GraalVM™

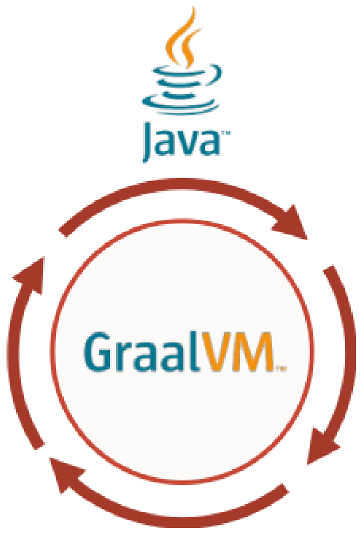
Compilers & Security

Gergö Barany
Matthias Neugschwandtner
Oracle Labs Austria

What is GraalVM?

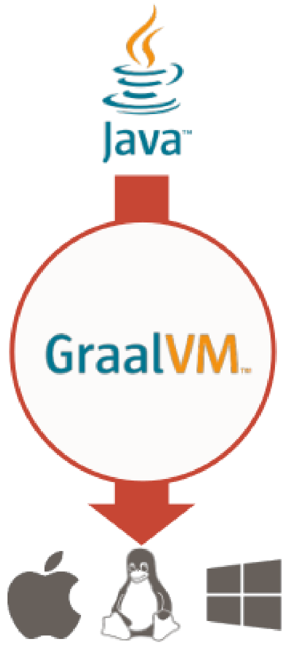
GraalVM: High-performance polyglot virtual machine

High-performance optimizing Just-in-Time (JIT) compiler



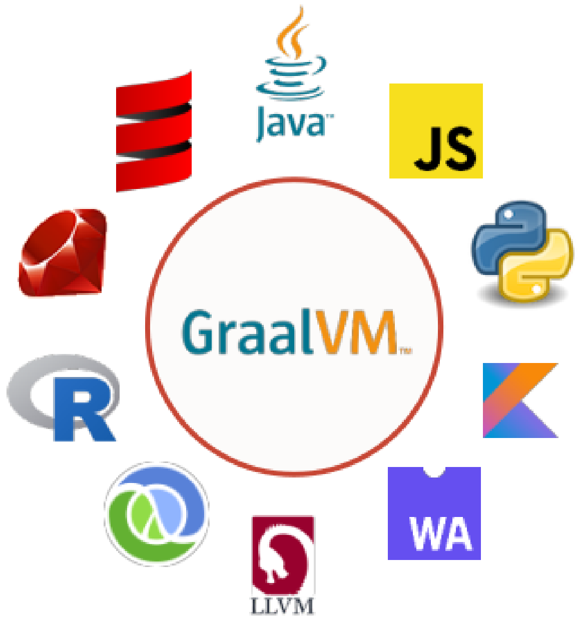
```
$ java -jar MyApplication.jar
```

Ahead-of-Time (AOT) "Native Image" generator



```
$ native-image MyApplication.jar
$ ./my-application
```

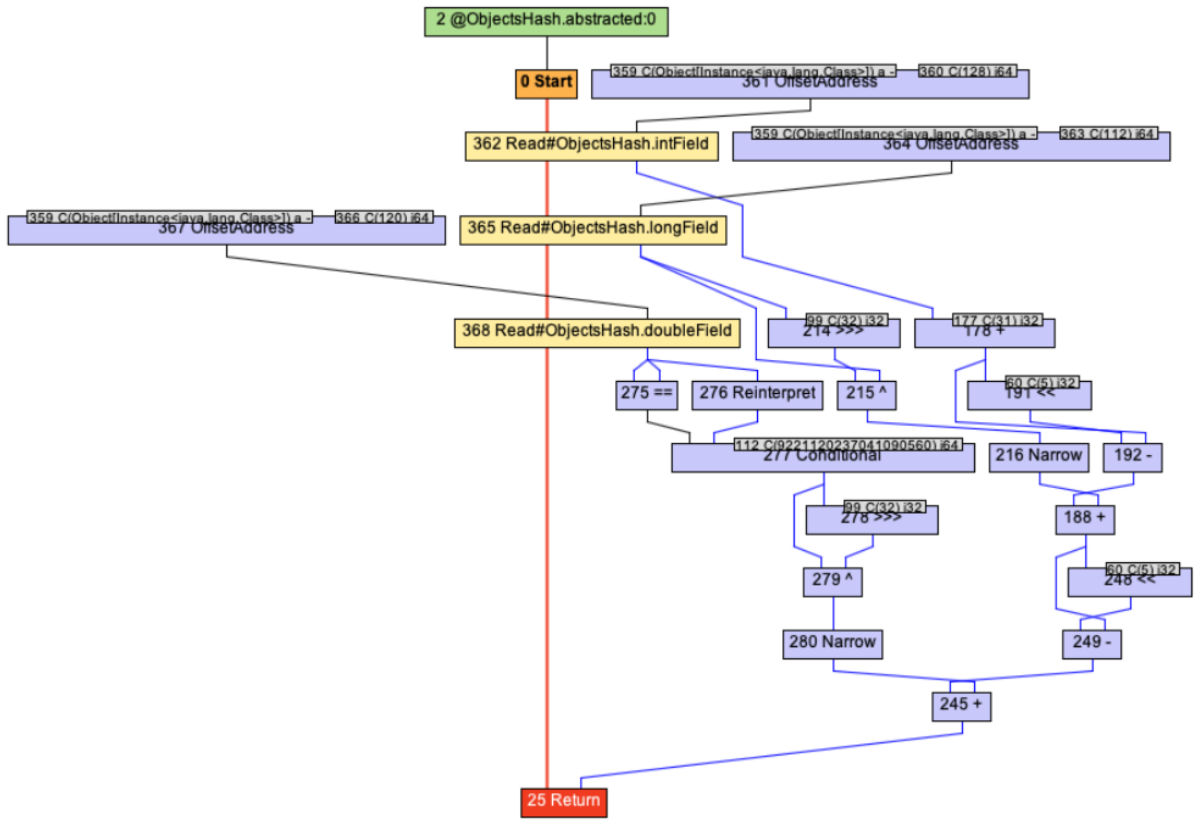
Multi-language support



e.g., polyglot plugins, JavaScript stored procedures in Oracle DB

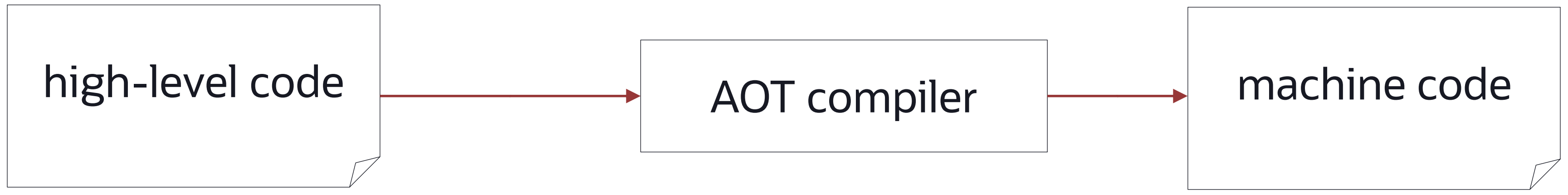
The GraalVM compiler

- all GraalVM technologies use one compiler
- written in Java
- "Sea of Nodes" representation:
  - some fixed nodes for strict control flow
  - most nodes "float" without fixed order
- many innovative optimizations



Compiler Security Aspects

Threat model – AOT compilation



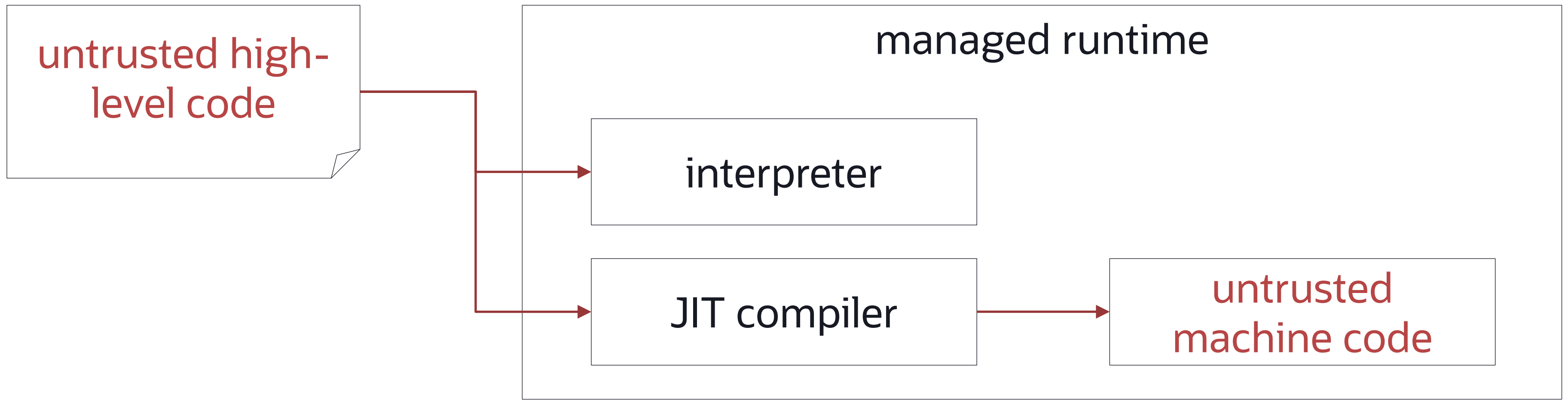
Threats:

- Correctness – bug in lowering the high-level code affects the implementation of a security mechanism
  - example: access control logic written in a high-level language

- Language level security mechanisms are not correctly implemented by the compiler
  - example: missing bounds check on array access

- Unforeseen properties when lowering
  - example: constant time cryptography

Threat model – JIT compilation



Adding a JIT compiler to a managed runtime increases the attack surface when processing untrusted code. It allows an attacker to

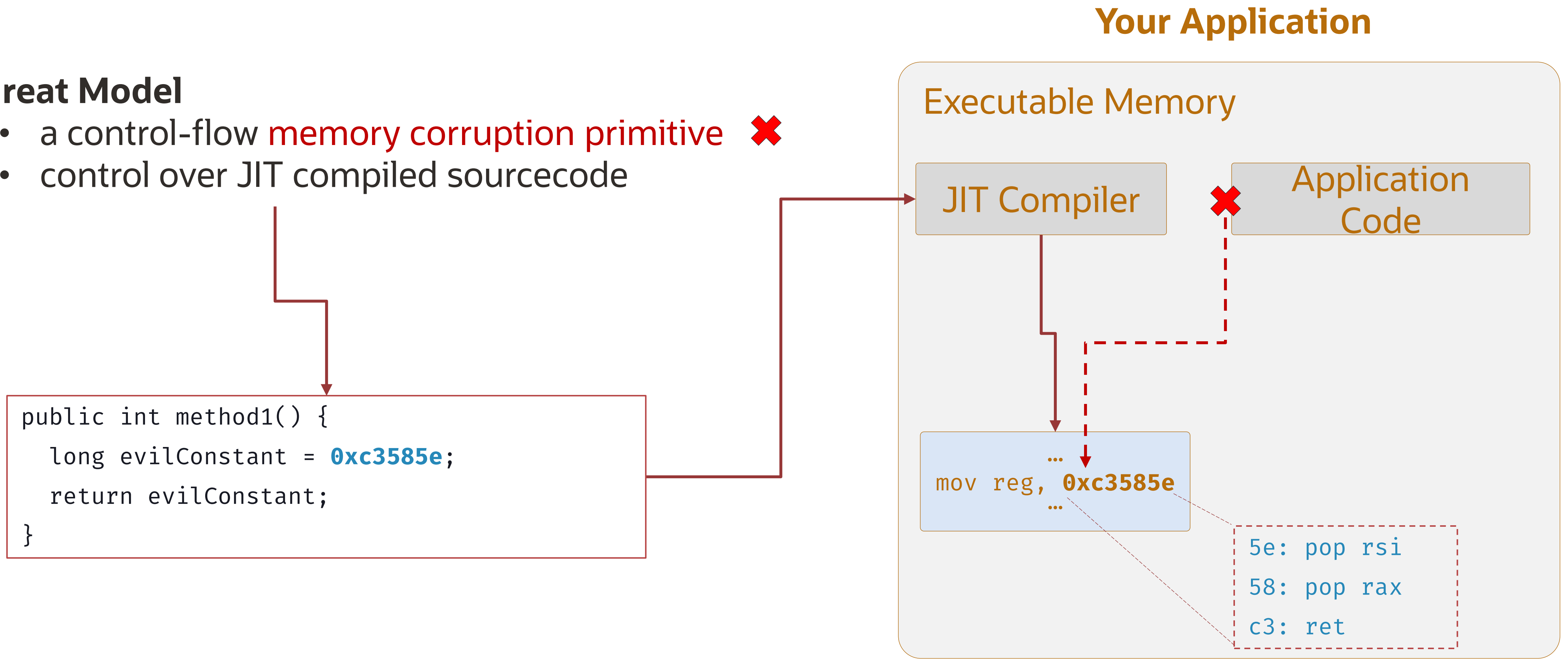
- place untrusted machine code in the address space
  - JIT spray attacks
  - speculative execution side channel attacks
- exploit bugs in the compiler implementation
  - less likely

JIT Spraying

Threat Model

- a control-flow memory corruption primitive ✖
- control over JIT compiled sourcecode

```
public int method1() {
  long evilConstant = 0xc3585e;
  return evilConstant;
}
```

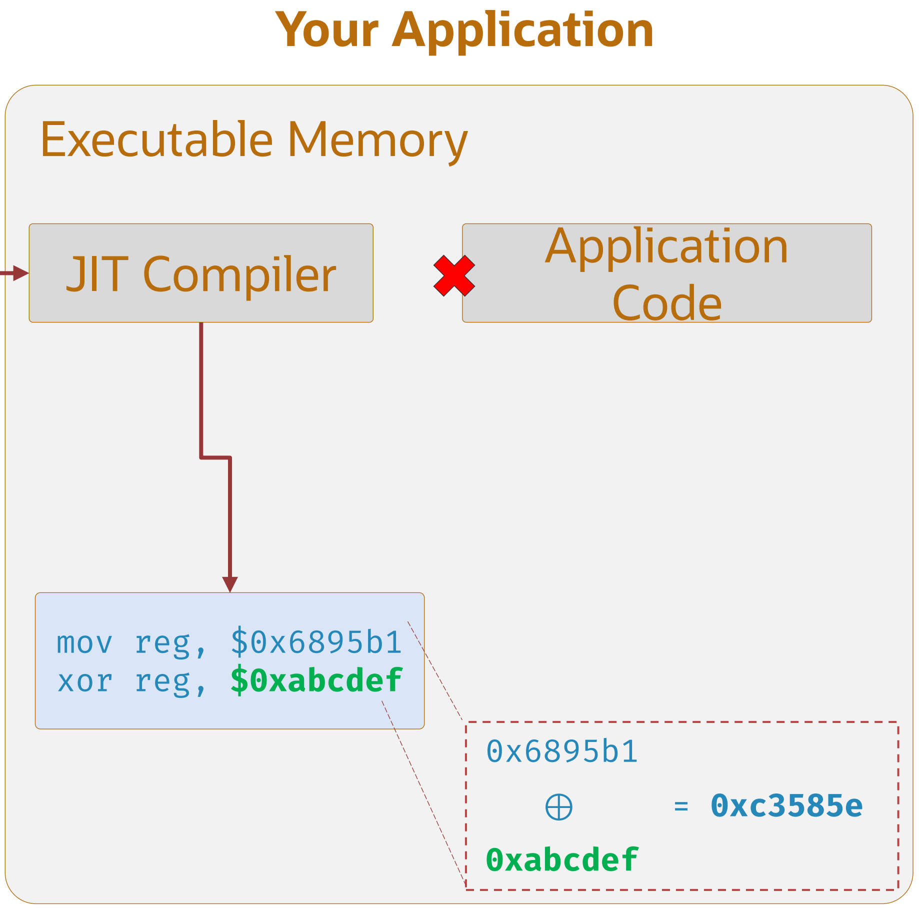


Constant Blinding

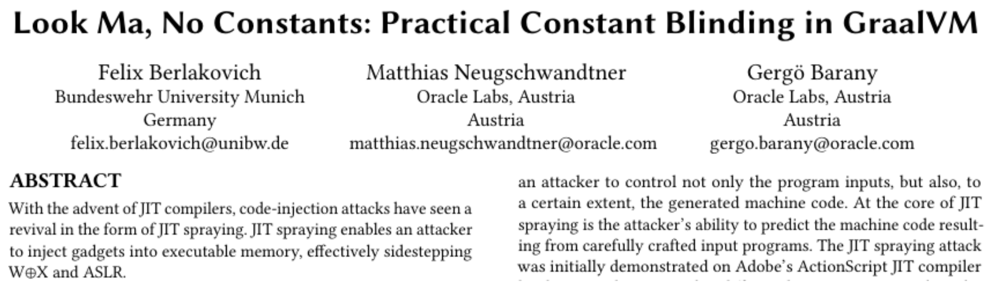
The Principle

- encrypt each constant with a **random key**
- decrypt at runtime

```
public int method1() {
  long evilConstant = 0xc3585e;
  return evilConstant;
}
```



```
mov reg, $0x6895b1
xor reg, $0xabcdef
```



Look Ma, No Constants: Practical Constant Blinding in GraalVM

Felix Berlakovich
Bundeswehr University Munich
Germany
felix.berlakovich@unibw.de

Matthias Neugschwandtner
Oracle Labs, Austria
Austria
matthias.neugschwandtner@oracle.com

Gergö Barany
Oracle Labs, Austria
Austria
gergo.barany@oracle.com

ABSTRACT

With the advent of JIT compilers, code-injection attacks have seen a revival in the form of JIT spraying. JIT spraying enables an attacker to inject gadgets into executable memory, effectively sidestepping W⊕X and ASLR.

an attacker to control not only the program inputs, but also, to a certain extent, the generated machine code. At the core of JIT spraying is the attacker's ability to predict the machine code resulting from carefully crafted input programs. The JIT spraying attack was initially demonstrated on Adobe's ActionScript JIT compiler

Entry Point Randomization

- Control flow hijacking attacks often rely on a predictable code layout
  - where in memory do I point my hijacked code pointer to?

- “Cheap” mitigation: randomize offset of JIT compiled code in memory
  - prefix function entry point with a random number of trap instructions
  - probabilistic defense

```
int3
int3
int3
sub     rsp,0x38
mov     rdi,0xe0
add     rdi,QWORD PTR [r15+0x8]
mov     eax,DWORD PTR [rip+0xfe8]
lea     rax,[r14+rax*8]
```

Control Flow Integrity

- Prevent attacker from redirecting control flow to arbitrary addresses

- Control Flow Integrity (CFI) “enforces” the intended control flow graph
  - ties indirect branches (jumps, calls) to their intended targets

- Coarse-grained CFI ties indirect branches to landing pads (single target class)

callsite

```
…
mov $rax, <target_addr>
cmp [$rax], ENDBR64
je succeed
int3
succeed:
call $rax
```

target

```
endbr64
…
```

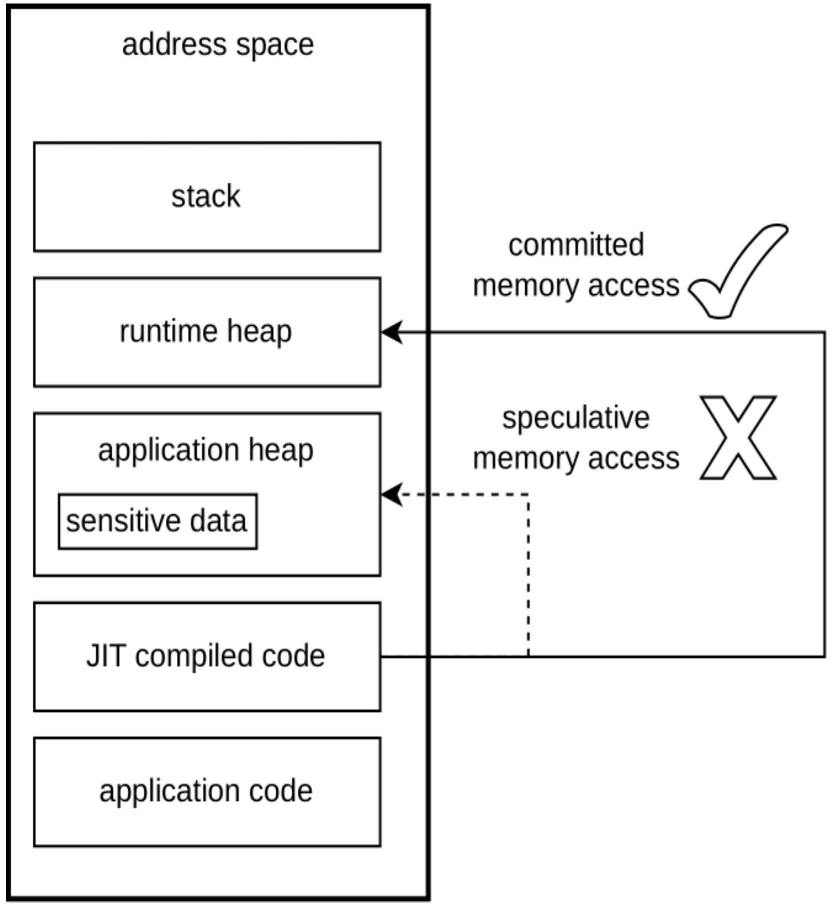
- Finer-grained CFI ties indirect branches (individual target classes)
  - requires precise CFG

Spectre – Speculative Execution Side Channels

Spectre allows an attacker to leak arbitrary memory from the same address space.

The attack requires three primitives:

- Trigger speculative execution on an architecturally infeasible execution path
  - e.g., incorrect branch prediction
- Target a memory location that contains sensitive information
- Side channel to infer the information read
  - timing information + a shared resource, e.g., cache or CPU's execution ports



Spectre – Compiler Mitigations

- Prevent triggering speculative execution on an architecturally infeasible execution path
  → insert speculation barrier instructions on critical branches

- Speculation barriers have a significant impact on performance → be smart about where to place them. Strategies:
  - Every basic block. Comprehensive, but high runtime overhead
  - Check if the target basic block is guarded by a bounds check
  - Check if the target basic block deoptimizes

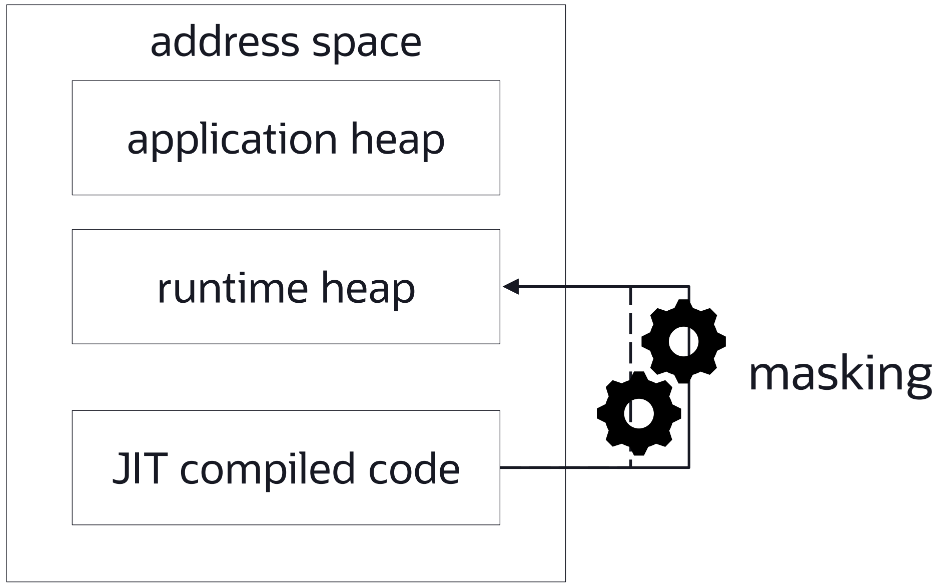
Spectre – Compiler Mitigations

- Prevent targeting a sensitive memory location – “safe” speculative execution
  - force all memory accesses to a given address range
  - use software fault isolation techniques such as masking that are effective during speculative execution

- GraalVM can use compressed references
  - object references are base + offset
  - the base register is already set to `r14`

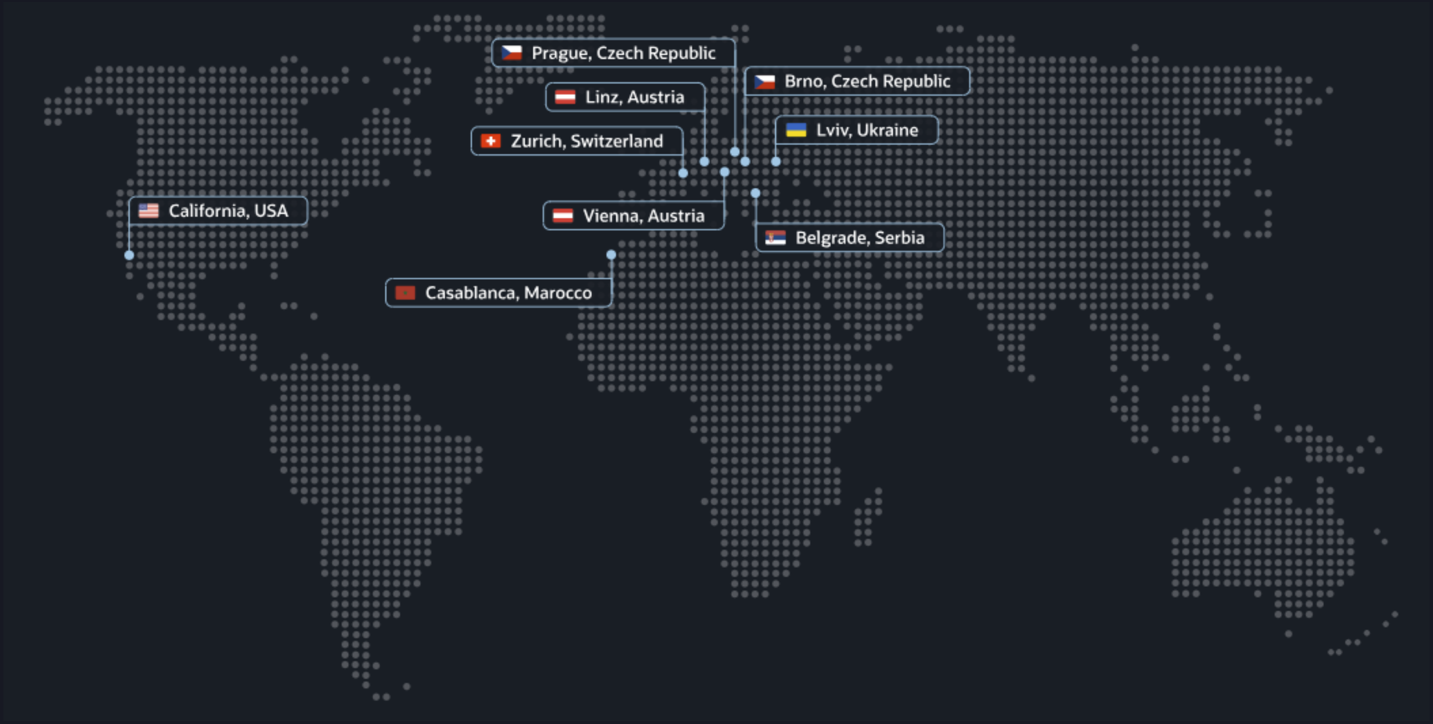
- Masking the memory access:

```
lea  reg, [index * scale + displacement]
mov  reg2, MASK
and  reg2, reg
mov  reg, [r14 + reg2]
```



GraalVM Internship Program

- Check out https://www.graalvm.org/community/internship/
- Application period closed, but we remain open to skilled candidates with a background in compilers.
- Send your CV to graalvm-internships_ww_grp@oracle.com


Prague, Czech Republic
Brno, Czech Republic
Linz, Austria
Lviv, Ukraine
Zurich, Switzerland
California, USA
Vienna, Austria
Belgrade, Serbia
Casablanca, Marocco