The Design and Implementation of the Mosquito Virtual Machine

Scott W. Dunlop
Ephemeral Security

May 27, 2006

Contents

Introduction

The Mosquito Virtual Machine (MOSVM) is a compact portable virtual machine capable of executing concurrent Mosquito Lisp (MOSLISP) processes on a wide variety of host operating systems and architectures. The Mosquito Secure Remote Execution Framework (MOSREF) is implemented using MOSVM and MOSLISP as a convenient platform abstraction.

The Mosquito Virtual Machine must be compact; in the normal operation and usage of MOSREF, the Mosquito Virtual Machine and an initial application loader written in Mosquito Lisp must be deployed across the network, occasionally using a remote code execution exploit, as part of a security penetration test.

MOSVM must also be portable across a wide range of host operating systems and environments, permitting the deployment of security tools to any host architecture commonly found in modern data centers, including Windows 2000, Linux, OpenBSD, and Solaris.

Since network security audits require many interactions with hostile or nonresponsive network applications, MOSVM provides a pervasive concurrency model that permits MOSVM processes to suspend themselves to await a response to a network request, or a timed event. MOSVM does not provide a preemptive multiprocess model, reducing the need for carefully planned synchronization operations.

Mosquito Lisp is a procedural language derived from Scheme and Lisp 1.5. Mosquito Lisp differs from Scheme in many places to reduce design complexity for MOSLISP implementations and make the language more approachable for programmers who are unfamiliar with Functional Programming. Many of these changes also make Mosquito Lisp implementations more efficient for common programming tasks.

Chapter 1

The Inner Interpreter

The inner interpreter executes a single process within the Mosquito Virtual Machine; the inner interpreter may be interrupted by calls to the halt (4.2), next (4.3), or pause (4.4) primitives, or when waiting on a network event.

MOSVM's inner interpreter is a stackless register machine built on top of the MOSVM memory management system (3). The interpreter employs a limited set of instructions to construct function calls, manage the lexical environment, and perform basic control operations; these basic instructions express the fundamental operations required to construct calls to the primitive functions (4) included with MOSVM.

1.1 Virtual Machine Registers

1.1.1 The Apply Register (AP)

The Apply Register contains a pointer to the next call frame, under construction. When a CALL instruction is executed, the call frame referenced by AP shall contain a list of arguments, the function called, and a copy of all registers that must be returned when the function returns.

1.1.2 The Context Register (CP)

The Context Register contains a pointer to the current call frame, which contains registers that must be restored on function return.

1.1.3 The Environment Register (EP)

The Environment Register refers to a list of vectors that provide storage for variables bound within the current lexical environment. The innermost lexical scope is first in this list.

1.1.4 The Instruction Register (IP)

The Instruction Register refers to the instruction currently being evaluated; instructions in MOSVM are of fixed length, and consist of a pointer to the operator, and two operand values. Unused operands should contain null values to simplify memory tracing.

1.1.5 The Result Register (RX)

All primitive functions in MOSVM leave the value yielded by their execution in RX; various load instructions also leave the loaded value in RX. The contents of RX is placed by the ARG1 and ARGN instructions in AP's argument list.

1.1.6 The Guard Register (GP)

The Guard Register contains a list of guard frames. These guard frames contain functions that will be called if an error occurs, providing a versatile exception handling system. New guards may be added to GP using the GAR instruction, and removed using the RAG instruction.

1.2 Virtual Machine Instructions

1.2.1 The Argument Instruction (ARG)

Places the value in RX as the next item in the call frame being constructed in AP. Emitted by the compiler after the evaluation of an expression that is a term in an application.

```
(call-add-item! ap rx)
(set! ip (next-instr ip))
```

1.2.2 The Call Instruction (CALL)

The Call Instruction causes the contents of the virtual machine registers, excluding RX, GP and AP, to be stored in the call frame referenced by AP. The CALL Instruction supports the application of Closures, Primitives and Multimethods.

```
(define fn   (call-fn ap))
(define args (call-args ap))

;; Multimethods require thrashing to find first fit method.
(while (is-multimethod? fn)
  (if (multimethod-accepts? fn args)
    (set! fn (multimethod-impl fn))
    (set! fn (multimethod-reject fn))))
```

```
(if (is-closure? fn)
  (begin (set-call-cp! ap cp)
         (set-call-ep! ap ep)
         (set-call-ip! ap (next-instr ip))
         (set! cp ap)
         (set! ep (closure-env fn))
         (set! ip (proc-instr (closure-proc fn))))
  (apply (prim-impl fn) args))
```

1.2.3 The Closure Instruction (CLOS)

Constructs a new closure using the current environment, EP, starting at the next instruction, then jumps to the address referenced in BX. AX is used as the name of the closure, and should be memorable.

```
(set! rx (make-closure ep ip))
(set! ip (instr-ax ip))
```

1.2.4 The Guard Instruction (GAR)

Constructs a new guard using the closure in RX, and the instruction referenced by AX.

```
(set! gp (cons (make-guard cp ap ep (instr-ax ip))
               gp))
(set! ip (next-instr ip))
```

1.2.5 The Jump If False Instruction (JF)

If the contents of RX is the false value, the IP register is set as described in the Jump Instruction.

```
(if (eq? #f rx)
  (set! ip (instr-ax ip))
  (set! ip (next-instr ip)))
```

1.2.6 The Jump Instruction (JMP)

Sets the IP register to reference the Instruction referenced by AX. Produced primarily by control structures in Mosquito Lisp.

```
(set! ip (instr-ax ip))
```

1.2.7 The Jump If True Instruction (JT)

If the contents of RX is not the false value, which is the Scheme / Mosquito Lisp definition of true, the IP register is set as described in the Jump Instruction.

```
(if (eq? #f rx)
  (set! ip (next-instr ip))
  (set! ip (instr-ax ip)))
```

1.2.8 The Load Bound Instruction (LDB)

The Load Bound Instruction employs the environment chain in the EP register to load a bound variable. The index in AX indicates the proximity of the lexical scope, with 0 being the innermost and subsequent values indicating higher and higher scopes. The index in BX indicates the variable's index.

The Mosquito Lisp compiler, unlike most Scheme implementations, employs a technique called Let Compression that allocates additional slots for all variables bound within a closure instead of creating anonymous closures to express these variables. Space for variables declared using inner defines is also allocated using this technique.

This optimization, while expensive at compile time, means a tremendous increase in efficiency for variable assignment and access. For more information, see the USEN and USEA instructions.

```
(set! rx (vector-ref (list-ref ep (instr-ax ip))
                     (instr-bx ip)))
(set! ip (next-instr ip))
```

1.2.9 The Load Constant Instruction (LDC)

The Load Constant Instruction places the AX operand in RX; this is often produced by the compiler when it encounters quoted expressions, or atomic values like strings, nulls, or integers.

```
(set! rx (instr-ax ip))
(set! ip (next-instr ip))
```

1.2.10 The Load Global Instruction (LDG)

The Load Global Instruction checks for a global value assigned to the symbol contained by AX, and, if one is present, places it in RX. If none is found, an error is signalled. This instruction is produced by the compiler when a variable is referenced that is not bound in the current lexical scope.

```
(set! rx (get-global (instr-ax ip)))
(set! ip (next-instr ip))
```

1.2.11 The New Frame Instruction (NEWF)

The New Frame Instruction creates a new call frame, and places it in AP. This instruction precedes every function application.

```
(set! ap (make-call-frame))
(set! ip (next-instr ip))
```

1.2.12 The Remove Guard Instruction (RAG)

Removes the first guard referenced in GP from the guard list.

```
(set! gp (cdr gp))
(set! ip (next-instr ip))
```

1.2.13 The Return Instruction (RETN)

The RETN Instruction restores the registers stored in the call frame referenced by CP, and is used to signal the completion of evaluation of a closure.

```
(set! ep (call-ep cp))
(set! ip (call-ip cp))
(set! cp (call-cp cp))
```

1.2.14 The Scatter Instruction (SCAT)

Places each item in the list in RX in the call frame being constructed in AP, sequentially. Emitted by the compiler when the form `(@ ...)` is encountered.

```
(for-each (lambda (rx) (call-add-item! ap rx))
          rx)
(set! ip (next-instr ip))
```

1.2.15 The Store Bound Instruction (STB)

The Store Bound Instruction assigns the value in RX to the variable referenced by BX in the environment referenced by AX in EP, similar to the Load Bound instruction (1.2.8).

```
(vector-set! (list-ref ep (instr-ax ip))
             (instr-bx ip)
             rx)
(set! ip (next-instr ip))
```

1.2.16 The Store Global Instruction (STG)

The Store Global instruction assigns the value in RX globally for the symbol referenced by AX. In MOSVM, assignment and retrieval of global variables is a constant time operation.

```
(set-global (instr-ax ip) rx)
(set! ip (next-instr ip))
```

1.2.17 The Tail Instruction (TAIL)

The Tail Instruction is similar to CALL, but copies information about the function call from AP to CP instead of replacing CP with AP. This is semantically equivalent to a CALL instruction, followed by a RETN instruction, and permits the compiler to support Scheme style tail call optimization. (In fact, Mosquito Lisp achieves tail call optimization by replacing all contiguous CALL and RETN instructions with a TAIL instruction.)

```
(define fn   (call-fn ap))
(define args (call-args ap))

;; Multimethods require thrashing to find first fit method.
(while (is-multimethod? fn)
  (if (multimethod-accepts? fn args)
    (set! fn (multimethod-impl fn))
    (set! fn (multimethod-reject fn))))

(if (is-closure? fn)
  (begin (set-call-data! cp (call-data ap))
         (set! ep (closure-env fn))
         (set! ip (proc-instr (closure-proc fn))))
  (apply (prim-impl fn) args))
```

1.2.18 The Use All Instruction (USEA)

Constructs an environment, and prepends it to EP with BX slots, then loads AX arguments from CP. Any extra arguments from CP are left in a list, and placed in AX + 1.

```
(define max (instr-ax ip))
(define env (make-vector (instr-bx ip)))
(let loop ((ix 0)
           (args (call-args cp)))
  (cond ((= ix max)   (vector-set! env ix args))
        ((null? args)) ;;; Do nothing.
        (else         (vector-set! env ix (car args))
                      (loop (+ ix 1) (cdr args)))))
```

```
(set! ep (cons env ep))
(set! ip (next-instr ip))
```

1.2.19 The Use Specific Instruction (USEN)

Constructs an environment, and prepends it to EP with BX slots, then loads AX arguments from CP. If there are more than AX arguments, an error is raised.

```
(define env (make-vector (instr-bx ip)))
(let loop ((ix 0)
           (args (call-args cp)))
  (if (not (null? args))
     (vector-set! env ix (car args)))
  (loop (+ ix 1) (cdr args)))
(set! ep (cons env ep))
(set! ip (next-instr ip))
```

1.3 Inner Interpreter Structures

To Be Documented.

1.3.1 Call Frame

To Be Documented.

1.3.2 Environment

To Be Documented.

Chapter 2

Process Scheduler

The Mosquito Virtual Machine provides a simple lightweight process model intended to simplify the implementation of network applications which often spend protracted periods waiting on input.

The Mosquito Virtual Machine will terminate if all of its processes are halted, or suspended.

A MOSVM process may be in one of the following states: active, paused, suspended or halted.

2.0.3 The Active Process

There may only be one or zero active processes in the Mosquito Virtual Machine. The active process controls the registers and flow of execution so long as it is active. An Active Process may Pause, Halt or Suspend itself.

2.0.4 Paused Processes

A Paused process is scheduled to become Active after the current Active process yields control. A Paused Process may be Halted or Suspended, or it may become Active if no other process is active.

2.0.5 Halted Processes

A Halted process has either completed its task, or has signalled an error that was not caught by a guard. Halted Process may not change its status, and, if there are no external references, is a candidate for reclaimation by the garbage collector.

2.0.6 Suspended Processes

A Suspended process is waiting on an event, and will not become active until one has been sent to the process. A Suspended process may be Halted, but can only be made Active or restored to Paused state when an event has been sent.

Chapter 3

Memory Management

To Be Documented.

Chapter 4

Primitive Functions

In the Mosquito Virtual Machine, a primitive function is any function which is wholly implemented in C, and included in the MOSVM executable.

To Be Documented.

4.1 The Error Primitive, error

To Be Documented.

4.2 The Halt Primitive, halt

To Be Documented.

4.3 The Next Primitive, next

To Be Documented.

4.4 The Pause Primitive, pause

To Be Documented.

4.5 The Re-Error Primitive, re-error

To Be Documented.

4.6 The Spawn Primitive, spawn

To Be Documented.

4.7 The Thaw Primitive, thaw

To Be Documented.

Chapter 5

MOSVM Object Packages

The Mosquito Virtual Machine provides functionality for extracting a MOSVM program from a serialized string. The MOSVM Object Package format is designed to be easy to implement and compact due to the design requirements of the Mosquito Virtual Machine.

The Mosquito Virtual Machine provides the capability to reconstruct a value from a MOSVM Object Package. The Mosquito Lisp compiler, which runs on the Mosquito Virtual Machine, provides the additional functionality required to encode Object Packages.

5.1 The Object Package Header

The MOSVM Object Package Header is a word, in IETF network short order, expressed in two bytes. If the most significant bit is set, the Object Package expresses an immediate integer, as explained in ″References and Immediates.″ (5.2) If the value is one of: 0x7FFF, 0x7FFD, or 0x7FFC, it is assumed to express one of the special atomic values, also explained in ″References and Immediates.″

Otherwise, this initial word expresses the number of records contained by the object package. The first record is the ″root object″ that was packages, and shall be returned by the MOSVM ″thaw″ primitive. (4.7)

5.2 References and Immediates

In the MOSVM Object Package format, a given value may be either a Reference or an Immediate. An Immediate value is an integer in the range 0 to 32767, inclusive, and is expressed as a 16 bit value with the most significant bit set, and the value expressed using the least significant bits. Immediates are used to express integer values that fit within the specified range. Larger integers are encoding using Integer Records. (5.3)

There are, in MOSVM, three special references that are assumed to refer constant values that are ubiquitous in Lisp: 0x7FFF, the null value, 0x7FFD, the false value, and 0x77FC, the true value.

This leaves reference space for 32,000 records, which is sufficient for packaging MOSVM modules and programs.

All References and Immediates are encoded in IETF network short order, and expressed in two bytes.

5.3 The Integer Record

Integer Records express integer values that cannot be expressed as immediates. (5.2)

A Integer Record consists of four bytes, in IETF network long order.

5.4 The Pair Record

Pair Records encode a pair of values.

A Pair Record consists of two Values.

The pair `(alpha . beta)` is packaged as:

```
0003
PAIR        -- 0001 0002
SYMBOL 0005 -- 'alpha'
SYMBOL 0004 -- 'beta'
```

5.5 The Procedure Record

Procedure Records encode a Mosquito Virtual Machine procedure; generally, these procedure records are used to express MOSVM programs and modules.

A Procedure Record consists a length of the procedure record, followed by a sequence of instruction fields. An instruction field consists of a byte, corresponding with the instruction code, and, optionally one or two words expressing the instruction's operands. These words are omitted if the instruction does not employ the operand.

This optimization is expensive, but results in a tremendous savings in MOSVM since most instructions do not employ an operand.

The expression `(print "Hello, world!")` compiles to:

`(newf)(ldg print)(ldc "Hello, world!")(call)(retn)`

```
0003
PROC   0009 -- NEWF LDG 0001 LDC 0002 CALL RETN
SYMBOL 0005 -- 'print'
STRING 0013 -- 'Hello, world!'
```

5.6 The List Record

List Records encode a sequence of pairs that comprise a traditional Lisp list, by only encoding the car of each pair, and implying that the cdr of each pair refers to the next. List Records may only be used if the rest of the package only refers to the first pair in the list – this case is so frequent, it merits a special optimization.

A List Record consists of a count of Pairs encoded, no less than 1, and a sequence of Values, one per pair, in first to last order.

The scheme list `(alpha beta charlie)` is packaged as:

```
0004
LIST   0003 -- 0001 0002 0003
SYMBOL 0005 -- 'alpha'
SYMBOL 0004 -- 'beta'
SYMBOL 0007 -- 'charlie'
```

5.7 The String Record

A String Record consists of a two byte length, in IETF network short order, and a sequence of bytes expressing the string's contents, in first to last order.

The string `"abc\0efg"` is packaged as:

```
0001
STRING -- 0007 'abc' 00 'efg'
```

Index