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

NDSS 2020

Importance of JavaScript Engine Protection

- Every browser has a JavaScript engine
- JavaScript engines are always exposed to malicious scripts

JIT Spraying

Start here

Script

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

JIT compile

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

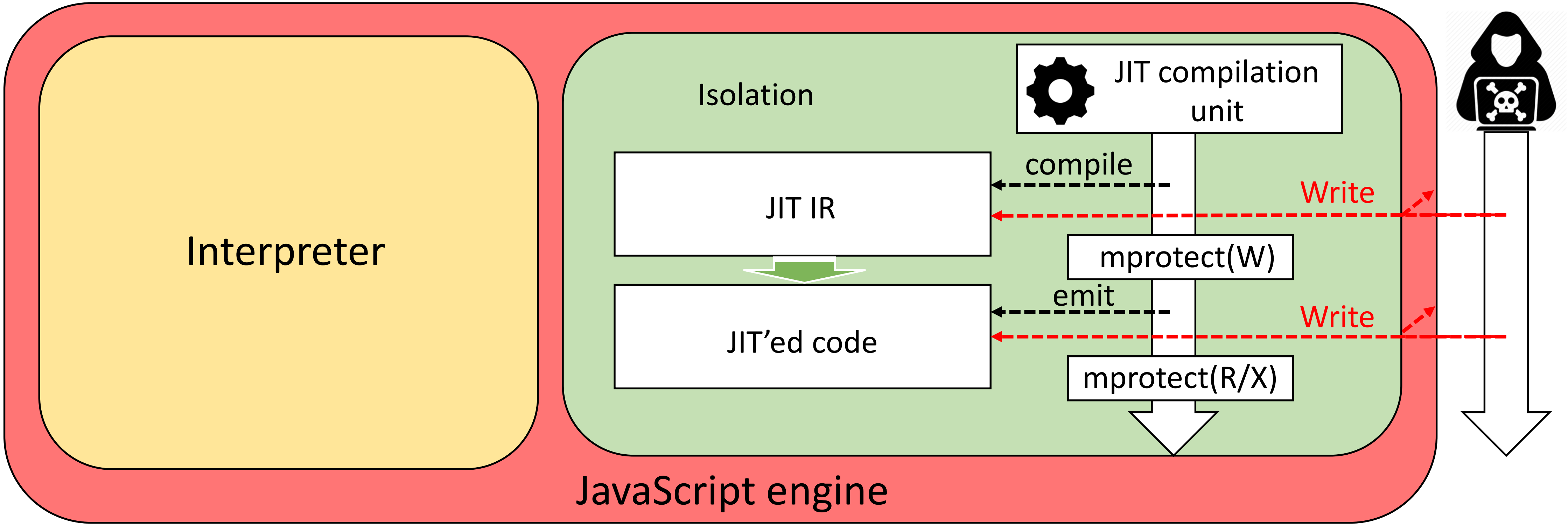
```
D9D0  FNOP
54    PUSH ESP
3c 35 CMP AL,35
58    POP EAX
90    NOP
90    NOP
3c 35 CMP AL,35
6a F4 PUSH -0C
59    POP ECX
3c 35 CMP AL,35
01c8  ADD EAX,ECX
90    NOP
3C 35 CMP AL,35
D930  FSTENV
      DS:[EAX]
```

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

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 utilizing race condition
  - Corrupt JIT IR when it is being compiled
  - Write on JIT'ed region when JIT'ed code is emitted to memory
- Putting JIT compilation into a separate process or trusted execution environment

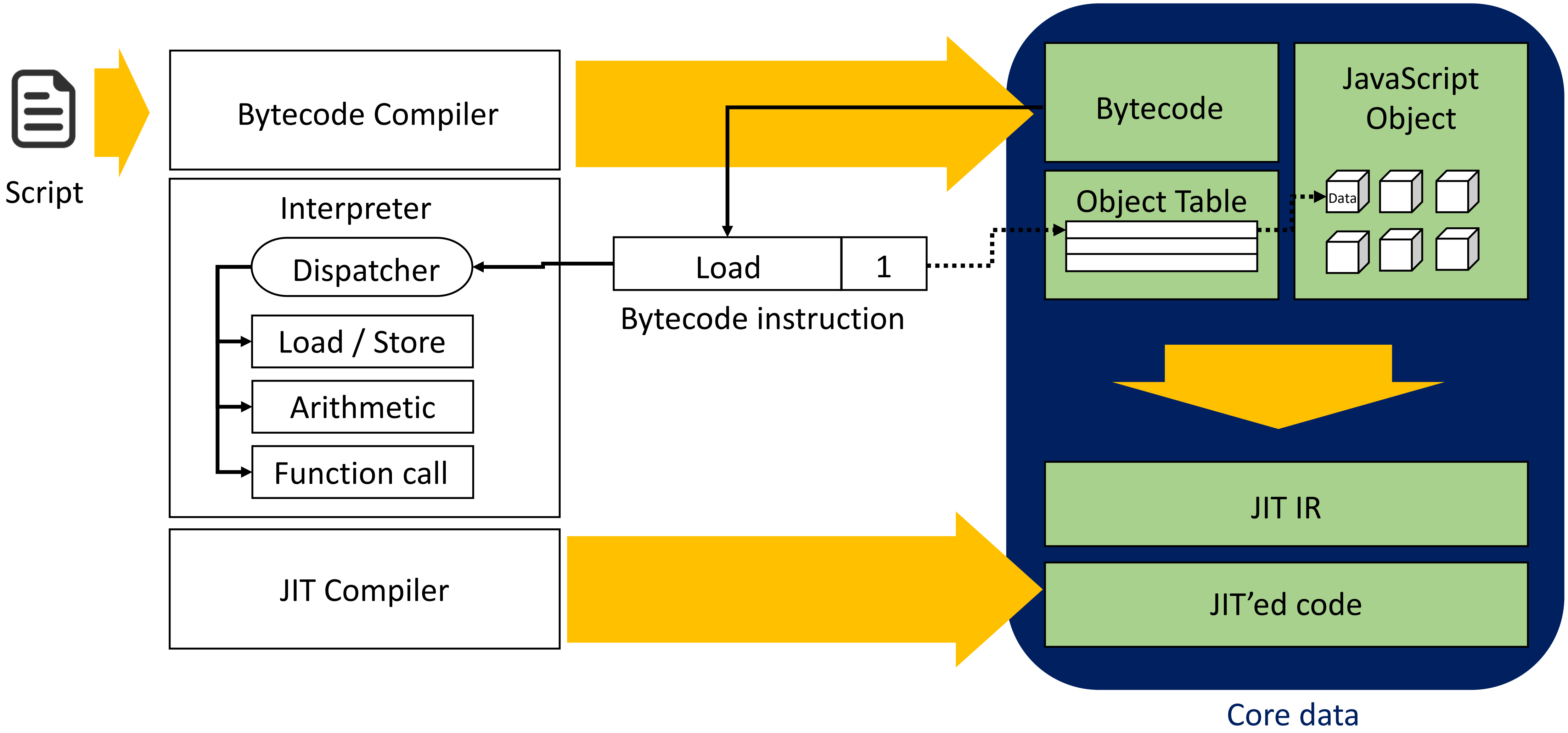


Song, C., Zhang, C., Wang, T., Lee, W., & Melski, D. (2015). Exploiting and Protecting Dynamic Code Generation. Presented at the Proceedings 2015 Network and Distributed System Security Symposium.
Frassetto, T., Gens, D., Liebchen, C., & Sadeghi, A.-R. (2017). JITGuard: Hardening Just-in-time Compilers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

Contribution

- Attack: Bytecode Interpreter attack
  - Change the behavior of interpreter execution by corrupting core data of the interpreter
  - Lead to arbitrary system call
- Defense: NoJITsu
  - Fine-Grained Memory access control
  - Protect JIT’ed code and the core data of interpreter
- Thorough Evaluation

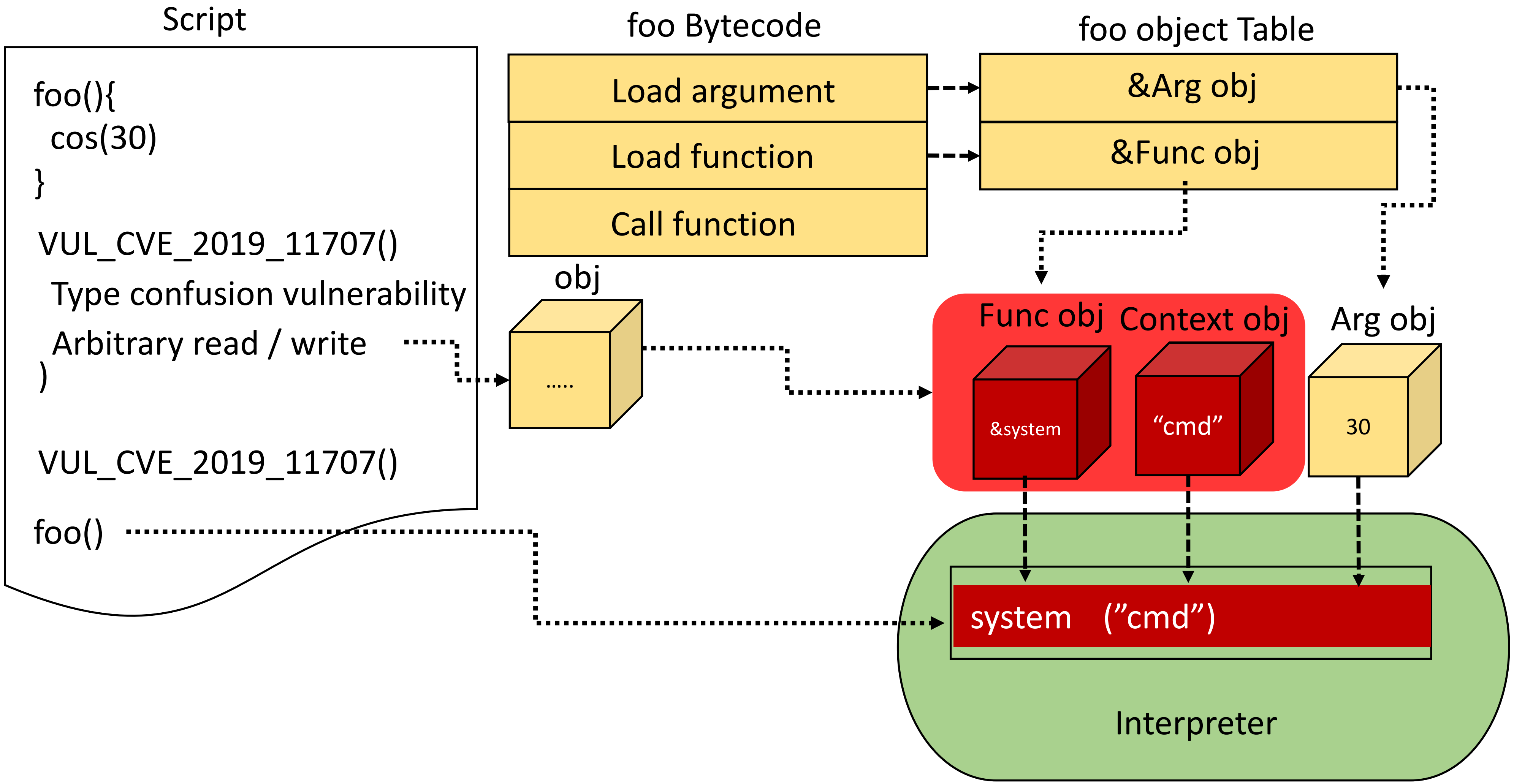
JavaScript Engine Execution Flow and Core Data



Bytecode Interpreter Attack

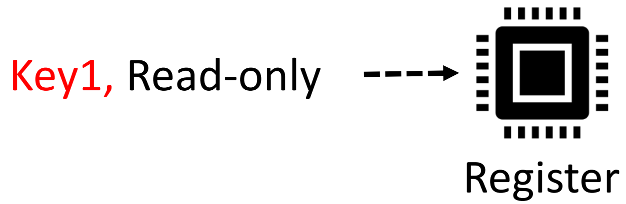
- Corrupt the function call routine to run a system call
- Attack on the SpiderMonkey
- Threat model
    - Memory-corruption vulnerability
        - Arbitrary read / write capability
    - Code-injection defense
        - W$\oplus$X enforced
    - Light weight code-reuse defense
        - ASLR, coarse-grained CFI

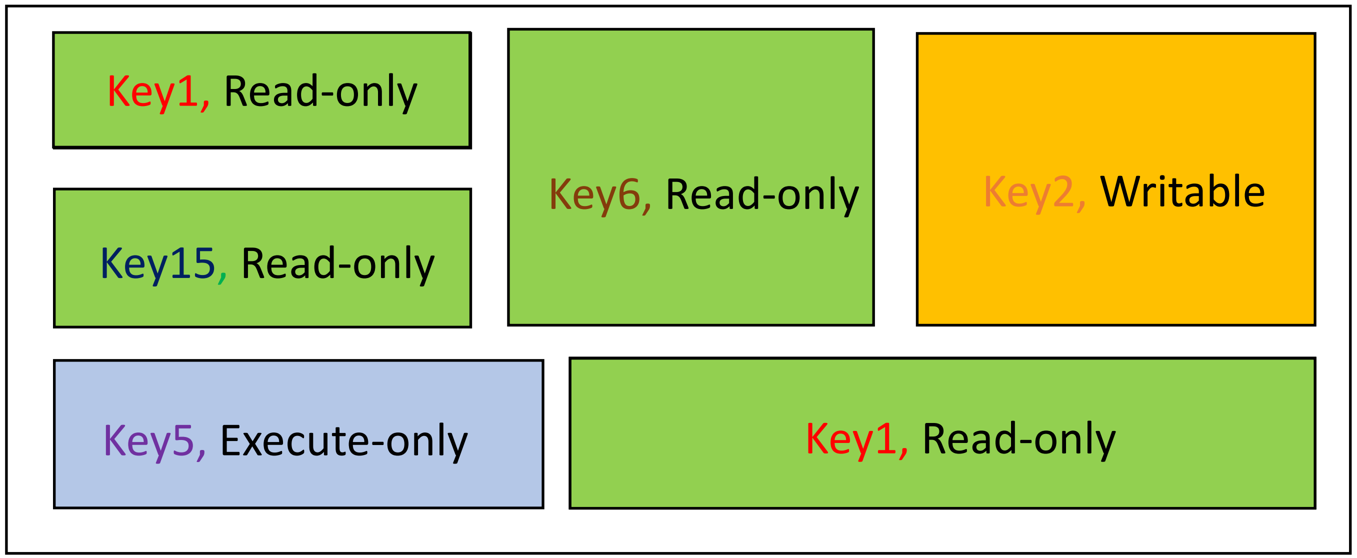
Bytecode Interpreter Attack



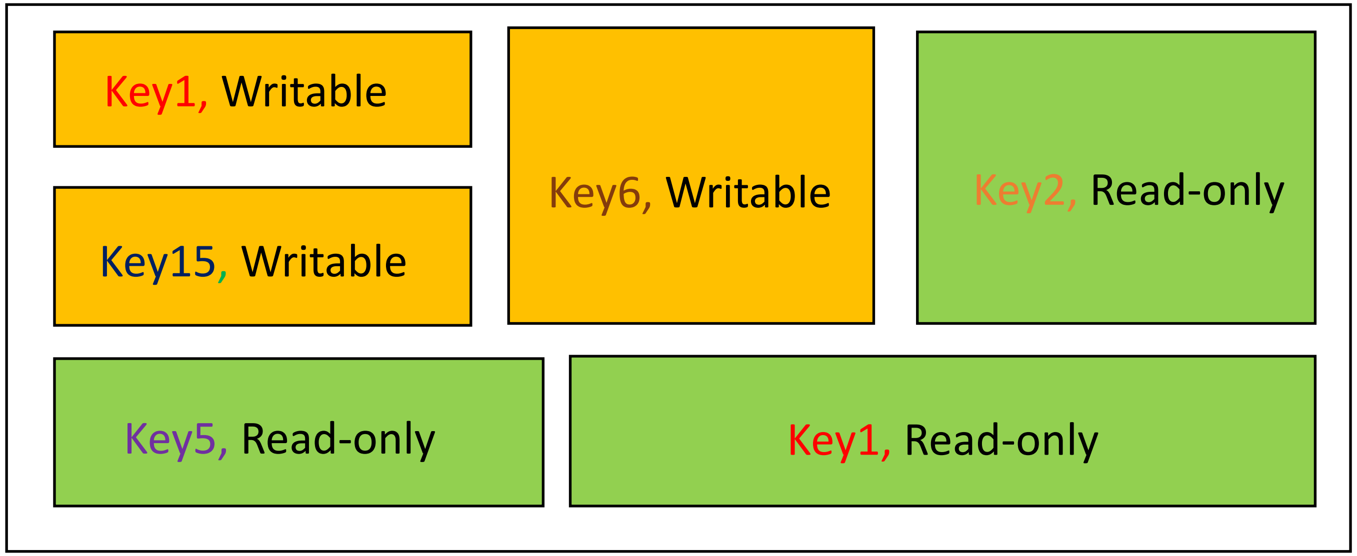
NoJITsu

- 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



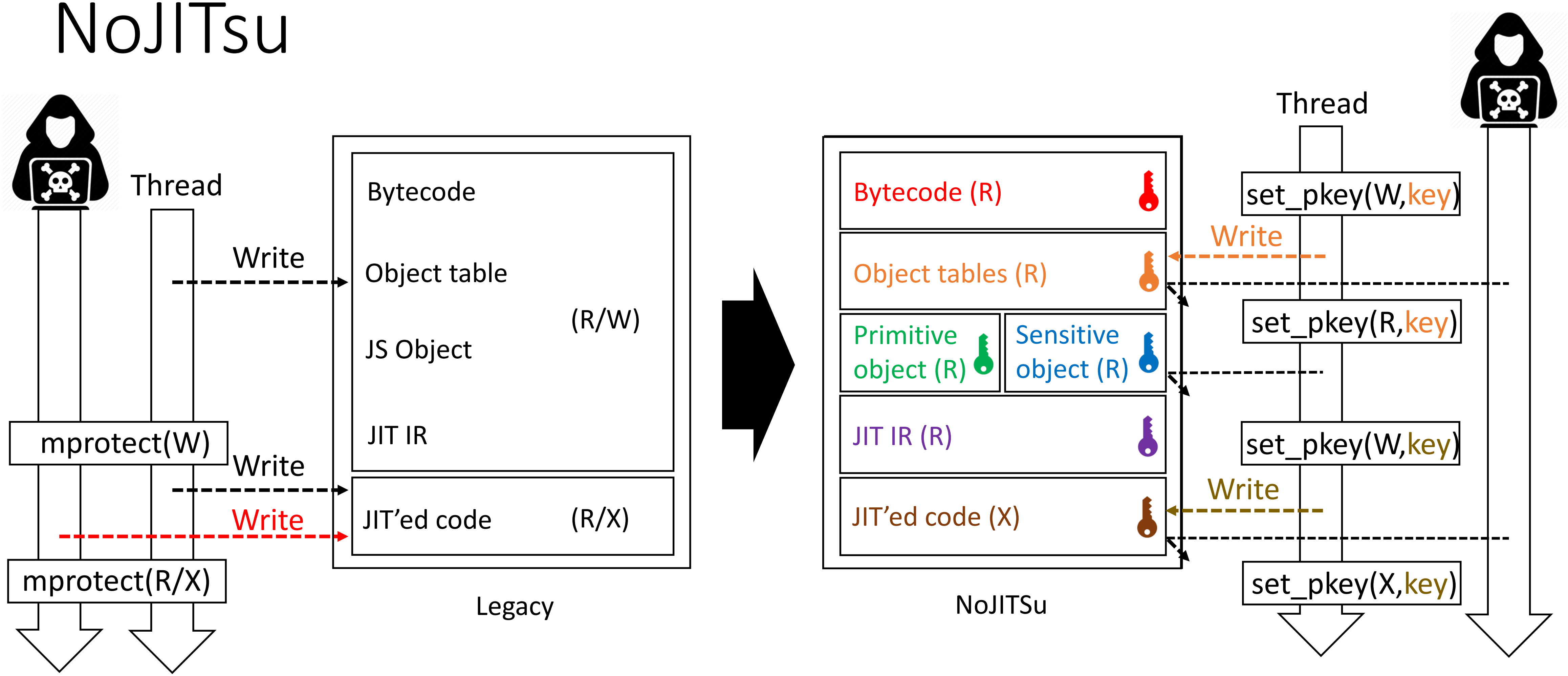


Memory (Thread1)



Memory (Thread2)

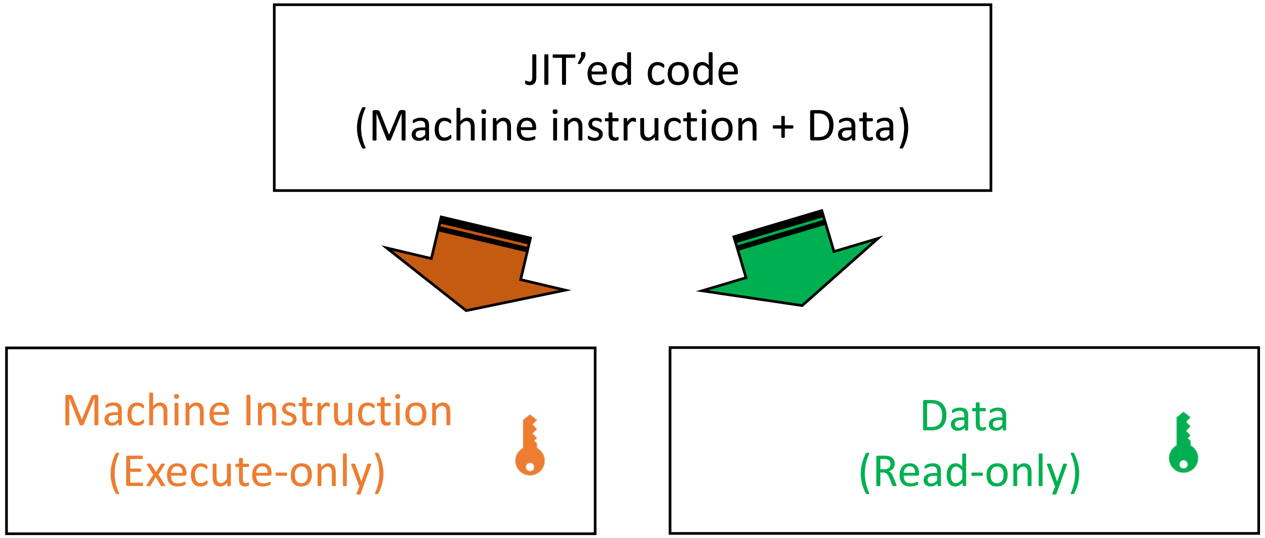
NoJITsu



- 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



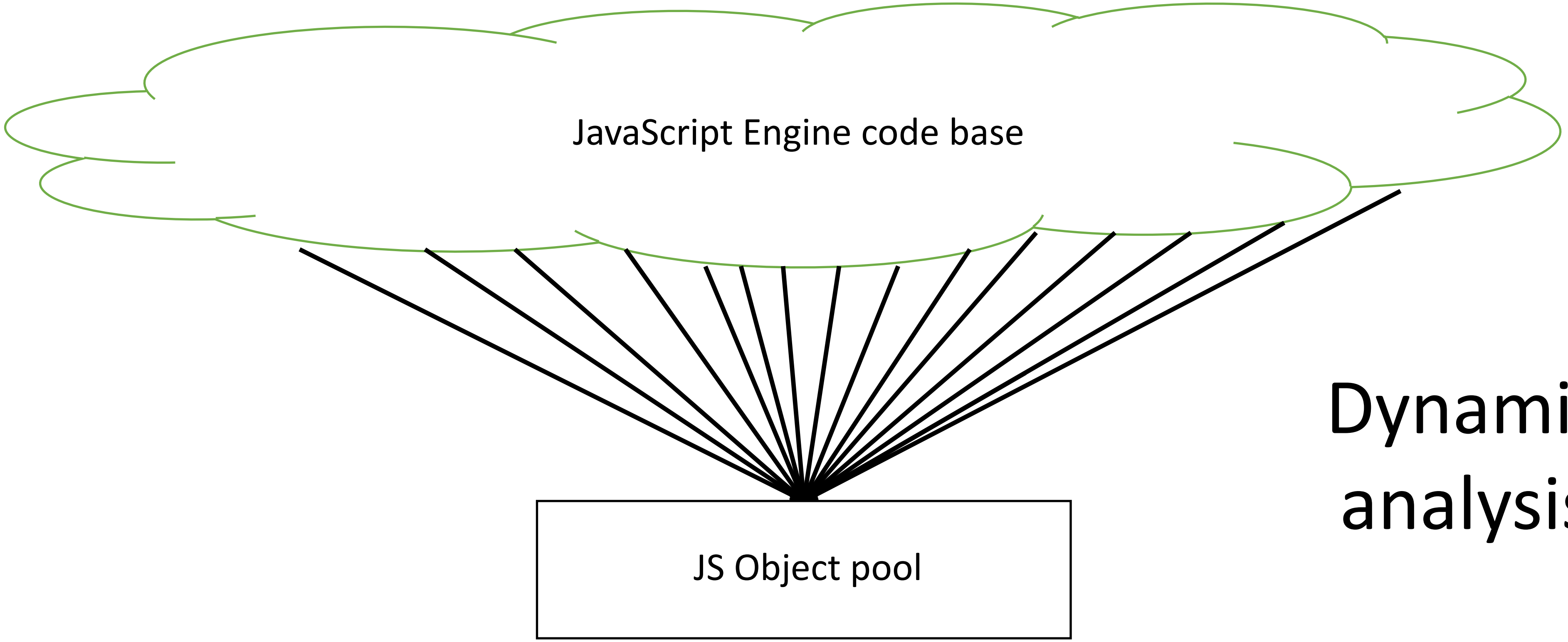

```
Compile_bytecode()
{
    .....
    .....
    saved_pkru = set_pkru(W, key_bytecode)
    write bytecode
    recover_pkru(saved_pkru)
    .....
    .....
}
```

JavaScript Object

- There are a huge number of write access instructions to JS object throughout JS code base.

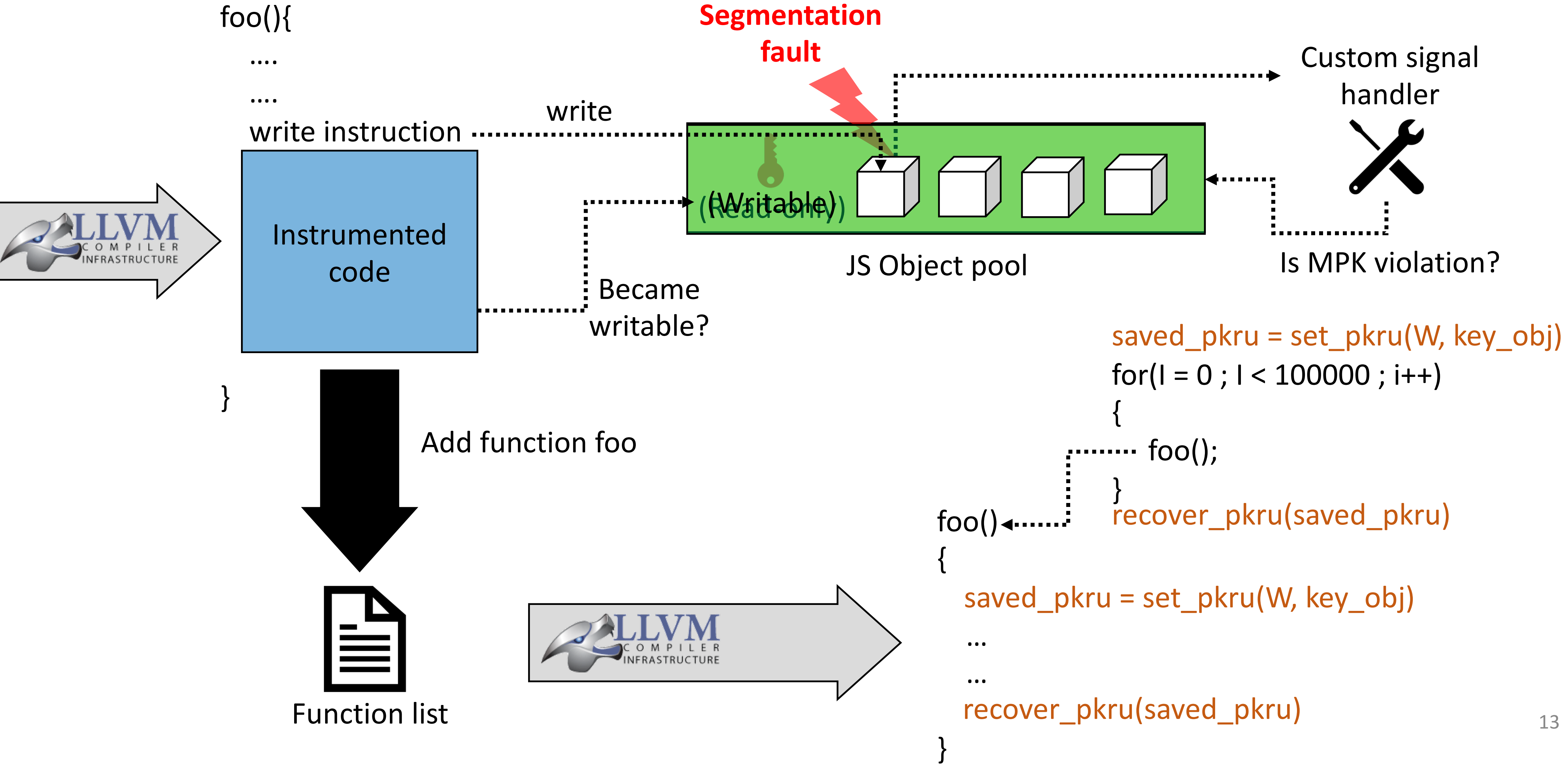
Manual effort?

Static analysis?

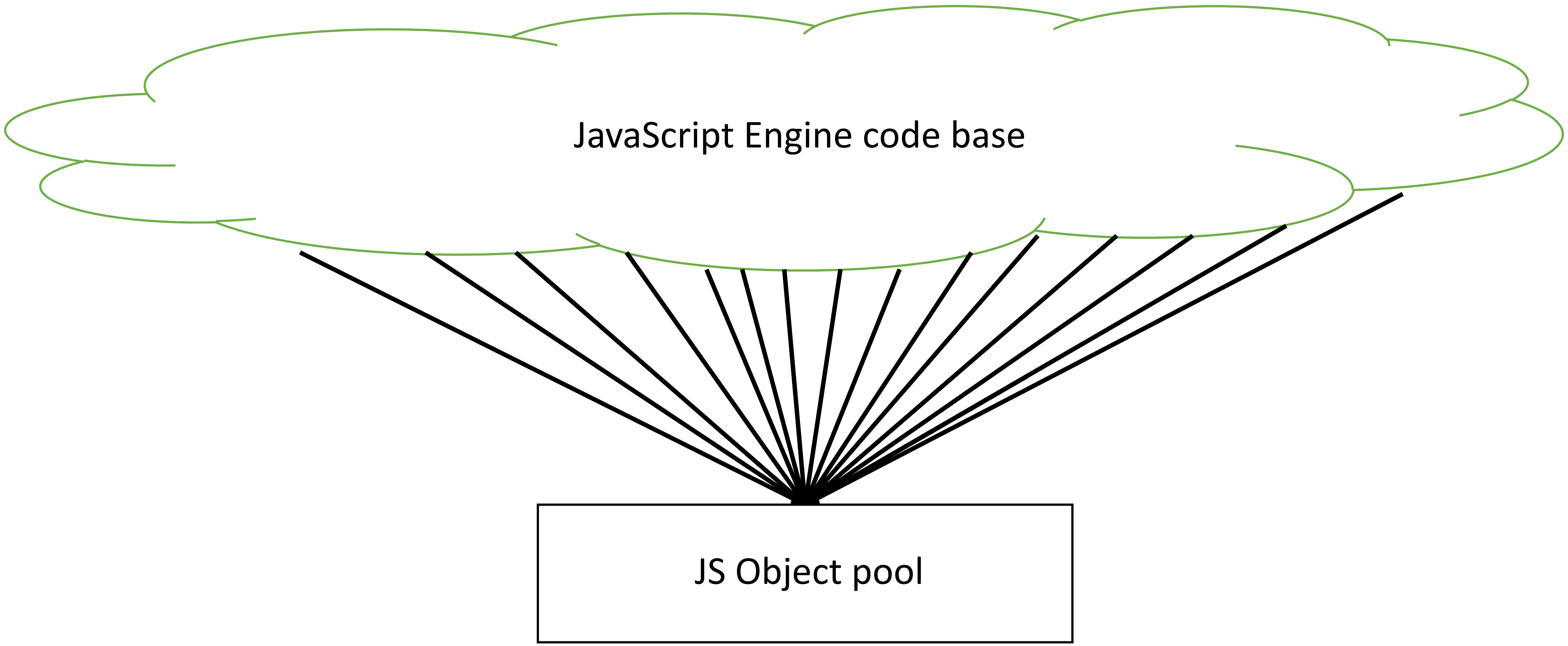


Dynamic analysis

Dynamic Analysis

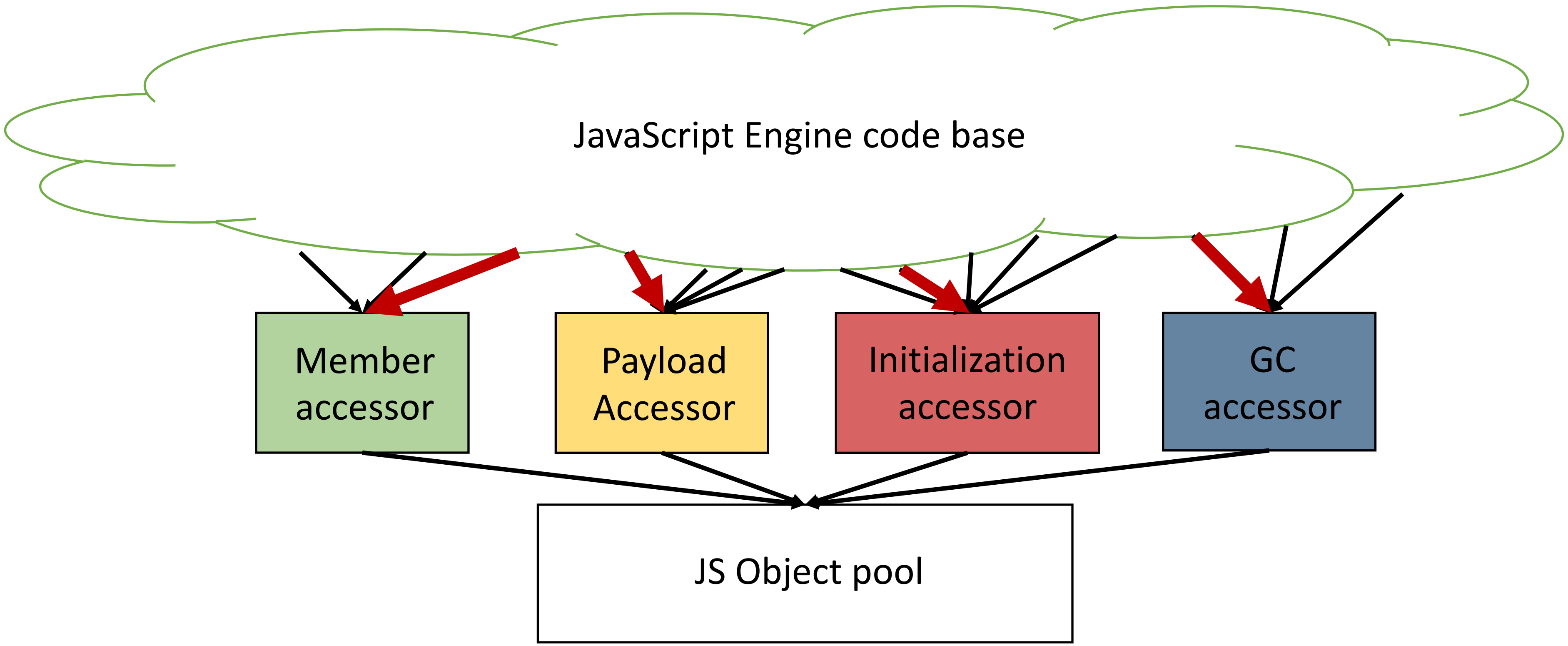


Dynamic Analysis – Input Set



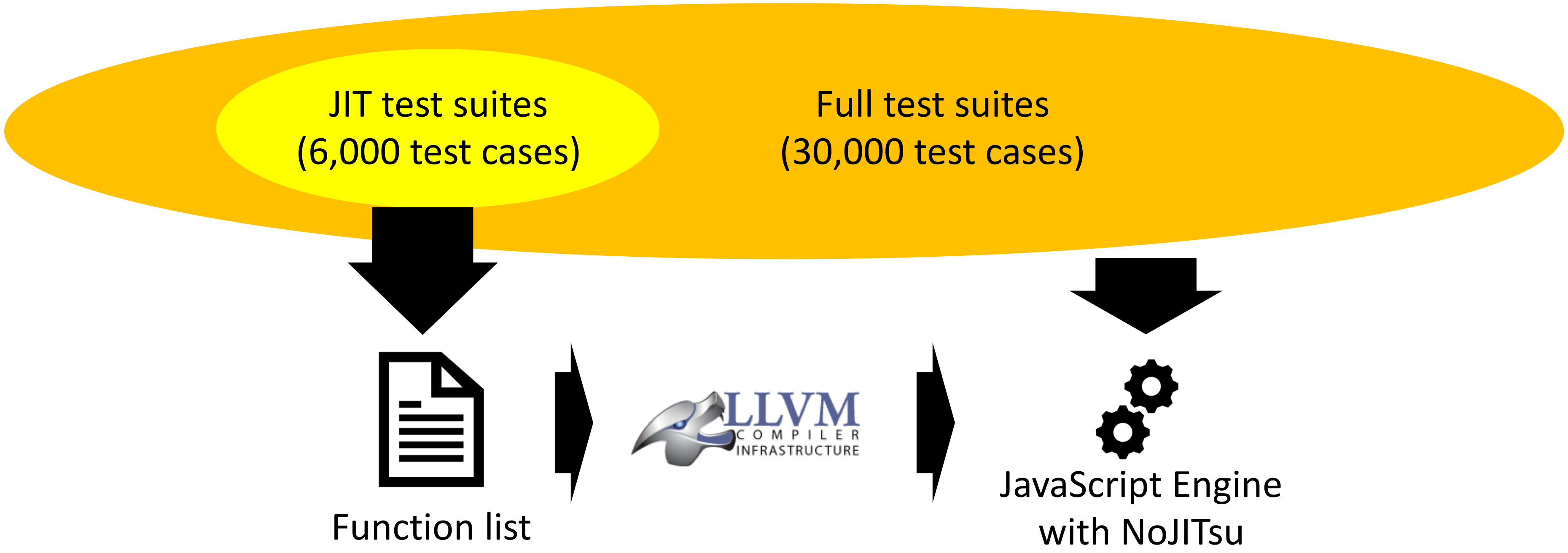
Dynamic Analysis – Input Set

- 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



Evaluation

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

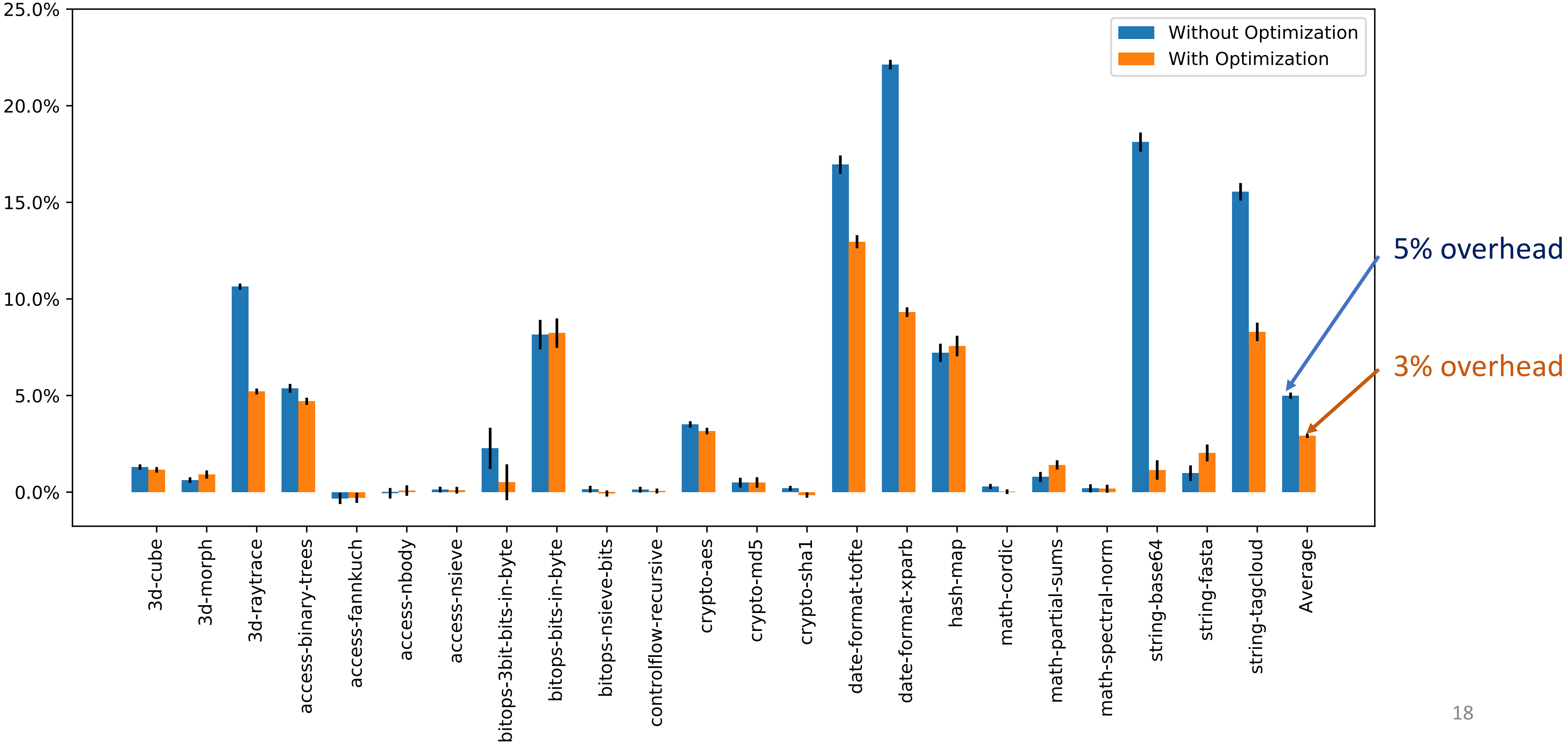


- Code-Reuse attack and bytecode interpreter attack
  - Successfully stop JIT-ROP and our bytecode interpreter attack

Evaluation

- Performance
  - LongSpider benchmarks
  - Intel Xeon silver 4112 machine under Ubuntu 18.04.1 LTS

Evaluation



25.0%
20.0%
15.0%
10.0%
5.0%
0.0%
Without Optimization
With Optimization
3d-cube
3d-morph
3d-raytrace
access-binary-trees
access-fannkuch
access-nbody
access-nsieve
bitops-3bit-bits-in-byte
bitops-bits-in-byte
bitops-nsieve-bits
controlflow-recursive
crypto-aes
crypto-md5
crypto-sha1
date-format-tofte
date-format-xparb
hash-map
math-cordic
math-partial-sums
math-spectral-norm
string-base64
string-fasta
string-tagcloud
Average
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 attack and our bytecode interpreter attack on JS engines with a moderate overhead

Thank You

Q&A

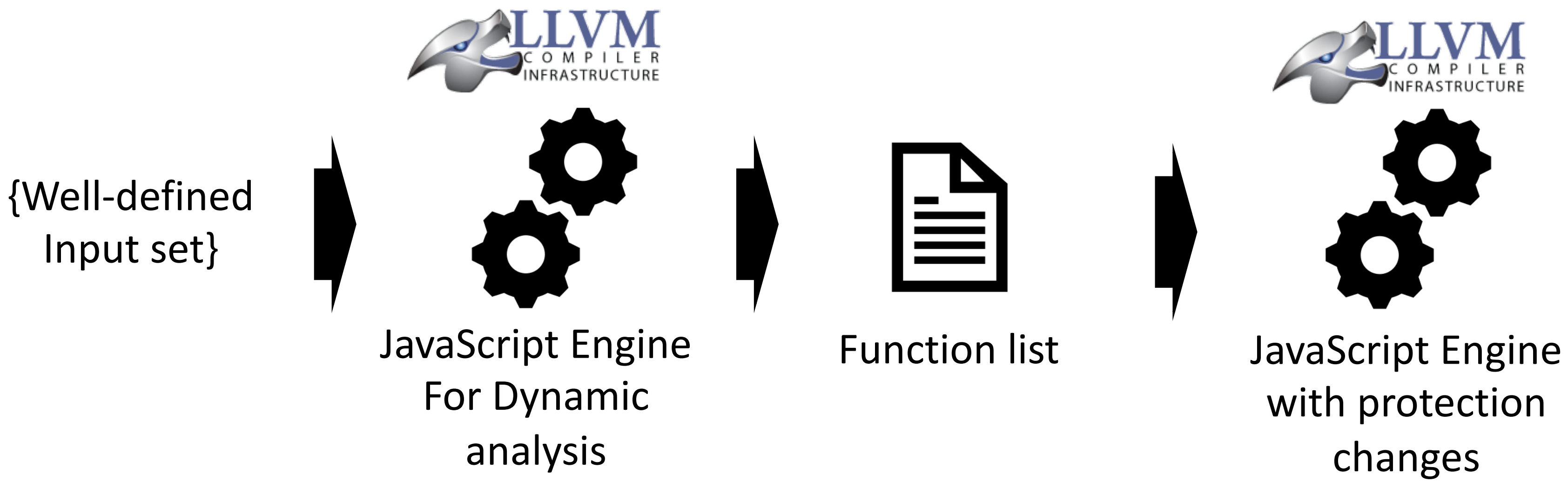
Performance Optimization

- Hoist protections out of loops

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

```
foo()
{
    saved_pkru = set_pkru(W, key_bytecode)
    …
    …
    recover_pkru(saved_pkru)
}
```

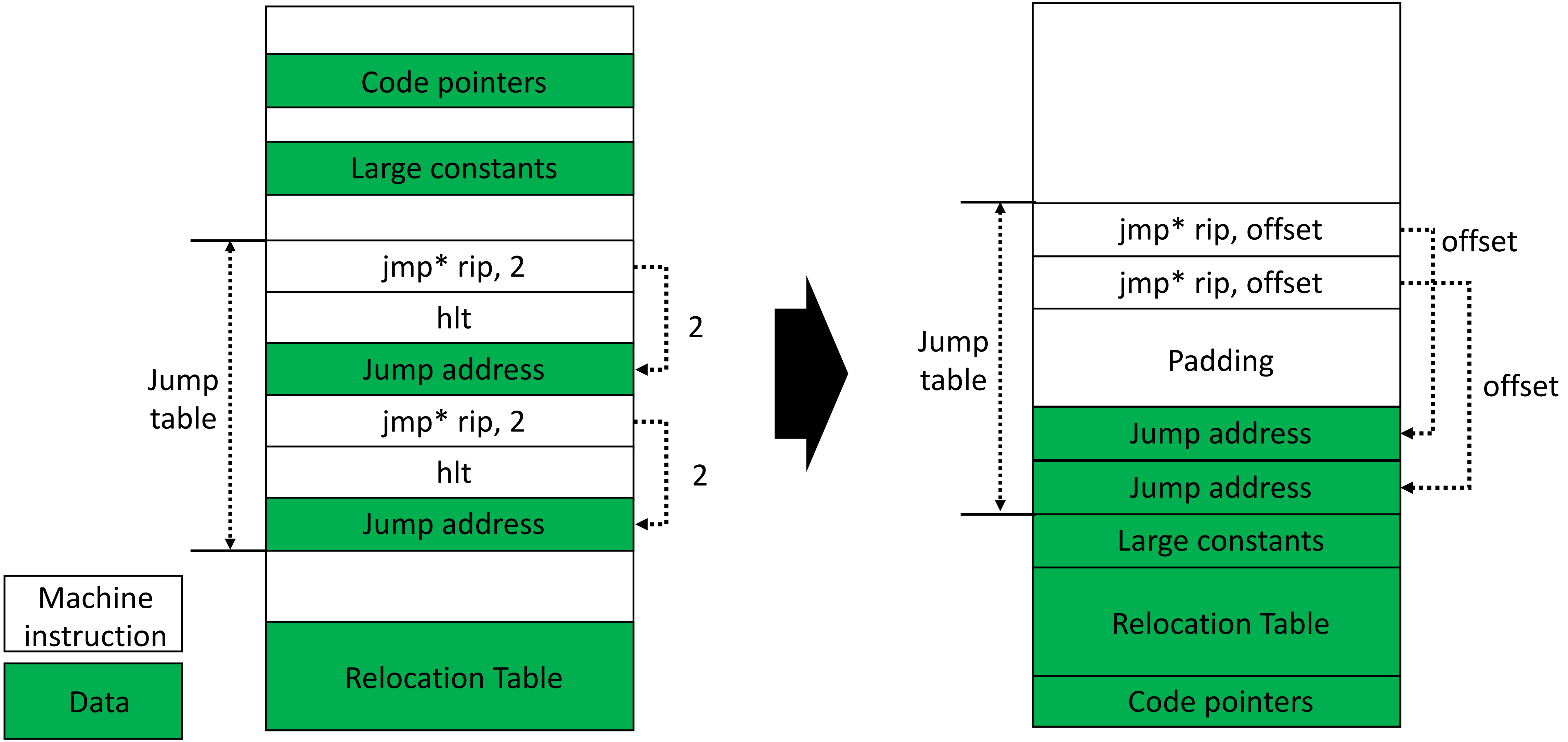
Dynamic Analysis



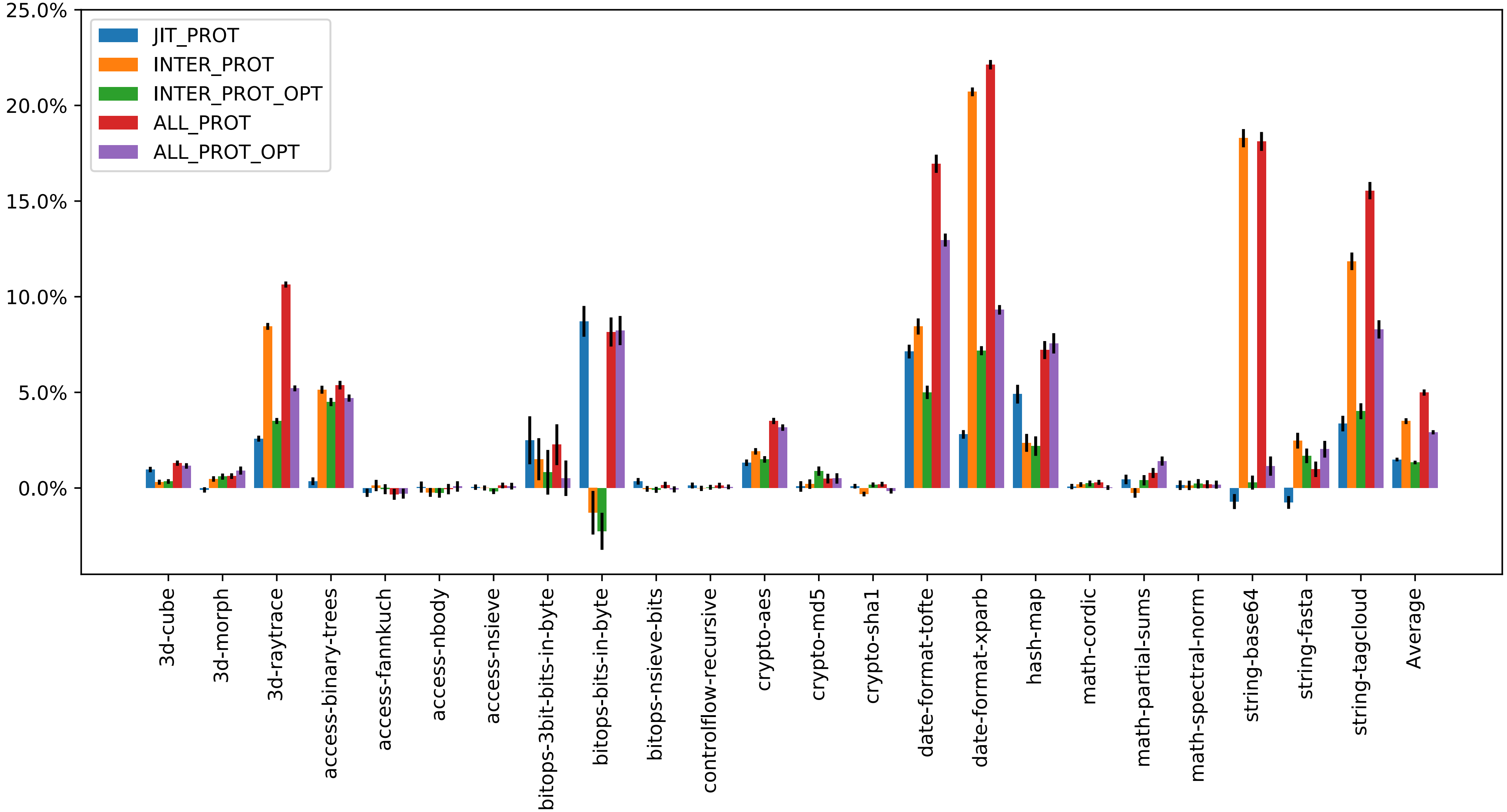
What is the well-defined input set?

```
foo()
{
   saved_pkru = set_pkru(W, key_bytecode)
   …
   …
   recover_pkru(saved_pkru)
}
```

Machine Code and Data Separation



Evaluation



25.0%
20.0%
15.0%
10.0%
5.0%
0.0%
JIT_PROT
INTER_PROT
INTER_PROT_OPT
ALL_PROT
ALL_PROT_OPT
3d-cube
3d-morph
3d-raytrace
access-binary-trees
access-fannkuch
access-nbody
access-nsieve
bitops-3bit-bits-in-byte
bitops-bits-in-byte
bitops-nsieve-bits
controlflow-recursive
crypto-aes
crypto-md5
crypto-sha1
date-format-tofte
date-format-xparb
hash-map
math-cordic
math-partial-sums
math-spectral-norm
string-base64
string-fasta
string-tagcloud
Average