NestedVM

Brian Alliet
Adam Megacz

Safe and Unsafe Languages

- Unsafe Languages: C, C++
- Safe Language: Java, C#,
- Safe languages don’t allow access to arbitrary memory locations, execution of arbitrary code, etc

Problem

- Large body of software written in unsafe languages such as C and C++
- Safe languages such as Java don’t have comparable alternatives to much of the C and C++ software available

Real-World Examples

- libjpeg (jpeg decoder)
- FreeType (True Type font renderer)
- libmspack (cab file decomposer)
- TeX (Knuth's typesetting system)
- Jay (yacc for Java, but you knew that)

Typical Solution - JNI

- Insecure
  - Requires executing unsafe code
  - Prohibited in many environments (such as Applets)
- Not Portable
  - Binary required for each target platform
  - Typically not known at deployment time

Potential Solutions

- Jazillian, c2j, Cappuccino, moCHA-java
- Convert C and C++ source code to Java source code
- Don’t support the entire C or C++ language
  - pointer arithmetic, casting between unrelated types, unions, etc

NestedVM

- Binary translation, not source code
- Converts unsafe machine code (rather than unsafe source code) to safe source code or bytecode
- Several key advantages

Language Agnostic

- Conversion to machine code is handled by existing compilers
- C and C++ specifications are complex
  - Supporting them best left to GCC
- ANY language can be supported, not just C and C++

Bug-for-bug compatibility

- Code that is dependent on quirks, bugs, or “features” of existing compilers works
- The compiler’s output is NestedVM’s input
- We don’t care what goes on inside the compiler

No build process modification

- The same compiler that produces native code produces the input for NestedVM
- Existing Makefiles, header files, complex preprocessor usage, weird compiler options are all supported
- Again, we don't care what goes on inside the compiler and build process

No post translation human intervention

- Entire input language is supported
- No need to tweak the output to get a working program
- Unsafe source can be modified, or upgraded without needing to make changes to the safe output

NestedVM Overview

Unsafe Source
Code
(.c)

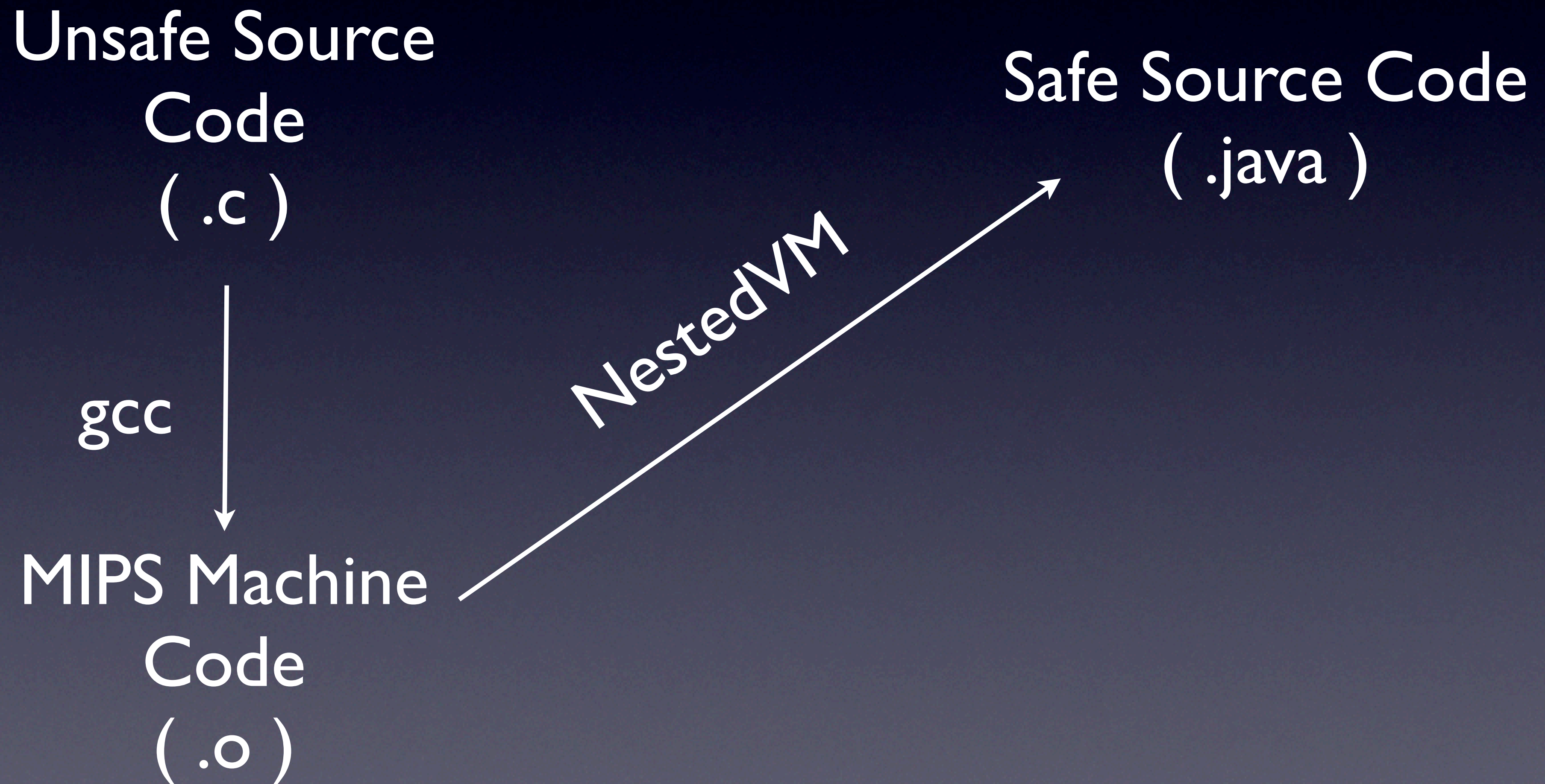
NestedVM Overview

Unsafe Source Code (.c)

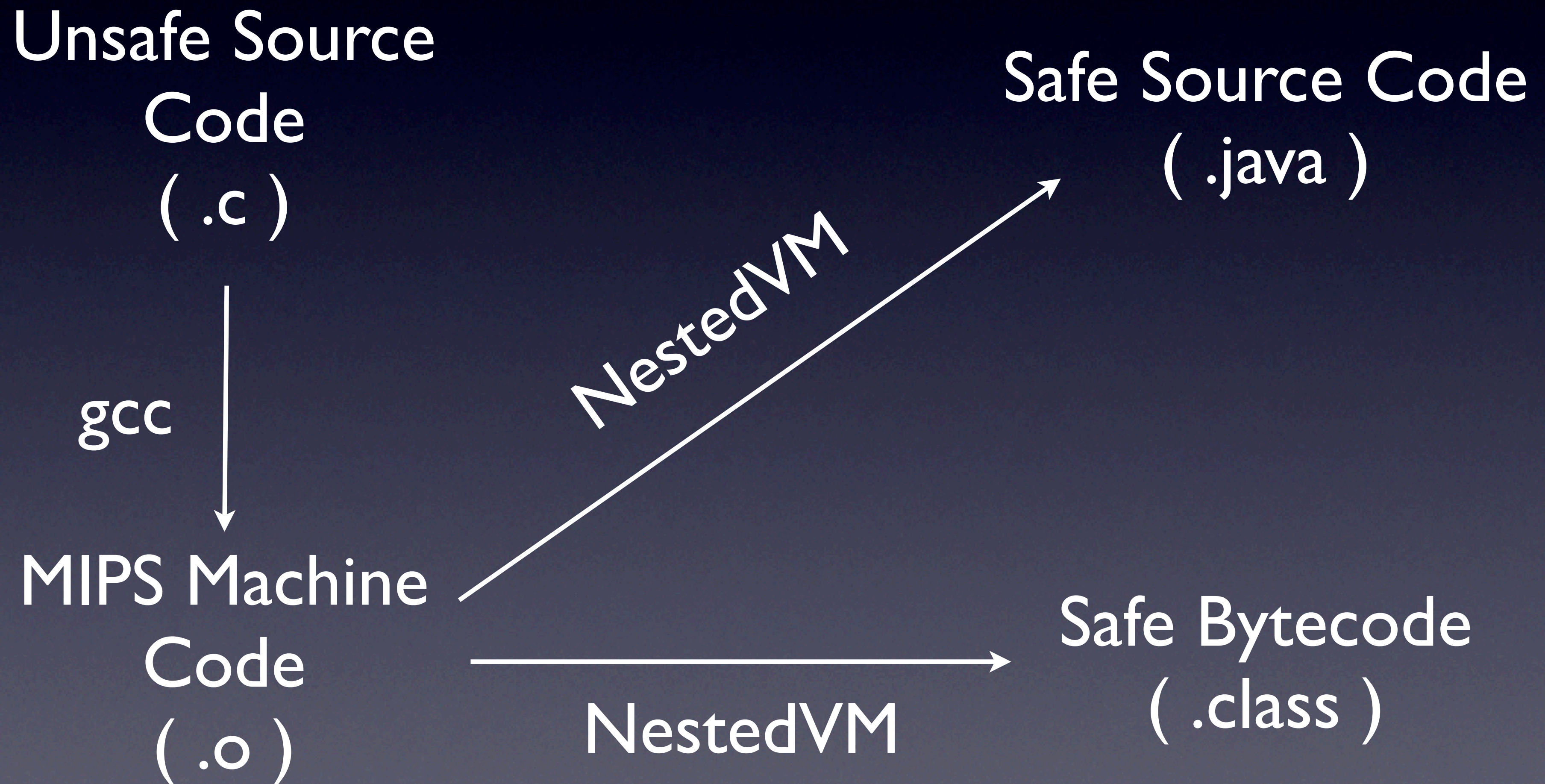
gcc

MIPS Machine Code (.o)

NestedVM Overview



NestedVM Overview



Why MIPS

- MIPS ISA is much simpler than x86, powerpc, and other popular architectures
- MIPS has been around a long time
  - Well supported by many tools, most importantly, GCC
- Most MIPS instructions map well to JVM bytecodes

MIPS Example

```
fact:
move t0, a0    ; load argument (n) in t0
bgtz t0, L0    ; branch if n > 0
li v0, 1       ; load 1 in register v0
jr ra          ; return
L0:
subu a0, t0, 1  ; subtract 1 from n
jal fact        ; call fact on n-1
mult v0, v0, t0 ; multiply
jr ra           ; return
```

Mapping MIPS to the JVM

- Code translation
- Memory representation
- System calls

Code Translation

```
public class Foo {
  int pc;
  int v0, v1, a0, t0, ...;
  void run() {
    for(;;) switch(pc) {
      case 0x1000: t0 = a0; // move t0 a0
      // bgtz t0, L0
      case 0x1004: if(t0 > 0) { pc = 0x1010; break; }
      case 0x1008: v0 = 1; // li v0, 1
      case 0x100c: { pc = ra; break; } // jr ra
/*L0*/case 0x1010: a0 = t0 - 1; // subu a0, t0, 1
      // jal fact
      case 0x1014: { ra = 0x1018;pc = 0x1000; break; }
      case 0x1018: v0 = v0 * t0;
      case 0x101c: { pc = ra; break; } // jr ra
} } }
```

Memory Representation

- Entire virtual memory space mapped to a giant int[][] array indexed by page, then address
- Breaking memory up by pages allows memory to be dynamically allocated
- Example memory read:

```
// lw v0, 20(sp) ; load 4 byte word at sp+20
v0 = memory[(sp+20)>>>16][(sp+20)&0xffff];
```

What we have so far

- Applying the translation techniques we can emulate an entire MIPS machine
- Memory can be read, manipulated, and written
- This is essentially all computers do, and this is enough to be useful

Example: Decoding a JPEG

- Write the compressed JPEG to some predefined memory location (System.arraycopy to the memory array)
- Run some decompression code
- Read the uncompressed JPEG from some predefined memory location

The next step

- Fiddling with bits in memory, while useful, won't allow many real world programs to work "out of the box"
- There is no way to interact with the world outside the CPU/Memory

Syscalls

- Applications interact with the Operating System via the SYSCALL instruction
- When a syscall instruction is executed control is transfered to the kernel (part of the operating system)
- The kernel decides what to do, does it, fiddles with the applications memory to get the results to it, and returns to the app

Syscall Example

```
// the signature for the write syscall (in C)
// write(int fd, char *buf, int len);

string: .asciiz “hello world\n”
li v0, SYS_write ; put the syscall number into v0
li a0, STDOUT    ; file descriptor number (int fd)
la a1, string    ; addr of hello world (char *buf)
li a2, 12        ; length of hello world (int len)
syscall
```

The NestedVM Runtime

- This is the NestedVM “kernel”
- It interacts with the outside world on behalf of the application (just like a real kernel)
- Maintains all the per process state information usually maintained by the kernel
  - File descriptor table, working directory, pid, etc

Accessing the Runtime

- All applications inherit from the Runtime class
- Individual applications essentially sit on top of a Runtime instance
- The syscall instruction is simply mapped to the syscall() method of the runtime class

```
v0 = syscall(v0,a0,a1,a2,a3); // syscall
```

Implementing the Runtime

- The syscall() method dispatches to individual Runtime methods based on the syscall number

```
int syscall(int sc, int a, int b, int c, int d) {
  switch(sc) {
      case SYS_write: return sys_write(a,b,c,d);
      case SYS_open: return sys_open(a,b,c,d);
      ....
  }
}
```

The Write Syscall

```
int sys_write(int fd, int addr, int len) {
  FD fd = fileDescriptors[fd];
  byte[] buf = new byte[len];
  // copy the bytes of “user memory” at addr
  // to the “kernel memory” in buf (a byte[] array)
  copyin(addr,buf,len);
  return fd.write(buf,0,len);
}

void copyin(int addr, byte[] buf, int len) {
  for(i=0;i<len;i++)
    buf[i] =
      (memory[addr>>16][addr&0xffff]>>>(...)&0xff;
}
```

Security Concerns

- All interaction with the outside world happens though the Runtime class
- Runtime class applications have no way of accessing the host when file and network IO are disabled

Implications

- Even untrusted unsafe code can be run without fear of security problem
- Say a buffer overflow was found in the JPEG decoder
- The worst an attacker could do is generate an incorrect image because the JPEG decoder would have no reason to allow access to the outside world or the host

Gory Details

- All the code examples in this presentation are made up
- The source code generated isn't nearly as neat and pretty as the examples
- The preferred way to use NestedVM is to generate JVM bytecode directly, which is even less pretty

Real Code Translation

- Registers are actually stored in local variable, not fields, but they are written back to fields when necessary
- Jumps to statically known targets are implemented with the JVM goto instruction, avoiding the costly switch() on the pc
- Case arms aren't actually generated for every pc, only potential jump targets

Many run() methods

- The JVM limits the maximum size of a method to 64K
- Multiple run methods and a “trampoline” solve the problem

```
public void trampoline(int pc) {
    for(;;) switch(pc&0xffffff00) {
        case 0xa0020000: run_a0020000(); continue;
        case 0xa0020100: run_a0020100(); continue;
        ...
} }
```

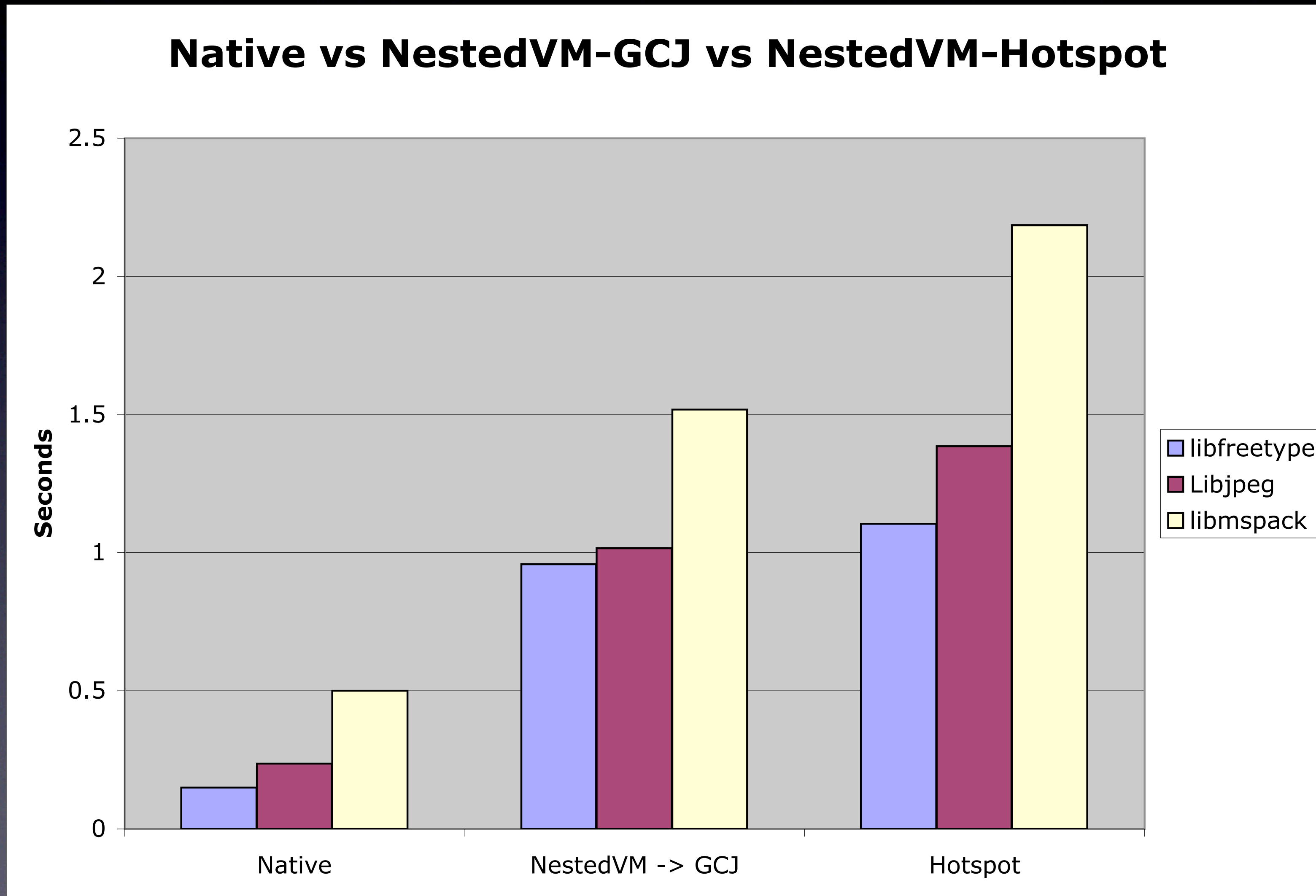
Real Memory Representation

- Memory is actually represented with two parallel arrays

```
int mem_read[65536][];
int mem_write[65536][];
```

- This allows us to implement memory protection (read only pages)
- Read only pages have a null mem_write entry, causing an NPE on access

Performance



Future Directions

- Target NestedVM to .NET, Parrot and other VMs
  - The techniques used by NestedVM are not JVM specific, they can be applied equally well to any safe virtual machine