Just in Time compilers - breaking a VM

Roland Lezuo <roland.lezuo@chello.at>
Peter Molnár <peter.molnar@wm.sk>

November 18, 2007

1 About CACAO

CACAO is a multiplatform Java Virutal Machine featuring a just-in-time compiler. Although CACAO features an interpreter, by default it works in JIT-only mode, so all code gets compiled prior to execution. The CACAO project was started in 1997 as a research project at Vienna University of Technology. Today the project is fully covered by the GPL v2 license.

2 CACAO Codegenerators

CACAO provides code generators for many platforms: currently code generators for ALPHA (FreeBSD, Linux), ARM (Linux) i386 (Cygwin, Darwin, FreeBSD Linux), MIPS (Irix, Linux), POWERPC (Darwin, Linux, NetBSD), SPARC64 (Linux), x86_64 (Linux) and s390 (Linux) are available. A code generator has to implement a defined internal interface consisting of a set of exoported functions and symbols and is linked in statically into the virtual machine.

3 Java bytecode

The Java compiler does not produce machine code which can be executed on the host CPU directly but an intermediate representation called *bytecode* targeting a virtual machine. There are around 200 bytecode instructions defined in the Java Virtual Machine Specification[1] The most notable difference between java byte code and usual machine code is that bytecode instructions

[1] `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`

Listing 1: Stack operations

```
iconst_3
iconst_5
iadd
```

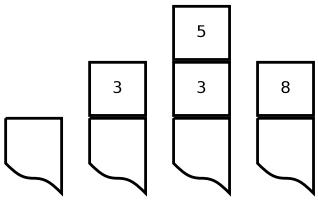

Figure 1: Stack changes

don't use registers as operands, but operate on a *operand stack* instead what leads the notion of a computation model called *stack machine*.

The program in listing 1 manipulates the stack as shown in figure 1: the instruction `iconst_3` pushes the integer 3 on top of the stack, `iconst_5` pushes 5, `iadd` takes the two topmost elements of the stack, adds them and pushes the result back. The stack is growing from the bottom to the top.

The operand stack consists of 32 bit wide *stack slots*. A single stack slot can accomodate a value of the primitive types `boolean, char, byte, short, int` or an object reference. To accomodate a `long` or `double` value, two stack slots are used.

Instructions are variable sized and consist at least of one byte - the opcode optionally followed by several bytes representing operands embedded in the instruction itself. The `getfield` instruction for example is used to retrieve the value of an object's field and contains a two byte field specifying the fields index. The object reference is poped from the stack and the result - the field's value - is pushed on the stack.

Arithmetic instructions are typed and special variands are defined for the various primitive types: (e.g. `iadd` adds two `int` whereas `ladd` adds two `long` values).

4 Register allocation

A naive compiler would generate machine code that would map the java operand stack to a stack located in memory. This is actually the approach used by the Jikes RVM baseline compiler and the approach kaffe's JIT used to use but is suboptimal, because of the property of memory accesses beeing

Listing 2: Codegeneration macros

```
#define M_OP3(opcode,y,oe,rc,d,a,b) \
  do { \
    *((u4 *)cd->mcodeptr) = (((opcode)<<26) | ((d)<<21)\
        | ((a)<<16) | ((b)<<11) | ((oe)<<10) | ((y)<<1)\
        | (rc)); \
    cd->mcodeptr += 4; \
  } while (0)

#define M_IADD(a,b,c) M_LADD(a,b,c)
#define M_LADD(a,b,c) M_OP3(31, 266, 0, 0, c, a, b)
```

expensive. CACAO instead allocates the slots of the java operand stack to CPU registers, for example stack slot 2 to the general purpose register 16. In the case that there are more stack slots needed than registers available, stack slots are mapped to memory locations. On RISC plattforms, they need to be loaded into registers before usage, and stored back afterwards.

5 Code generation macros

The code generator iterates over all instructions of the method to be compiled and depending on the opcode, translates them into native machine code. The generated machine code is written to temporary memory and afterwards copied to an executable memory location. It is generated by macros, so care has to be taken for side effects of arguments which could be evaluated twice. To ease maintenance of the code generators, all platforms try to adhere to naming conventions originally inspired by the alpha architecture. Listing 3 shows the implementation of java's `iadd` operation, and addition of two 32 bit signed values on POWERPC64. First, the operands are loaded, then the macro `M_IADD` is used to emit machine code that adds the values in two registers and stores the result in a desitnation register, `M_EXTSW` is needed for sign extension and is platform specific and finally the result is stored in the destination register. `jd` and `iptr` contain a pointer to the state of the JIT compiler and the currently processed instruction. The implementation of the macro `M_IADD` is shown in listing 2.

The operands of bytecode instructions are allocated to registers or memory. On load-store architectures, memory operands need to be loaded into registers prior to use what is achieved using the function`emit_load_s1`, `emit_load_s2`

Listing 3: Codegeneration for `iadd`

```
case ICMD_IADD:
  s1 = emit_load_s1(jd, iptr, REG_ITMP1);
  s2 = emit_load_s2(jd, iptr, REG_ITMP2);
  d = codegen_reg_of_dst(jd, iptr, REG_ITMP2);
  M_IADD(s1, s2, d);
  M_EXTSW(d,d);
  emit_store_dst(jd, iptr, d);
  break;
```

and `emit_load_s3`. In case the operand was allocated to a register, they simply return the register number, otherwise, code is generated to load the memory operand into a scatch register and the number of the scratch register is returned. The destination register of an operation is retrieved using the function `codegen_reg_of_dst`, which may again return a scratch register for memory destinations and finaly `emit_store` generates code to store the result in case it belongs to memory. See listing 3 for an example showing the implementation of the `iadd` byetcode instruction on POWERPC64.

6 Post compile time code patching

One reason the generated code is written into a buffer is due to unresolved jumps. Imagine a forward jump in a method wheter the target address points into code still not generated and the compiler does not know the exact offset in advance as it depends on the instructions in between. For that reason a post-pass has been added to the compiler which patches the code after generation. During machine code generation a function named `codegen_add_branch_ref` is responsible for collecting positions of branches that could not be resolved and associating them with target basic blocks. The branch instructions are then patched using the machine dependend function `md_codegen_patch_branch` to contain the correct offset after the complete method has been compiled. By using the machine dependent patching function the post compilation phase can be kept platform independent.

7 Data segment

The generated code makes use of constant values: integer constants, address constants (function entry addresses, addresses of static members). Some

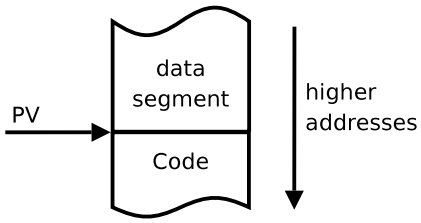

Figure 2: Data segment layout

architectures support immediate values of the native word size, so such values can be embedded in the instruction flow whike other architectures have a fairly limited range of immediate operands, so those values need to be placed into memory. Beacause of this the executable method's code has a block of memory prepended called the *data segment* (see figure 2) holding those constant values. On most architectures, there is one `pv` register reserved to hold the *procedure vector* - the current method's entry point. The values on the data segment can then be loaded relatively to the `pv` register with negative offsets, or relatively to the current program counter with negative offsets.

The data segment of each method always contains a *method header*. This is a data structure containing metadata about the method, like a pointer to a method descriptor, the stack frame size, the exception table, the line number table (see **??** for details).

8 Runtime code patching - Patchers

In java, classes are loaded by the run-time system only if they are needed. If generating code for a method that depends on other classes (uses static fields, calls methods), the runtime system needs information about the referenced class, and therefore it has to be loaded as well. One attempt called *eager loading* consists of loading all those referenced classes at compile time but it showed to be suboptimal, because at run-time, the code using the referenced class may actually never be reached. A better attempt is to deffer expensive class loading to the point, where the code that uses the class is reached. This is called *lazy loading*.

For *lazy loading*, incomplete code that has to be *patched* at run-time with the missing information is generated. The first instruction of the imcomplete code portion is replaced by a trap instruction and a *patcher reference* is created: a datastructure containing data about the missing information associated with the position of the trap instruction.

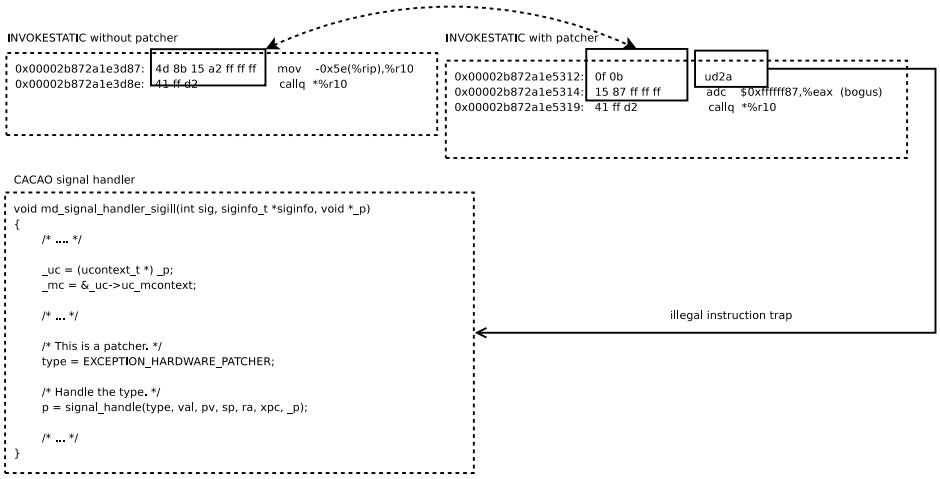

Figure 3: Patcher assembler output (new)

Consider the example of a `getstatic` instruction, which loads a static field of a given class. The class may be unresolved when the bytecode is translated in which case the runtime system has to load and initialize the class, resolve the address of the member prior to execution of the generated code. For this purpose the first instruction of the machine code sequence is replaced by an illegal instruction. Once it is reached, the operating system delivers a signal the the virtual machine and control is passed to the registered signal handler. The signal handler needs to be able to differ patchers from exceptions, so it first examines the failing instruction, whether is really corresponds to a patcher call. The handler then looks up the proper patcher by using the mapping of positions to be patched to *patcher references* and invokes. The code generator needs to provide a function called `emit_trap` capable that generates a trap instrucion.

Figure 3 shows the generated assembler code on the x86_64 architecture: the illegal instruction (`u2da`) is generated where patching is needed and once reached control flows to a signal handler written in C. The disassembler wrongly interpretes the bytes `15 87 ff ff ff` as `adc` instruction. They are part of the offset of the `mov` instruction covered by the `ud2a` instruction.

A race condition exists when patching the trap instruction in case he instruction can not be overwritten atomically on multiprocessor machines. One thread could just patch back the original code, while a different thread executes exactly this code and comes across a half patched instruction. For that reason single word instructions are used for trapping, as they can be written back atomically.

9 Compiler invocation

Beacause just-in-time compilation of methods is expensive and accounts to run-time, CACAO tries to deffer it, simillary as it does for class loading. A method is normally compiled the first time it is called. To achieve this, when a class gets loaded, for each method a so called compiler stub is generated. A compiler stub is a small piece of code, usually a single trap instruction combined with a pointer to the method's descriptor. Pointers to compiler stubs are placed where method entry points would be placed normally: in the class descriptor and in virtual function tables.

If such a compiler stub is invoked, the trap instruction causes control to be passed to a signal handler which extracts the method descriptor from the stub and passes it to the compiler subsystem. The compiler generates machine code for the method and returns the method's entry. Then, the machine code before the call instruction is examined, to determine the *method pointer*: the address where the pointer to the stub's entry was loaded from. This is a virtual function table entry, the data segment, or an immediate operand in executable code. This location is then overwritten with the actual method entry, so that further calls to the method are redirected to the newly generated machine code.

10 Exceptions

Exceptions are an integral part of the Java language used a lot. Nonetheless exceptions are rare events and occur irregularly.

Each method has an exception handler table associated. This table describes the start and end instruction of each exception handler directly corresponding to the Java language `try` clause. When an exception occurs at some point in the program, a lookup is performed in the exception table. The type of the occurring exception is compared to the type of each handler covering the throwing instruction.

If a match can be found the handler is executed, else the exception is propagated outside the method. For the caller this looks like a throwing invoke instruction. As the caller of a method is unknown at compile time, the caller has to be determined at runtime. This is achieved by looking up the return address which is stored on the stack. The offset is known as CACAO knows about the stack usage of each method. Stack space is allocated on method entry and no dynamic allocation is performed.

An operation called "stack unwinding" is performed whenever an exception is propagated to its caller. As control flow continues at the invok-

ing instruction all callee saved registers have to be restored for each stack frame unwound. Callee saved register are stored on the method stack when a method is entered, therefore the restore operation is implemented by loading these registers from known stack locations.

This process either terminates when an appropriate handler has been found or the whole stack is unwound in which case the exception is unhandled and the program will be aborted.

In CACAO no explicit code is generated for calling back the runtime when an exception occurred but an illegal memory operation is performed. POSIX compatible operation systems provide a signal handling mechanism which invokes a function in this case. This signal handler tests if the memory operation was performed intentionally and if so it calls the exception handling code. In case the memory access took place unintentionally an internal exception is thrown and the vm aborts.

When native functions have been called they could have thrown an exception too. Natives can not throw exceptions directly but have to notify the runtime by setting a flag in the environment. When they return the environment is checked for an exception and exception handling code is executed when needed. Exception handling is complex because natives may call back into Java code. The stack layout is only known in JIT code, native code has a different stack layout and stack unwinding would fail when a native frame is found. Therefore a chained data structure called stackframe info is built up when invoking natives. Figure 4 illustrates this chaining. Technically there are no `stackframeinfo` structures for JIT frames, as this stack layout is known and contains all needed information already.

11 Bytecode Verification

Because the java virtual machine was designed to provide a sandbox environment, it can't just start executing untrusted bytecode. It would be easy to construct malicious bytecode that if executed would crash the virtual machine. Therefore all bytecode is subject to verification prior to execution. Bytecode verification includes basic sanity checks of the class file, type checking of bytecode instructions, checks for operand stack underflow and enforcement of access protection as required by the java language.

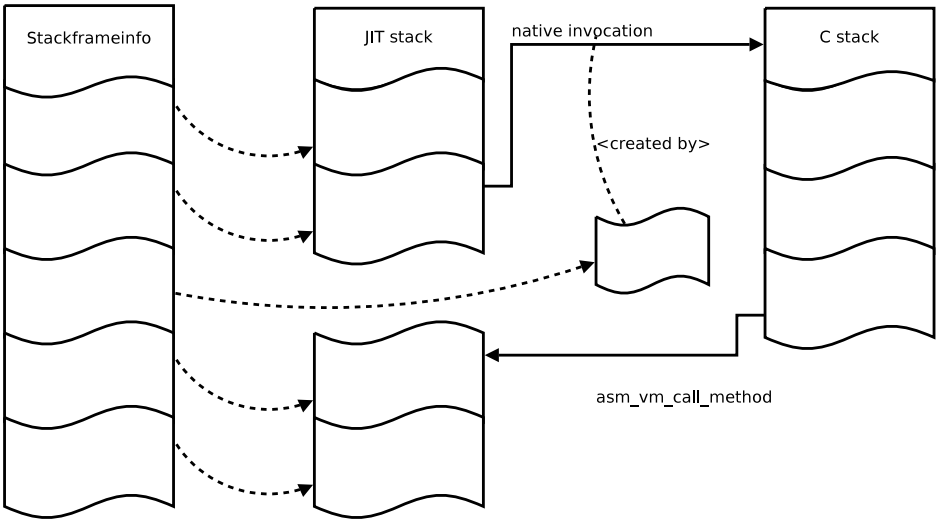

Figure 4: Stackframeinfo chaining with native invocation

12 Problematic byte code instructions

When looking for security problems you should first start by looking at "strange" behaviour defined in the specification. The Java Virtual Machine Specification is available online. Chapter 6 ha a list of all bytecode instructions. A JVM vendor has to implement them acording to their specification. By looking through that list some strange instruction show up.

- TABLESWITCH, LOOKUPSWITCH The tableswitch instruction is used to implement the switch/case statement and is an optimization of the more generic lookuptable instruction. The lookuptable is followed by possible 2^{32} pairs of integer, address pairs. Tableswitch is followed by 2^{32} possible addresses. That is quite a number! Espcially when one also knows that the size of a single method is limited to `0xFFFF` bytes by limitations from the classfile format.

- JSR, RET Another example are the jsr and ret instructions. Their purpose is to implement the try/finally clause of the Java language. The jsr instruction does no invoke any methods (despite its name), it jumps to the finally block and stores the return address on the stack. The ret instruction fetches the return address from a local variable, for an intentional asymetry. The bytecode verifier has to treat return addresses as an additional type to prevent hackers from returning to an integer value they calculated.

This alone are no security problems per se, but they are subtile details which have to be implemented 100% correct to keep the sandbox tight.

13 Problematic assembler instructions

When translating the byte code into machine code appropiate instruction have to be selected. There are different approaches for code generators. Some vendors define a description language and generate the code responsible for instruction selecting, others implement this by hand. Whatever approach is taken, the instructions available are determined by the architectur the code is executed on.

13.1 POWERPC64

The POWERPC64 architecture is an enhancement of the POWERPC architecture and offers 64 bit address space and a 32 bit compatibility mode. All instruction have a fixed 32 bit size. Immediate values are of course even smaller than 32 bits. As a consequence loading a 64 bit address takes more than 1 assembler instruction.

```
lis 4,msg@highest   # load msg bits 48-63 into r4 bits 16-31
ori 4,4,msg@higher  # load msg bits 32-47 into r4 bits  0-15
rldicr  4,4,32,31   # rotate r4's low word into r4's high word
oris    4,4,msg@h   # load msg bits 16-31 into r4 bits 16-31
ori     4,4,msg@l   # load msg bits  0-15 into r4 bits  0-15
```

It takes 5 to be exact. When generating code the size of the generated code is an important factor. Not only for execution speed. And using 5 instruction to load an address (something happening very frequently) can not be afforded. For that reason relative addressing modes are used whenever possible. Assuming that register `r12` contains a valid base address loading an 64 bit value may be implemented as short as the next listing shows.

```
ld      4,0x1234(12)
```

This is just one instruction. In CACAO a datasegment is used to store constant values and a register is reserved to point to the start of the datasegment. So when needing to load an address, a relative addressing load instruction can be used.

The problem here is that the offset is limited to 13 bits, that is 8192 bytes or 8 KiB. The interesting question is what happens for bigger offsets? That depends on the implementation, but it will probably be one of the following 3 cases:

- good: The compiler checks the offset, detects the overflow an emits an instruction sequence capable of correctly handling the case.

- not so good: The offset is trimmed to fit into 13 bit, an integer overflow occures which can lead to an exploit.
- even worse: The offset is not trimmed. As most code generators OR together bitfields it is very likely that the instruction will be changed. This can most likely be exploited.

14 Examples found in CACAO

14.1 PPC64 32 bit interger overflow vulneribility

When loading addresses the offset is truncated to 32 bit (M_LLD macro in codegen.h). This leads to offsets larger than 4 GiB to wrap around and accessing the datasegment at the beginning. The attacker has full control over the contents of the datasegment as the content is determind by the method executed. One way to fill the datasegment is by creating address and interger constans (ICONST and ACONST bytecode instructions). The exploit is of theoretical nature as a 4 GiB sized datasegment implies a 4 GiB sized class file which is not possible.

14.2 PPC64 25 bit integer overflow vulneribility

The POWERPC64 branch instruction takes a 23 bit offset argument, but needs 4 byte aligned target addresses, which effectivley gives a 25 bit branching offset. In CACAO conditional branches are not tested correctly for an overflow and branch addresses are trimmed to fit into 23 bit. An branch offset of `0x3FFFFFF` will be interpreted as `-1` and therfore jump backwards instead of forwards. By jumping backwards the datasegment is targeted which is in control of an attacker. The size of a method must be around 64 MiB for this explot to work. As java methods may only consist of 65535 instructions (classfile limitation) each bytecode instruction would need to use 1024 bytes of instruction code. There is no byte code instruction using 1024 byte of assembler instructions, so no exploit can be developed targeting this weakness.

14.3 x86_64 32 bit integer overflow vulneribility

A similar vulneribility has been found for x86_64. But it can not be exploited by the same argument as above.

14.4 All architecture exception handler exploit

In CACAO there are special conventions for propagating the exception object during *stack unwinding*. A `ATHROW` instruction is implemented as follows: the pointer to the exception object and the faulting program counter are placed into scratch registers `itmp1` and `itmp2` respectively and an assembly language function, `asm_handle_exception` is jumped to that performs *stack unwinding*. The program counter and exception type are then used to find an exception handler block which is jumped to. The handler code expects the register `itmp1` to contain the exception object pointer. This approach makes use of the assumption that the only way to reach an exception handler is via the *stack unwinding* process. This is actually always true for compiler generated bytecode but at bytecode level it is perfectly leagal to directly jump into an exception handler block without an exception thrown. The exception handler code then interprets the contents of the scratch register `itmp1` as exception pointer. Because `itmp1` is used in arithmetic operations as scratch register, it contents can easily be controlled and set to an arbitrary value.

To exploit this vulnerability a virtual method on this arbitrary object pointer is going to be invoked. When calling an object's Nth virtual method, first the pointer to the *virtual function table* is loaded from offset 0 of the object pointer. Then, the method's entry point is loaded from slot N of the virtual function table. Finally, the method's entry point is jumped to.

Using arrays, a fake object and a fake virtual function table with all entries pointing to shell code are constcutred as shown in the source code in figure 5. To set up the pointers in the arrays a method is needed to get the address of the first element of a java array. This can easealy be achieved by abusing of the default `toString()` implementation which outputs a string containing the object's class name and its address in memory. In cacao's implementation, an array starts with a fixed-sized header followed by data elements, so the address of element 0 is calculated by adding a fixed offset to the array pointer. Now if a virtual function on this fake object is called, control is passed to the shell code.

```
int addressOf(Object o) {
        // extract and return address from o.toString()
}

// Architecture dependent size of array header
// First array element is at this offset from array pointer
int arrayHeaderSize = 16;
// Shell code
byte[] code = { /* shell code, null bytes allowed*/ };
// Virtual function table with 100 slots
// Each element (method entry) points to the shell code
int[] vftbl = new int[100];
for (int i = 0; i < vftbl.length; ++i)
        vftbl[i] = addressOf(code) + arrayHeaderSize;
// Object, first words points to virtual function table
int[] obj = new int[1] { addressOf(vftbl) + arrayHeaderSize };
// Object pointer has to point to element 0 of obj
int objPtr = addressOf(obj) + arrayHeaderSize;
```

Figure 5: Constructing a fake java object

14.5 16 and 12 bit invoke virtual integer overflow on PPC32 and S390 exploit

As described in section 14.4, to call a virtual method, two loads are involved: the load of the virtual function table, and then the load of the method entry from a specific slot of the virtual function table. The displacement of a load instruction has a limited range: on i386 and x86_64 it is limited to 32 bits, on ppc to 16 bits, on s390 to 12 bits. If the load of the method entry is implemented as a single load instruction, the maximal load displacement limits the number of virtual methods that can be supported by such a design: $2^{31}/4$ on i386, $2^{31}/8$ on x86_64, 8192 on powerpc and 4096 on s390. The question is, what happens if a class happens to contain more virtual methods? On most achitectures, this case is protected by an assertion. If assertions are turned off, the displacement of the load will just be trimmed to fit into the maximal displacement bitsize. That in turn means that, if we call a virtual method who's entry fails to get loaded because of the displacement limitation, a different method will be called.

To exploit this vulnerability, let's suppose the displacement in the load instruction is unsigned, and that it can be used to load a maximum of MAX methods from the virtual function table. A class with MAX virtual methods is generated, each taking one word sized integer as argument and just returning that argument followed by two methods with the signatures `Object intToObject(int i)` and `int objectToInt(Object o)`. If `objectToInt` is called, its entry should be loaded from slot $MAX + 1$ of the virtual function table but after trimming the offset, the entry will be loaded from slot

1 instead, where a method resides that reinterprets the object reference as integer and just returns it. This way pointers can be converted to integers and vice versa, bypassing the type system.

Once this type unsafe "casting" functions are available a fake object is constructed like in section 14.4 with `objectToInt` used to get the addresses of the arrays and `intToObject` used to "cast" the address of the fake object to an `Object`. If calling some virtual method on this object pointer, controll is passed to the shell code.