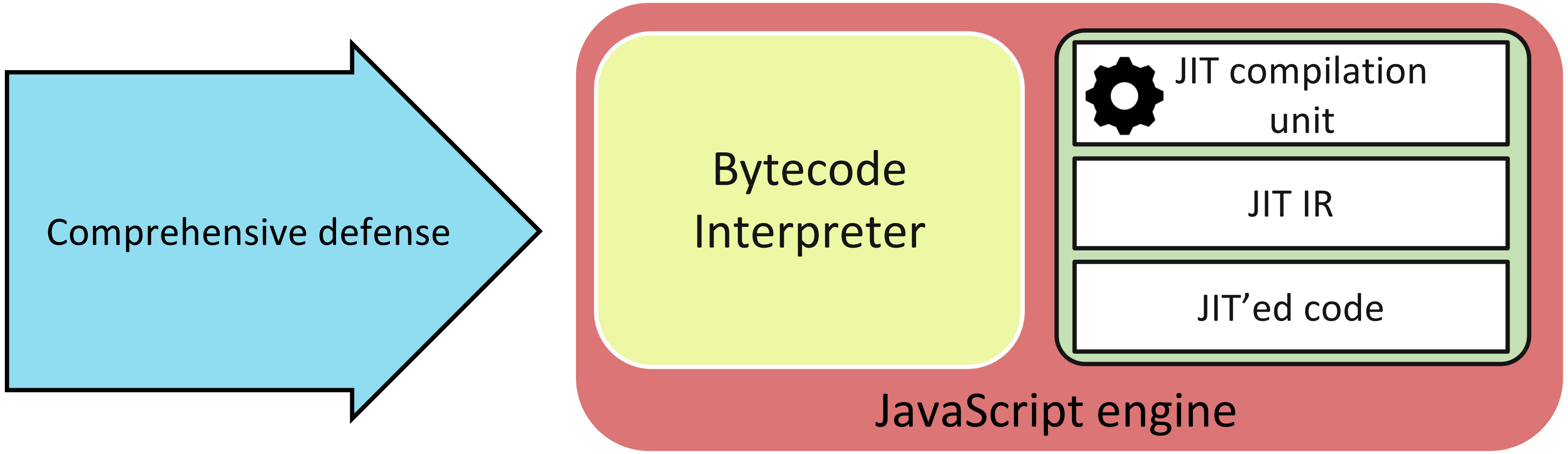
System call

Interpreter

Bytecode

Sandboxed

Bytecode interpreter attack



Comprehensive defense

Bytecode Interpreter

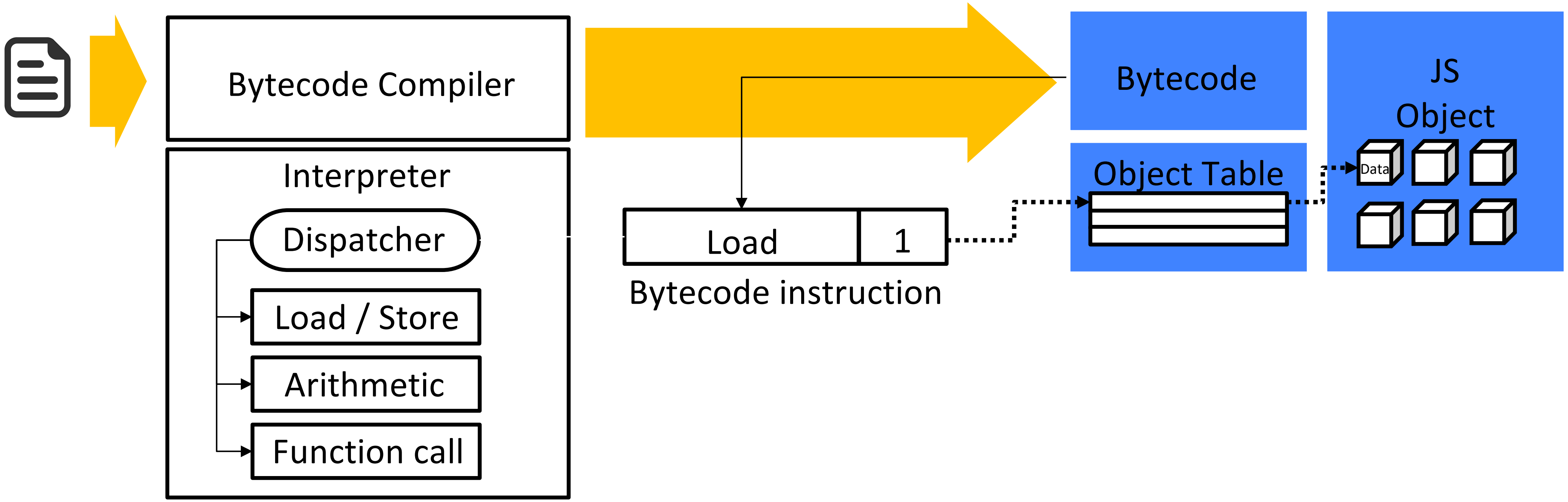
JIT compilation unit

JIT IR

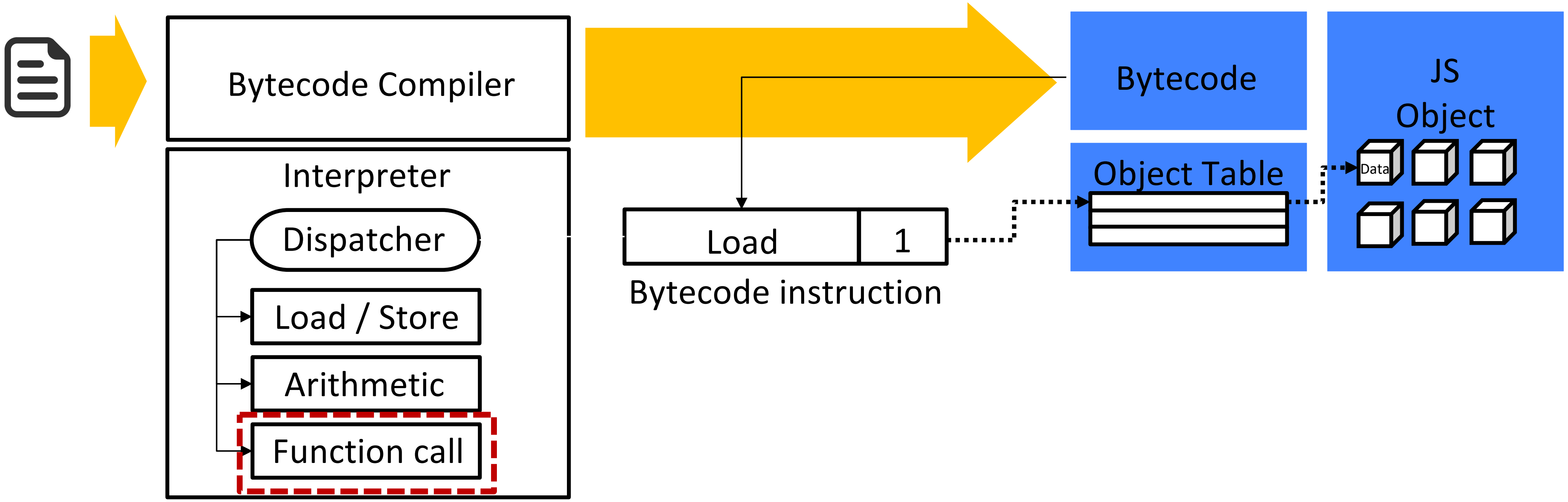
JIT'ed code

JavaScript engine

Bytecode Execution Flow



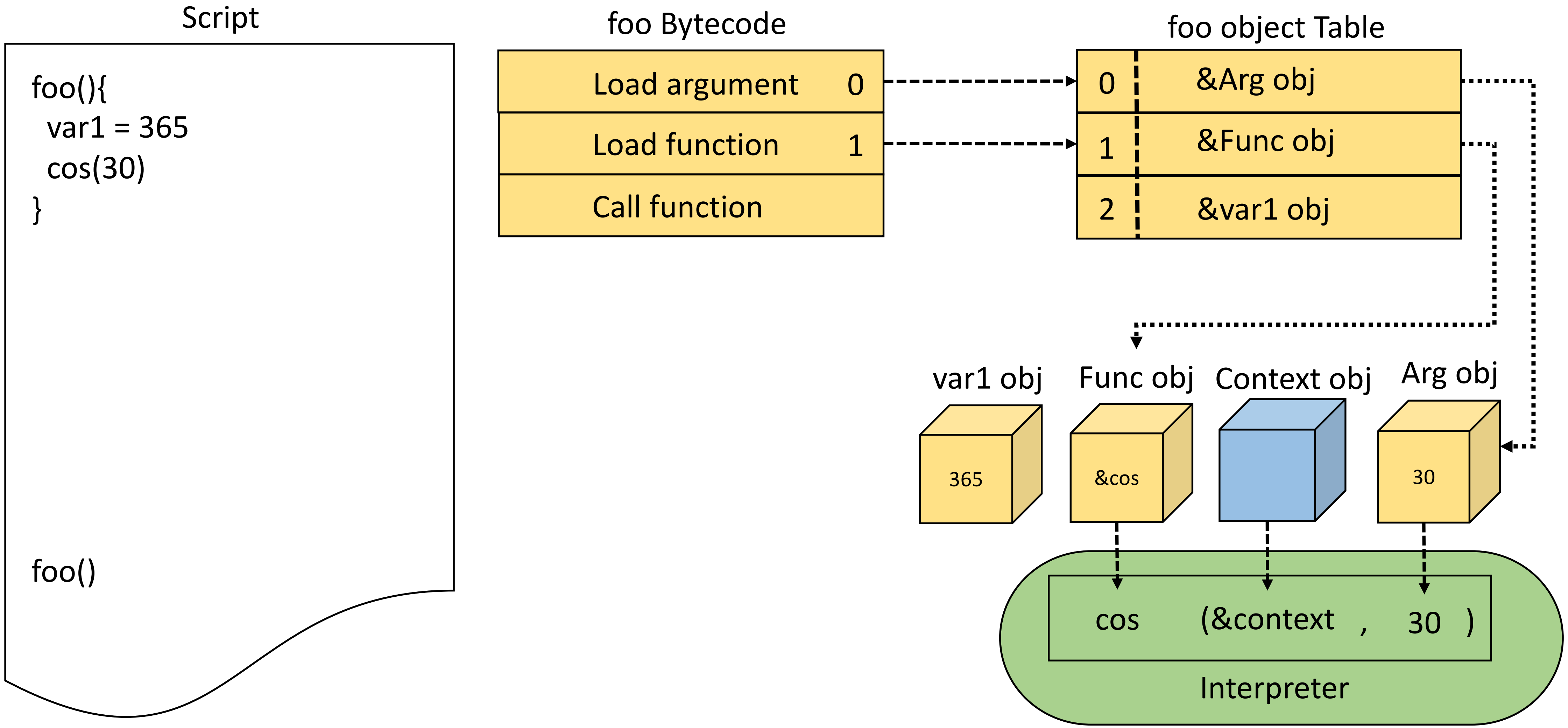
Bytecode Execution Flow



Threat model

- Memory-corruption vulnerability
  - Arbitrary read / write capability
- Code-injection defense
  - W⊕X enforced
- Light weight code-reuse defense
  - ASLR, coarse-grained CFI

Bytecode Interpreter Attack



Script

```
foo(){
  var1 = 365
  cos(30)
}

foo()
```

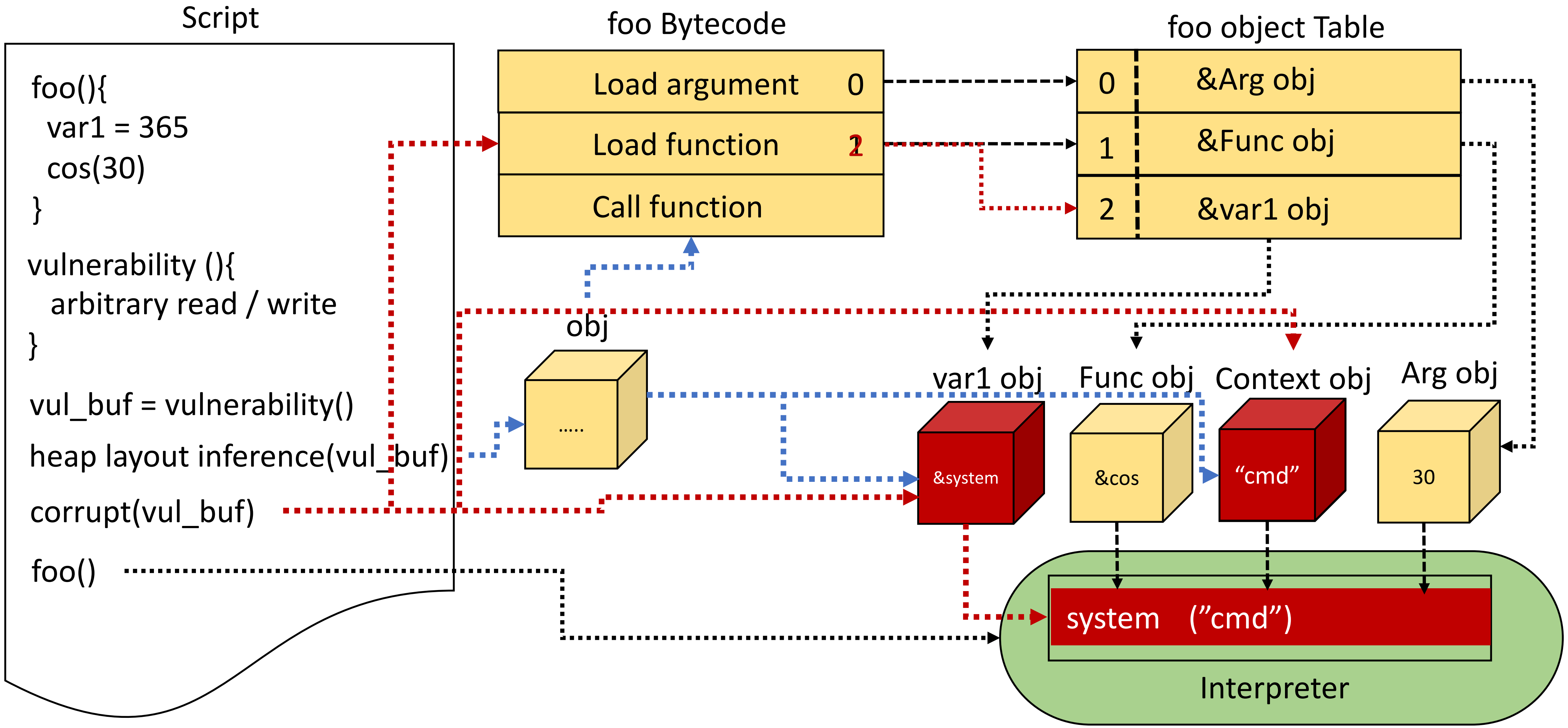
foo Bytecode

Load argument	0
Load function	1
Call function	

foo object Table

0	&Arg obj
1	&Func obj
2	&var1 obj

Bytecode Interpreter Attack



Script

```
foo(){
 var1 = 365
 cos(30)
}
vulnerability (){
  arbitrary read / write
}
vul_buf = vulnerability()
heap layout inference(vul_buf)
corrupt(vul_buf)
foo()
```

foo Bytecode

Load argument	0
Load function	1 2
Call function	

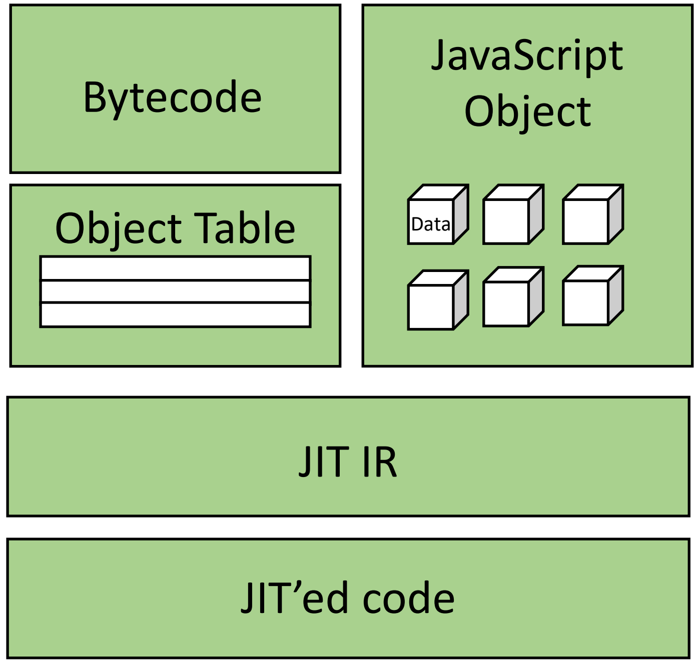
foo object Table

0	&Arg obj
1	&Func obj
2	&var1 obj

Comprehensive Defense: NoJITsu

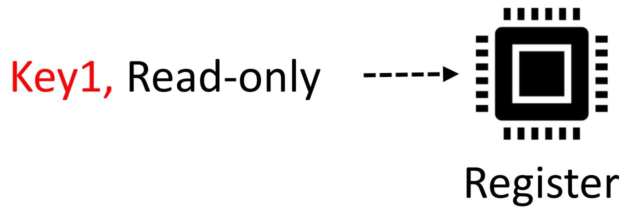
- Protect core data in script engine execution
  - Bytecode, object tables, data objects, JIT IR, and JIT'ed code

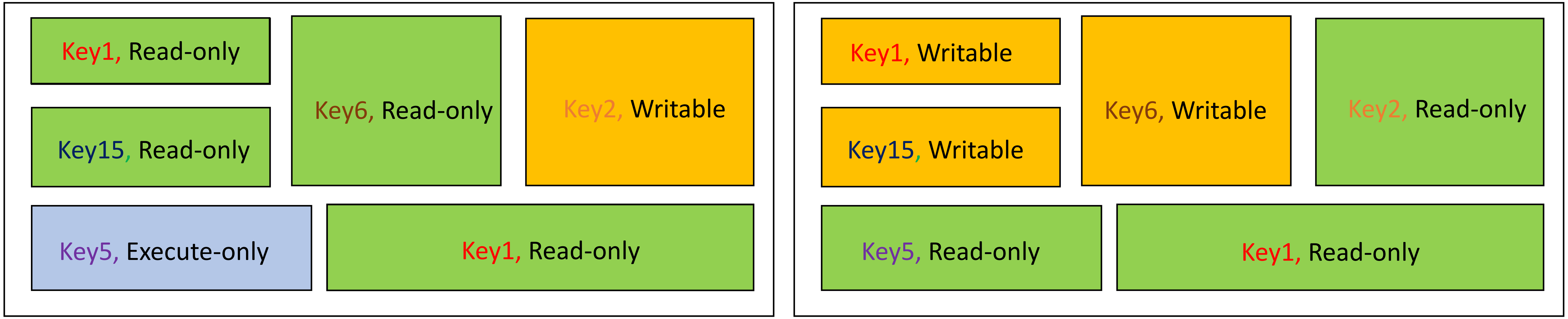
- Fine-Grained Memory access control over the core data.
  - Minimize the permission of data as small as possible
  - Challenge: Overhead from enforcing fine-grained memory access control



Intel Memory Protection Key

- Fine-grained memory access control through Intel Memory Protection Key
- Intel MPK (Memory Protection Key)
  - A new hardware feature to control the protection of memory
  - Fast permission change
  - Support execute-only permission
  - Thread local

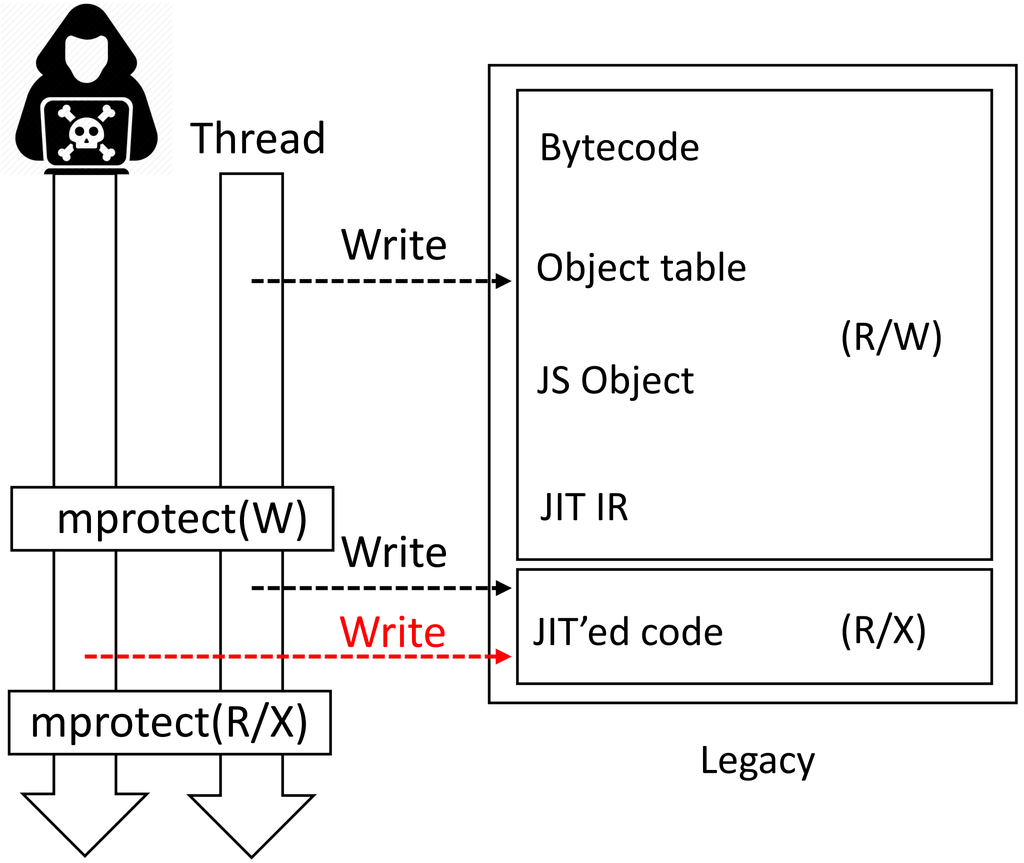




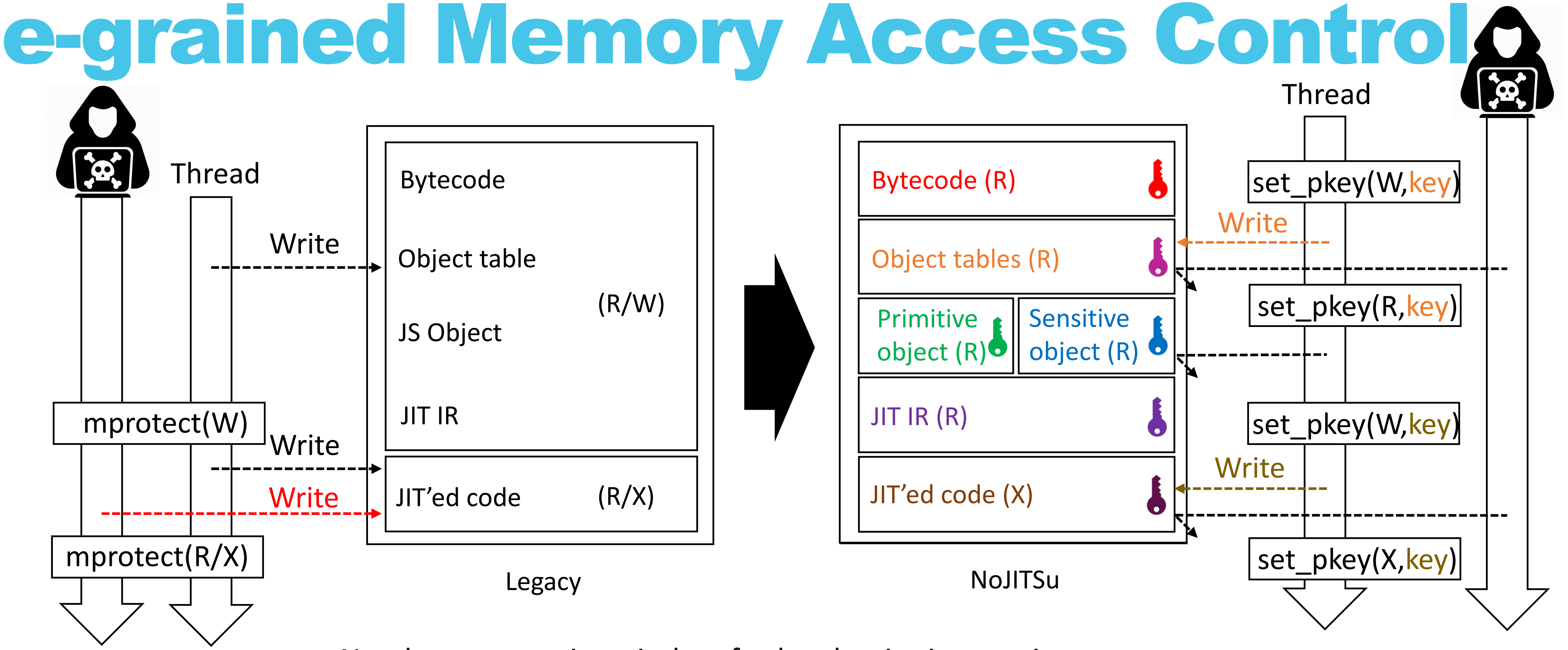
Memory (Thread1)

Memory (Thread2)

Fine-grained Memory Access Control



Fine-grained Memory Access Control

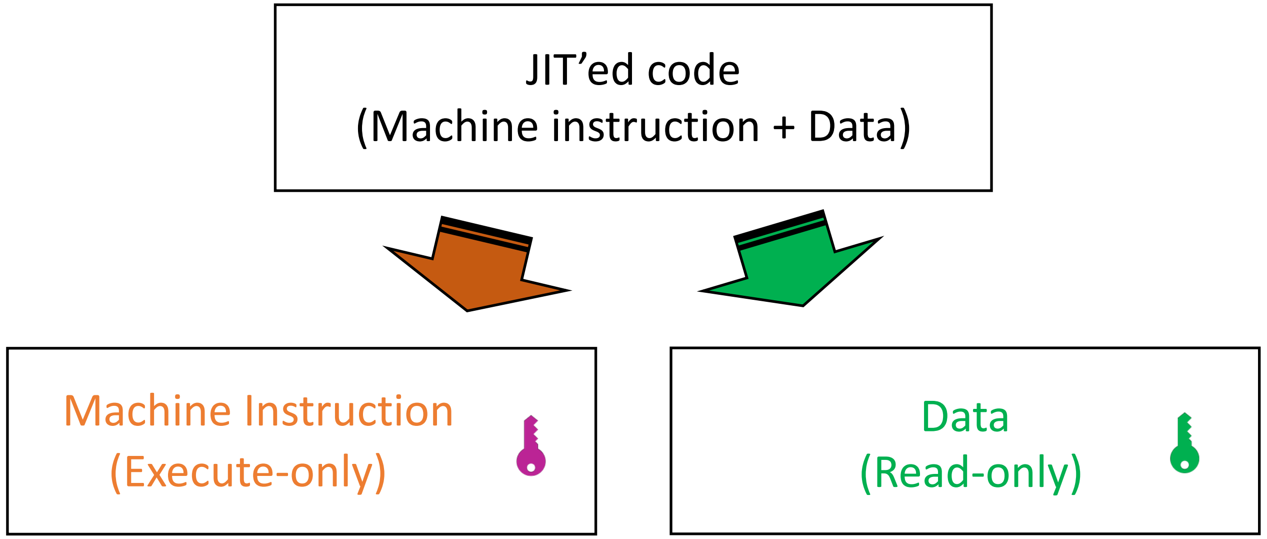


- Need to open write window for legal write instructions
  - How do we find all write instructions to each kind of data.
  - How do we implement permission changes for them.

Bytecode, Object Table, JIT IR and JIT'ed Code

- Bytecode, indirection table
  - Only need write permission at bytecode compilation
- JIT'ed code, JIT IR
  - Only need write permission at JIT compilation
  - JIT'ed code contains data needing read-permission (Jump table, Large constant)

```
Compile_bytecode()
{
    …..
    …..
    saved_pkru = set_pkru(W, key_bytecode)
    write bytecode
    recover_pkru(saved_pkru)
    …..
    …..
}
```



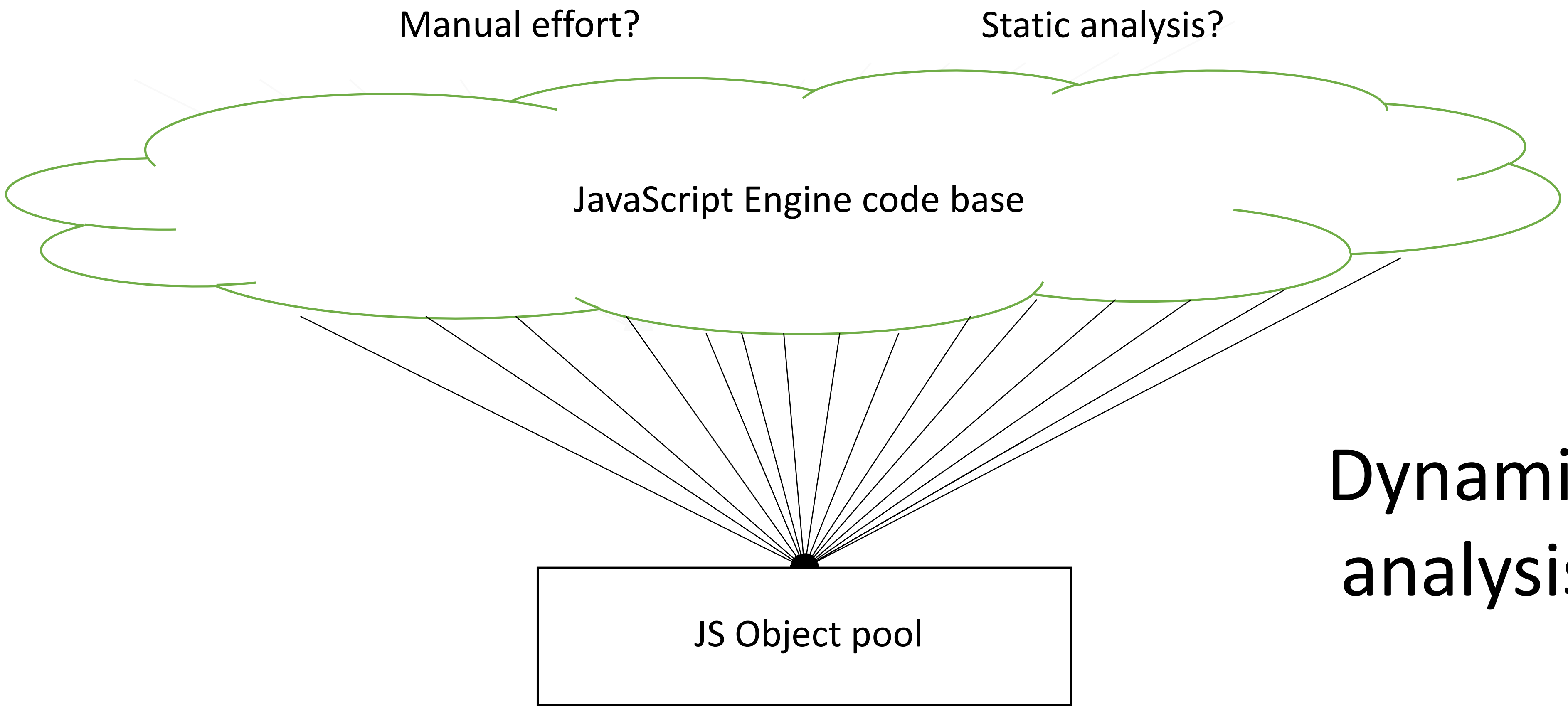
JIT'ed code
(Machine instruction + Data)

Machine Instruction
(Execute-only)

Data
(Read-only)

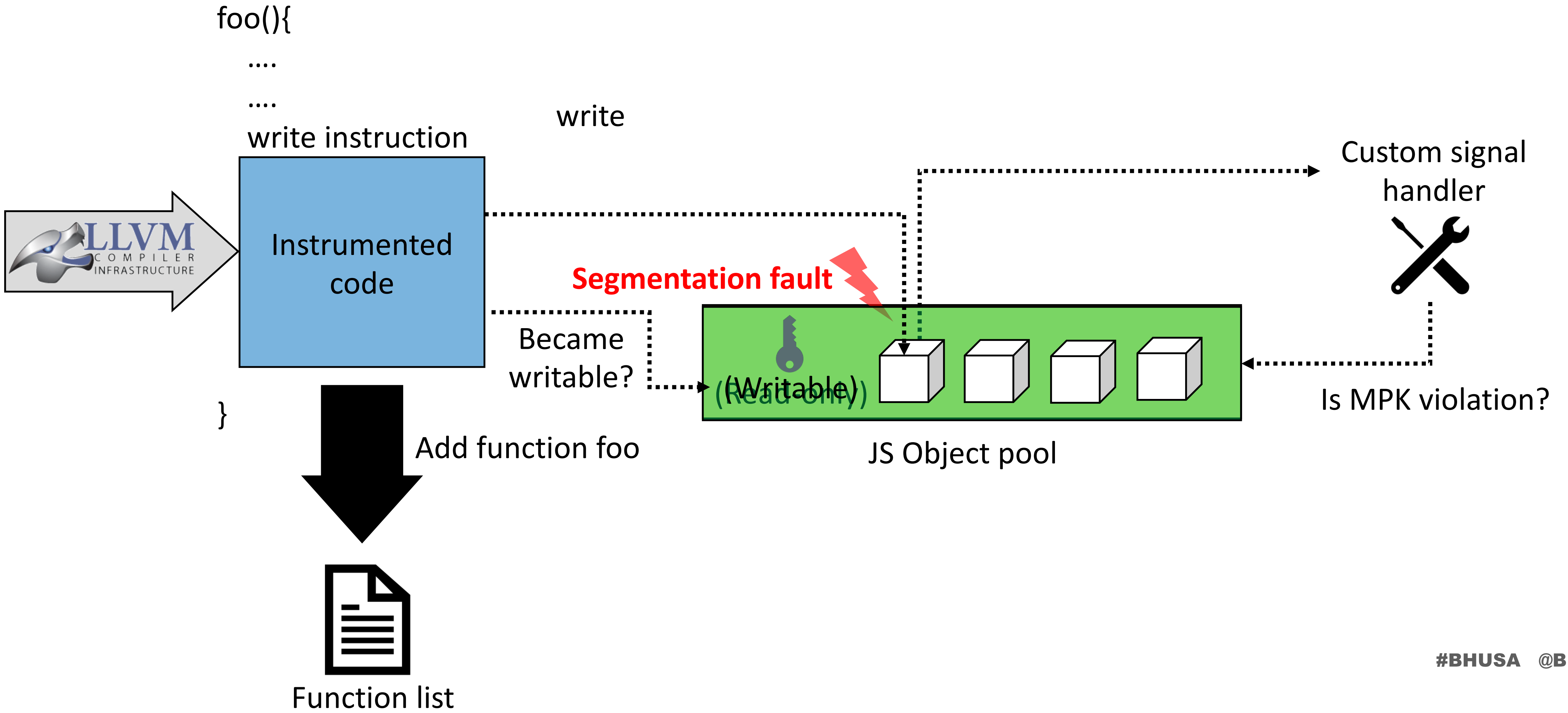
JavaScript Object

There are a huge number of write access instructions to data object throughout a JavaScript code base.



Dynamic analysis

JavaScript Object - Dynamic Analysis



foo(){

....

....

write instruction

}

Instrumented code

write

Custom signal handler

Segmentation fault

Became writable?

(Writable)

(Read-only)

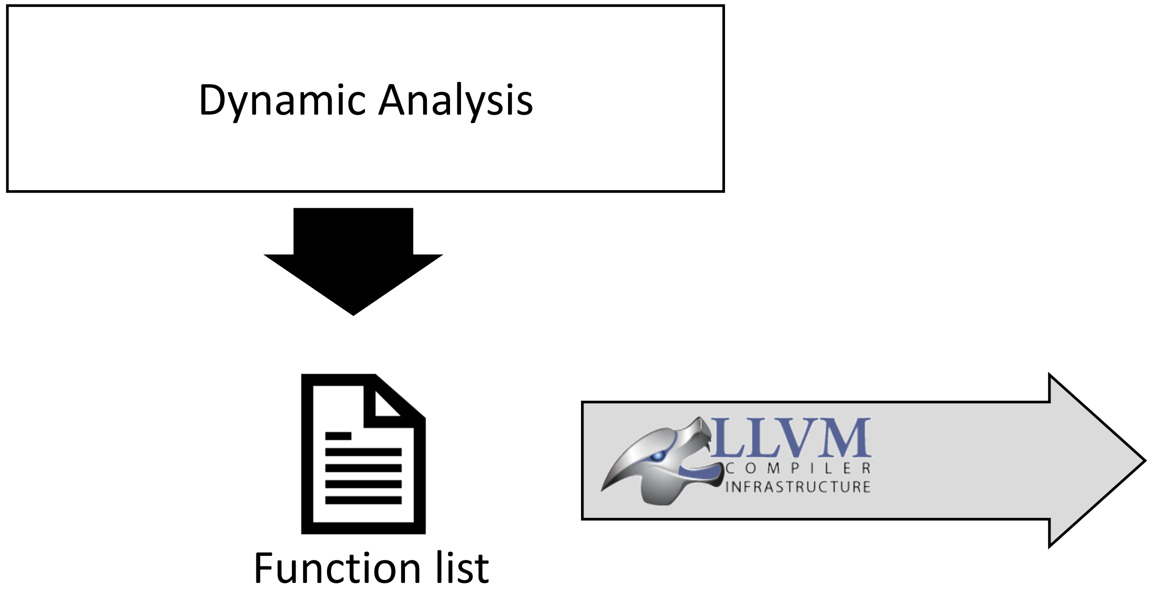
Is MPK violation?

JS Object pool

Add function foo

Function list

JavaScript Object - Enforcement

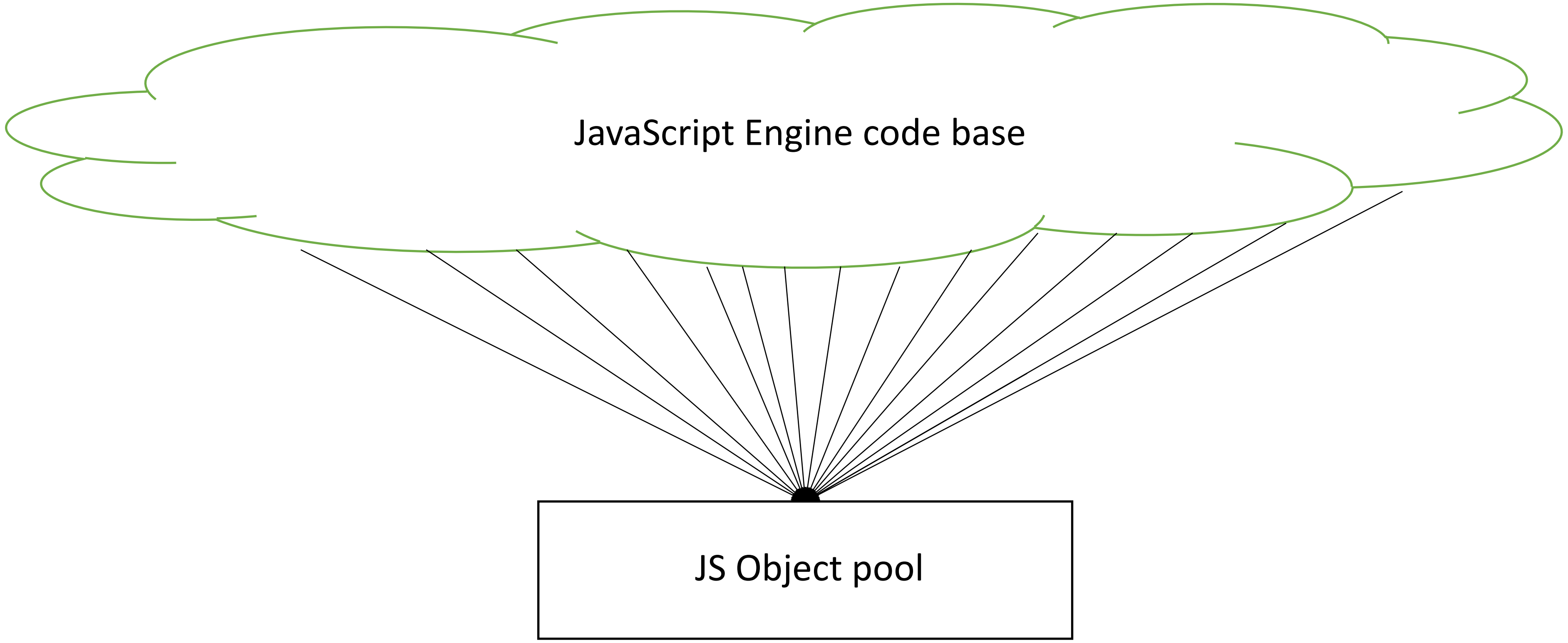


Dynamic Analysis

Function list

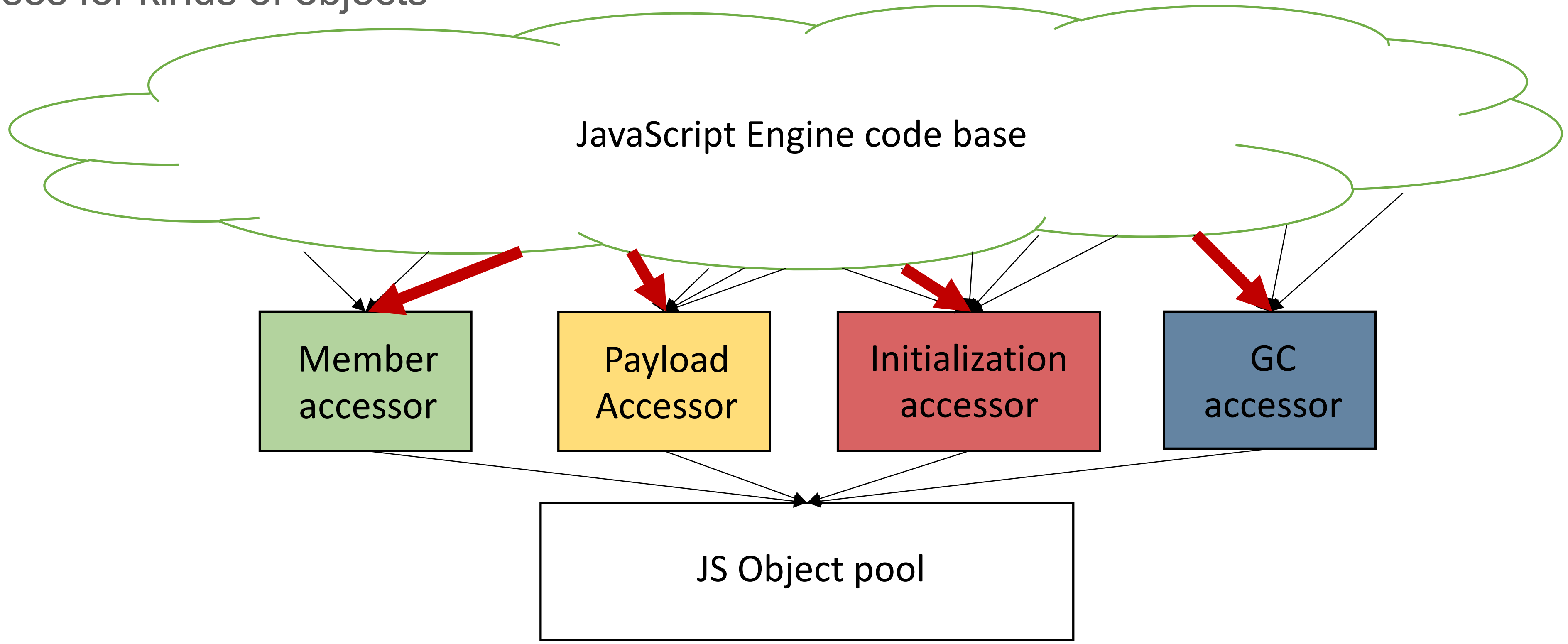
```
                        saved_pkru = set_pkru(W, key_obj)
                        for(I = 0 ; I < 100000 ; i++)
                        {
                            foo();
                        }
                        recover_pkru(saved_pkru)
foo()
{
    saved_pkru = set_pkru(W, key_obj)
    …
    …
    recover_pkru(saved_pkru)
}
```

Dynamic Analysis – Input Set



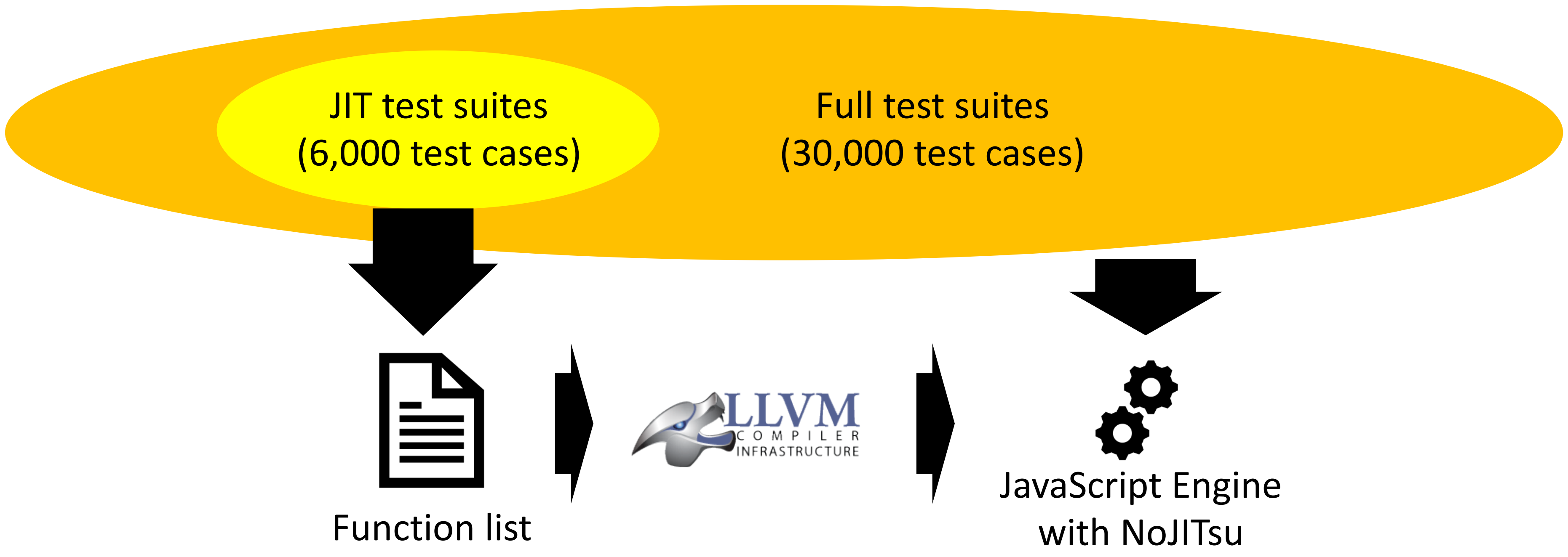
Dynamic Analysis – Input Set

- Member accessor, Payload Accessor, Initialization accessor, GC accessor
- Gateways to write on JS object and extensively shared among other functions
- Use official JavaScript test suites as our input set
  - Include test cases for kinds of objects



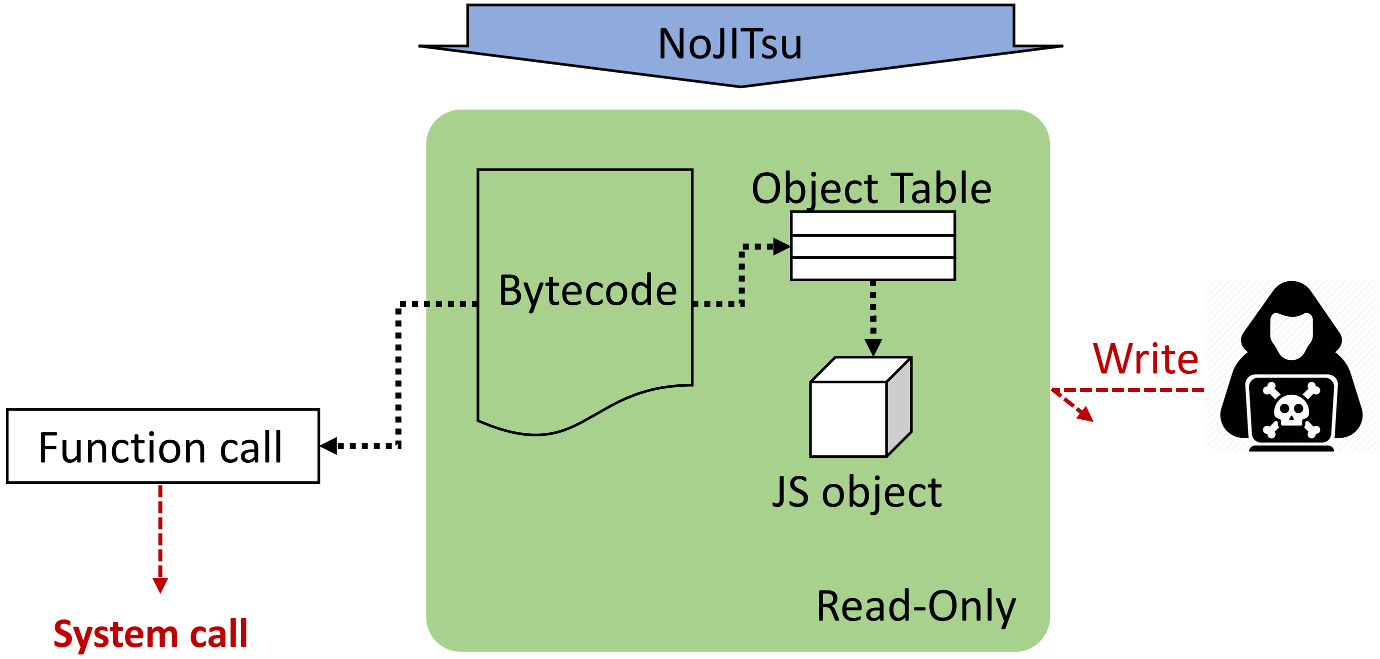
Accessing Coverage of Dynamic Analysis

- Pick only 1/6 of full test suites as input set for dynamic analysis
- Successfully run full test suites without error



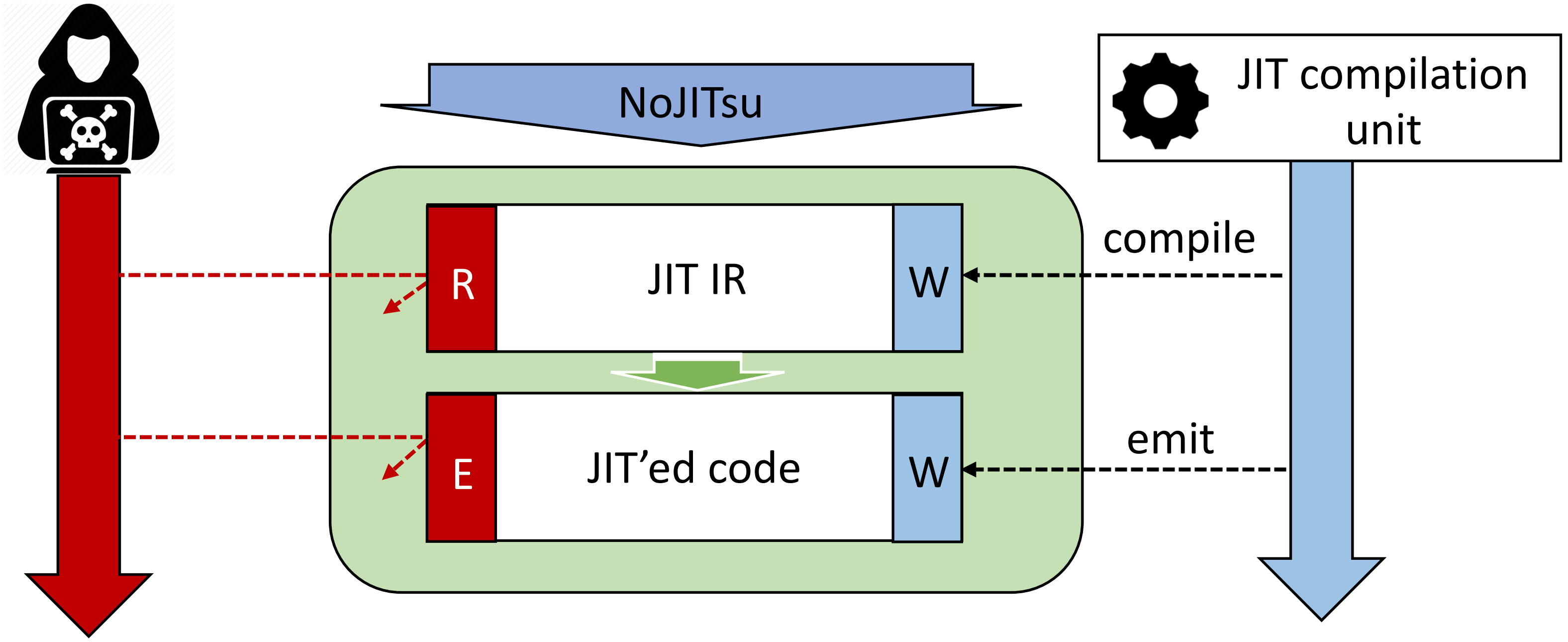
Attack Analysis

Bytecode interpreter attack



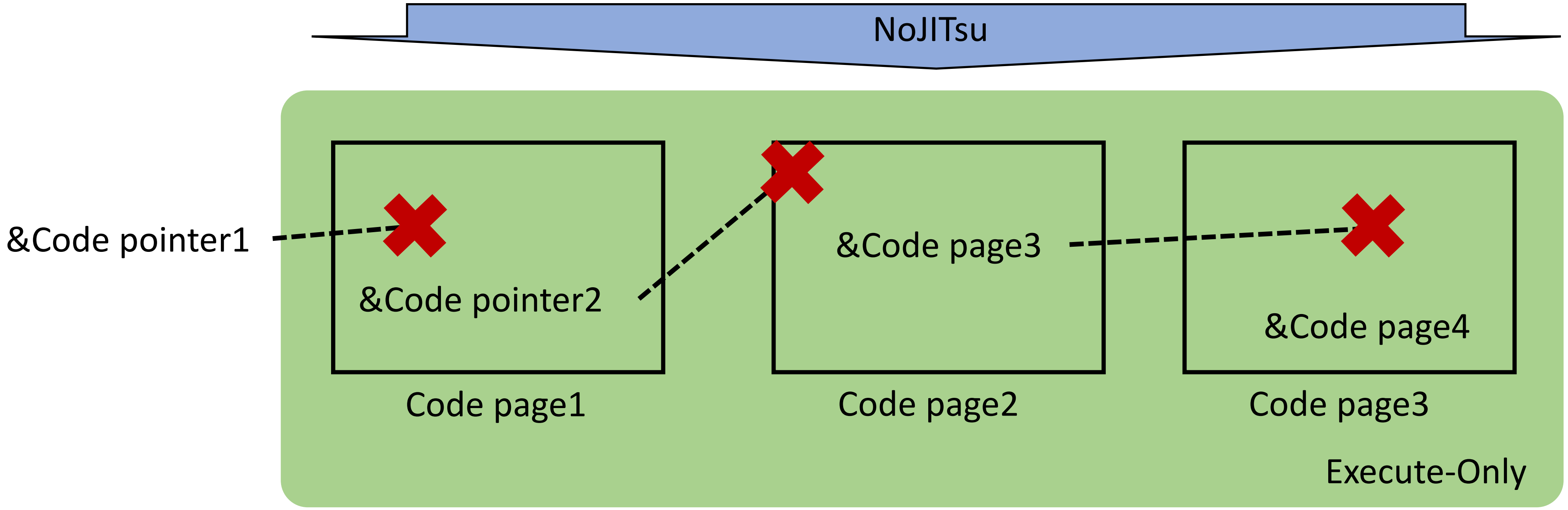
Attack Analysis

JIT code injection attacks



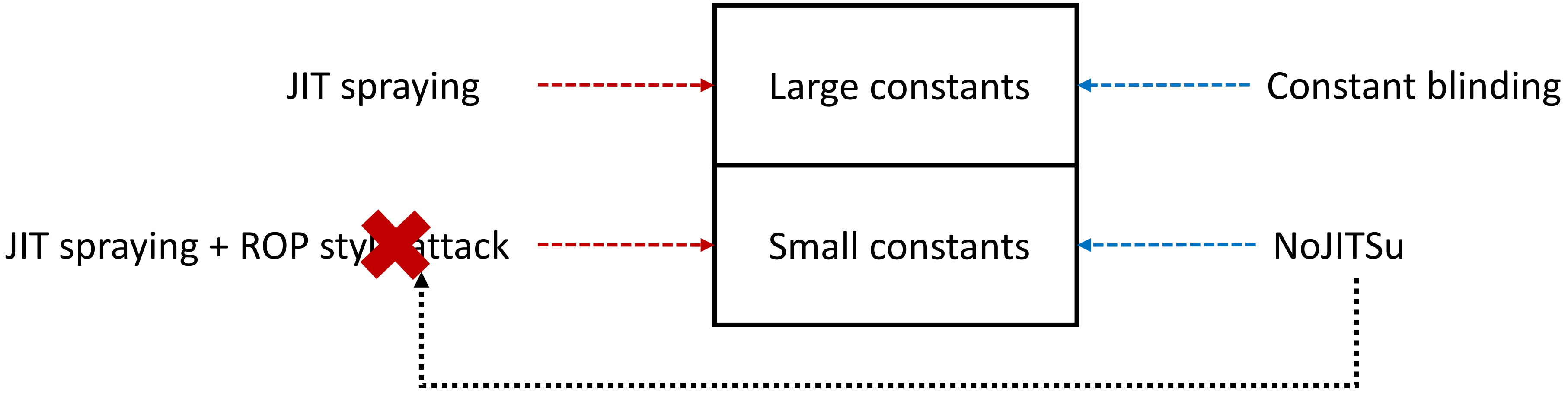
Attack Analysis

Advanced code-reuse attack (JIT-ROP)



Attack Analysis

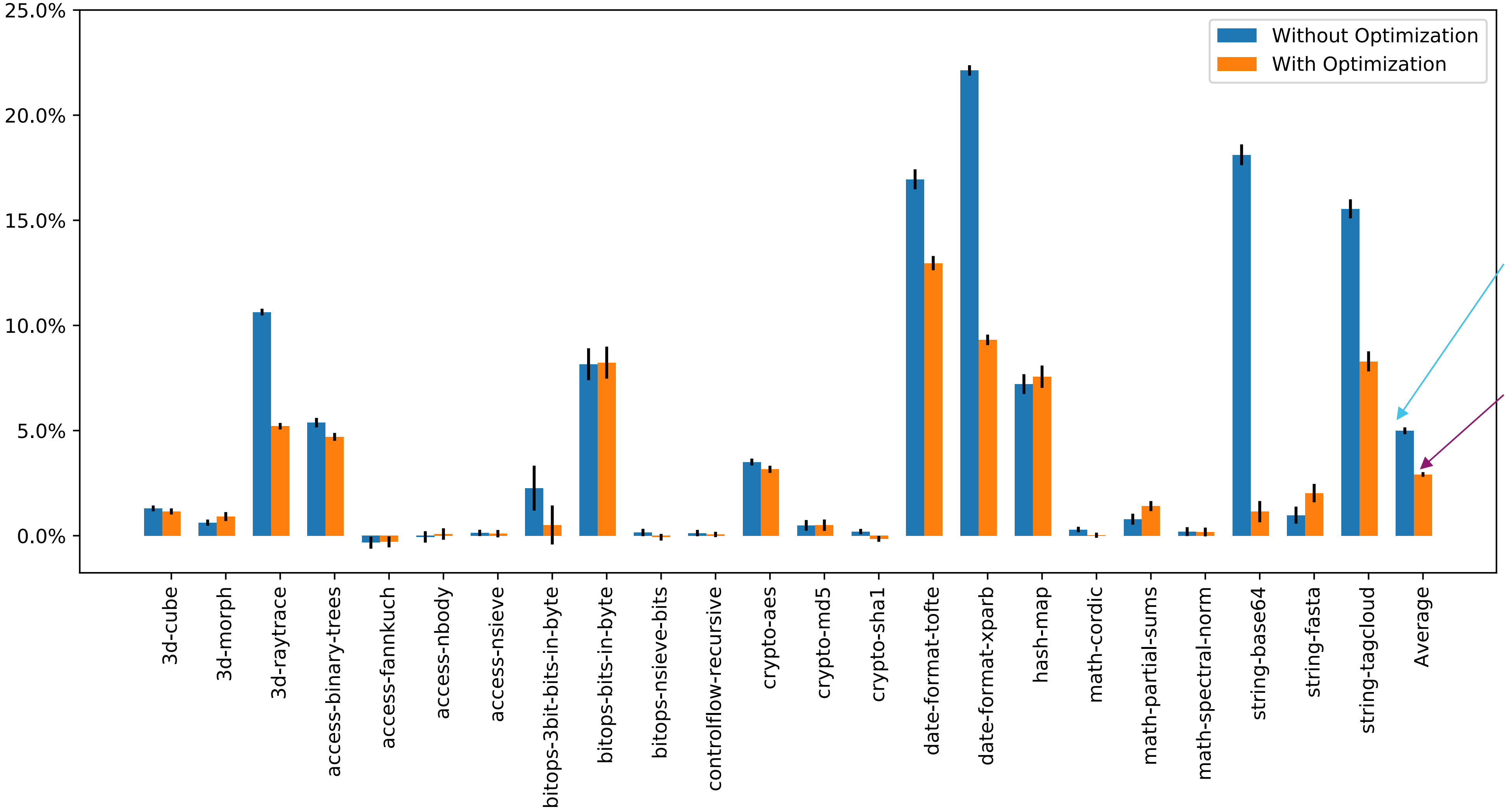
JIT spraying
- Combination of constant blinding and NoJITSu



Performance Evaluation

- Implemented NoJITsu on Spidermonkey.
- LongSpider benchmarks (longer version of the standard JavaScript benchmark suite)
- Intel Xeon silver 4112 machine under Ubuntu 18.04.1 LTS

Evaluation



5% overhead

3% overhead

Conclusion

- Demonstrate a new attack that leverages the interpreter to execute arbitrary shell commands
- Propose NoJITsu, hardware-backed fine-grained memory access protection for JS engines
- Evaluate our defense, showing the effectiveness in code-reuse and injection attack and our bytecode interpreter attack on JS engines with a moderate overhead

black hat
USA 2020

AUGUST 5-6, 2020
BRIEFINGS

Thank you

Q&A

#BHUSA @BLACKHATEVENTS