FIRMAMENT
Operation Manual

Roberto Jung Drebes

Version 1.0

1 Introduction

This manual presents FIRMAMENT's specification and details its operation: the faultlet concept, instructions recognized by the tool's virtual machine and the supported control commands. This document is a starting point for people interested in testing network protocols and aplications using FIRMAMENT.

2 Faultlets

A typical drawback of usual communication fault injectors is the restricted way in which fault scenarios can be specified. The coding of the operations to be performed over messages depends on the possible actions allowed by the tool. The approach ends up being very strict and specific for pre-existing protocols.

Even when it is possible to specify actions based on message content, this is usually limited, since tests can be made on fixed message offsets. The data of a packet cannot be read and used as a parameter for the injector actions, nor processed combinedly to determine such actions.

An example of the inconvenience caused by this limitation is the processing of the IPv4 protocol header. IPv4 has a variable header size. If we would like to inspect its data area, it is not enough to simply jump over a fixed number of initial bytes belonging to the header. It is necessary to read, from the header, its size, and use this information as an index for accessing the packet data area. Making it even more complex, the IPv4 header size is indicated in the number of 32 bit words, so if the header size is wanted in bytes the information should be multiplied by 4.

This complication is not exclusive of IPv4. IPv6, unlike IPv4, has a basic header with a fixed size, but the transport layer header does not always immediately follows the basic header. Since it uses a chained header structure, the transport layer header is located only after the structure which may contain optional headers, and so faultlet processing should repeatedly verify the following header type until it finds a UDPv6 or TCPv6 header.

Traditional communication fault injectors can be developed considering these issues and by doing so support such protocols, through the previous knowledge about its format. However, this approach is not extensible to new protocols, and limits the test engineer to the scenarios and protocols already supported by the tool. Even if the tool supports the processing of IP headers, the test target can be an application protocol. Although many aplication protocol are documented through RFCs (*Request for Comments*), this is not always the case,

and so it is complex to develop a generic protocol testing tool using the usual approaches to specify scenarios.

FIRMAMENT extensibility comes from its new approach for fault scenary specification, through a simple, however flexible, technique. The tool employs the *faultlet* concept, a fault aplication whose difference is the power of expression of fault scenarios.

A faultlet is executed over each packet that crosses the communication flows, and it is able to inspect and modify the contents of the packet, discard, duplicate or delay it. In addition to the packet, a faultlet can act over the flow state variables: a set of general purpose registers. It can, therefore, move data between the packet and its state variables, and perform logical, arithmetic and control operations over these data. Hence, the execution flow of a faultlet can be modified by data read from the packet. This is the key characteristic of the tool: the possibility to create advanced control structures, with loops and branches, making FIRMAMENT appropriate to test any protocol.

Faultlets are independently configured for the input and output flows[1] of the IPv4 and IPv6 protocol families. A faultlet is executed by the FIRMVM virtual machine, a module of FIRMAMENT connected to the communication subsystem through the Netfilter interface. It is the FIRMVM virtual machine that actually associate a packet to a faultlet and the state variables. The next section describes the FIRMVM, listing the instructions and the state variables available to faultlet programming.

3 The FIRMVM virtual machine

The FIRMVM virtual machine is the module responsible for interpreting and processing the instructions which specify fault scenarios. FIRMVM works over 4 package flows, associated to the input and output of the IPv4 and IPv6 protocols. The fault scenario specification for these flows is done independently, writing faultlets to the tool's virtual files interface described in Section 6. This section presents the state variables avaliable for faultlets processing as well as the avaliable instructions for building faultlets.

3.1 State variables

In addition to the contents of the packets, each flow can operate over a set of general purpose registers. The registers are intianted when a flow is started and its state is kept during the sequential processing of all its packets, therefore being able to be used as state variables.

A set of 16 general purpose 32 bit registers is avaliable for each flow. From the point of view of the faultlet programmer, the registers use a network byte order representation (MSB/BIG_ENDIAN), i.e, it is in this format data must be read or written to the registers. However, FIRMAMENT does automaticallt convert from the native hardware format to this format. The register specification for the programmer in the FIRMASM language (described next) is always in the form R_x, where x is an integer from 0 to 15.

Each register can be configured to be auto-incremented periodically. To each increment is associated a wake up period, in millisenconds, and the register is incremented when this period is reached. Instructions allow the activation and deactivation of the auto-increment individually for each register. Increment precision dependends on the timing resolution of the platform's timer interrupt, tipically between 1 and 10 ms.

[1] The processing of distinct faultlets on receiving and sending allows fault injection associated to assimetric topologies.

3.2 Instructions

Faultlets are specified using a group of 29 instructions organized in 7 classes. The instructions have a textual representation, called FIRMASM, used by the assembler `firm_asm` presented in Section 5. The instruction set and its textual representation are listed below.

3.2.1 Input and output instructions

- **`READB Ry Rx` (Read byte from packet)**

  Reads in R_x the contents of the byte of the packet pointed by R_y (in bytes). If the content of R_y is greater than the length of the packet in bytes, the value of R_x is not modified.

  $R_x \leftarrow$ (8 bits) packet[R_y]

- **`READS Ry Rx` (Read short from packet)**

  Reads in R_x the contents of the short (16 bits) of the packet pointed by R_y (in bytes). If the content of R_y is greater than the length of the packet in bytes (minus one, to leave room for the remaining byte), the value of R_x is not modified.

  $R_x \leftarrow$ (16 bits) packet[R_y]

- **`READW Ry Rx` (Read word from packet)**

  Reads in R_x the contents of the word (32 bits) of the packet pointed by R_y (in bytes). If the content of R_y is greater than the length of the packet in bytes (minus three, to leave room for the remaining bytes), the value of R_x is not modified.

  $R_x \leftarrow$ (32 bits) packet[R_y]

- **`WRTEB Ry Rx` (Write byte to packet)**

  Writes to the byte of the packet pointed (in bytes) by R_y the contents of the least significant byte of register R_x. If the content of R_y is greater than the length of the packet in bytes, the instruction has no effect.

  packet[R_y] $\leftarrow$ (8 bits) R_x

- **`WRTES Ry Rx` (Write short to packet)**

  Writes to the short (16 bits) of the packet pointed (in bytes) by R_y the contents of the least significant short of register R_x. If the content of R_y is greater than the length of the packet in bytes(minus one, to leave room for the remaining byte), the instruction has no effect.

  packet[R_y] $\leftarrow$ (16 bits) R_x

- **`WRTEW Ry Rx` (Write word to packet)**

  Writes to the word (32 bits) of the packet pointed (in bytes) by R_y the contents of the register R_x. If the content of R_y is greater than the length of the packet in bytes (minus 3, to leave room for the remaining bytes), the instruction has no effect.

  packet[R_y] $\leftarrow$ (32 bits) R_x

- **`SET y Rx` (Set a value into a register)**

  Register R_x is set with the absolute value y.

  $R_x \leftarrow$ (32 bits) y

3.2.2 Logic and arithimetic instructions

- **`ADD Ry Rx` (Add register)**

  Register R_x receives the value of the arithmetic addition of registers R_y and R_x.

  $R_x \leftarrow R_x + R_y$

- **`SUB Ry Rx` (Subtract register)**

  Register R_x receives the value of the arithmetic subtraction of registers R_y and R_x.

  $R_x \leftarrow R_x - R_y$

- **`MUL Ry Rx` (Multiply register)**

  Register R_x receives the value of the arithmetic multiplication of registers R_y and R_x.

  $R_x \leftarrow R_x \times R_y$

- **`DIV Ry Rx` (Divide register)**

  Register R_x receives the value of the integer arithmetic divison of register R_x by the contents of register R_y.

  $R_x \leftarrow R_x \div R_y$

- **`AND Ry Rx` ("AND" bitwise register operation)**

  Register R_x receives the value of the "AND" bitwise logic operation of registers R_y and R_x.

  $R_x \leftarrow R_x \,\&\, R_y$

- **`OR Ry Rx` ("OR" bitwise register operation)**

  Register R_x receives the value of the "OR" bitwise logic operation of registers R_y and R_x.

  $R_x \leftarrow R_x \mid R_y$

- **`NOT Rx` (Invert register)**

  Register R_x receives the value of the logical bitwise invertion of register R_x.

  $R_x \leftarrow !\, R_x$

3.2.3 Packet action instructions

- **`ACP` (Accept packet)**

  Packet is accepted. Faultlet processing is concluded.

- **`DRP` (Drop packet)**

  Packet is dropped. Faultlet processing is concluded.

- **`DUP` (Duplicate packet)**

  Packet is duplicated. Faultlet processing is concluded.

- **`DLY Rx` (Delay packet)**

  The packet is delayed for the number of milliseconds given in R_x. Faultlet processing is concluded.

3.2.4 Branch instructions

- **`JMP x` (Unconditional branch)**

  Faultlet's execution flow is branched to the instruction located at positon x of the faultlet.

- **`JMPZ Ry x` (Branch if zero)**

  Faultlet's execution flow is branched to the instruction located at positon x of the faultlet if the content of register R_y is zero.

- **`JMPN Ry x` (Branch if negative)**

  Faultlet's execution flow is branched to the instruction located at positon x of the faultlet if the content of register R_y is negative.

3.2.5 Auto-increment manipulation instructions

- **`AION Rx Ry` (Auto-increment activation)**

  Activates the auto-increment for register R_y, which increments it periodically using the content of R_x as a period (in milliseconds).

- **`AIOFF Ry` (Auto-increment deactivation)**

  Deactivates the auto-increment for register R_y.

3.2.6 String manipulation instructions

- **`CSTR Ry Rx "string"` (Compare string)**

  Compares the content of the packet starting at position indicated in R_y to the string `string`[2]. The value '1' is written to R_x if they are equal, '0' otherwise.

- **`SSTR Ry "string"` (Writes string)**

  Writes the string `string` to the packet, from the position indicated in R_y to the end of the packet or of the string, whichever occurs first.

3.2.7 Miscelaneous instructions

- **`MOV Ry Rx` (Copy register)**

  Register R_x is set with the value stored in R_y.

- **`RND Ry Rx` (Random number)**

  Register R_x is set with a random value which has a modulo between 0 and the value stored in R_y.

  $R_x \leftarrow\ z$, such that $-R_y < z < +R_y$

- **`SEED Rx Ry Rz` (Pseudo-random number seed)**

  FIRMAMENT uses an algorithm based on the Tausworthe generators to obtain pseudo-random numbers. The default seed value is chosen automatically, but it can be explicitly expressed using this instruction, indicated through the three registers passed as parameters (R_x, R_y e R_z).

[2] String length is limited to 255 characters.

- **`DBG Rx "string"` (Debug aid)**

  Prints to the event log the `string` string, using the contents of R_x to replace its first occurence of the scape sequence in the `printf` format (e.g., `%d` for a decimal representation, `%x` for an hexadecimal representation and so on).

- **`DMP` (Dump packet)**

  Dumps (prints) the full packet contents in hexadecimal form through the system log.

- **`VER Rx` (Virtual machine version)**

  Register R_x is filled with the version number of the running FIRMVM virtual machine. The textual representation of the version number, expressed in the form $A.B$ (where A represents the major version and B the minor), is stored in the register in the form $(A << 16) + B$.

  $R_x \leftarrow$ version(FIRMVM)

4 Watchdog mechanism

The FIRMVM register machine is *Turing-complete*. As such, faultlets executed over it can put the system into an infinite loop, causing the fault injection node to crash. Since it is impossible to avoid this, FIRMAMENT has a mechanism to interrupt such situations, and do not permit the faultlet processing function to freeze.

A watchdog mechanism detects fuctions that take excedingly long to complete, interrupting their execution. In this case, when reaching this time limit the packet is accepted and continues to flow through the protocol stack as is (i.e., if the faultlet alters the packet, these changes are kept). The default timeout of the watchdog mechanism is 20 milliseconds, but it can be modified (or even disabled) through the `settimeout` control command presented in Section 6.

This feature, beyond offering an extra protection to the user regarding the use of ill-constructed faultlets, allows the tool to be used in soft realtime aplications, since it is then possible to limit the tool's maximum intrusiveness through an upper limit to the time it takes to perform the fault injection and packet processing.

5 FIRMASM assembler

FIRMAMENT does not directly decode the literal form of the instructions presented in Section 3. Instead, it decodes instruction in a binary format which is more appropriate to be processed in kernel space. This requires a translation, or assembly, of the literal form into the binary format.

`firm_asm` is the FIRMAMENT assembler. When invoked, it receives an input file as parameter, which should be described using the FIRMASM language, as well as an optional parameter indicating the output filename. If the second parameter is omitted, the default value of `fa.out` is assumed. This parameter can indicate directly one of the virtual files existing in the `/proc/net/firmament/rules` directory, described in the next section, and this prevents the creation of intermediary files.

The assembly utility verifies parameter typing of the instructions, preventing malformed *faultlets* from being loaded into the injector. This permits some simplifications to the kernel-

internal implementation part of the instruction decoder, which provides savings in complexity and intrusivity during experiment runtime.

The tool also support labels: offset indicators which can be used as parameters to the branch instructions. The definition of a label can contain up to 10 characters, it should always be located before an instruction and finish with ':' character. A reference to the label should ommit this character. Also, both labels as well as instructions are case insensitive.

String manipulation instructions allow the user to indicate non-printable characterers (or any other ASCII character[3]) by its octal or hexadecimal representation by escape sequences. The complete listing of special characters supported can be seen on Table 1.

Table 1: String escape codes

`\a`	bell (alert)	`\t`	tab
`\b`	backspace	`\v`	vertical tab
`\f`	form-feed	`\\`	backslash
`\n`	new-line	`\`*ooo*	octal number
`\r`	carriage return	`\x`*hh*	hexadecimal number
`\"`	double quote		

The instruction which loads absolute values into registers (`SET`) can interpret hexadecimal or decimal numbers. For hexadecimal representation, numbers must be indicated with a preceeding "`0x`" and always considered unsigned. For decimal, either positive or negative numbers can be used.

Faultlets can be commented using the semicolon charecter (';') at any point in the input (except inside string sequences, in which case it is considered part of the sequence). All text in the same line following this character is ignored.

There is also a complimentary tool, called `msa_mrif`, which allows the conversion the other way, "unassembling" faultlets from its binary representation into the mnemonic format. This tool allows the inspection of faultlets loaded at the virtual files of a rule located at `/proc/net/firmament/rules` or the plain debugging of files assembled by the `firm_asm` assembler.

6 Virtual files

FIRMAMENT is operated through the manipulation of virtual files existing at the `net/firmament` directory of the `proc` file system. There are two types of files. Faultlets for each packet flow are written to and read from rule files located at the `rules` directory, in the assembled format generated by `firm_asm`. Control commands are passed to the fault injector through the `control`.

- `/proc/net/firmament/rules/ipv4_in`

  Receives and stores the faultlet which is run whenever an IPv4 packet is received by the host.

- `/proc/net/firmament/rules/ipv4_out`

  Receives and stores the faultlet which is run whenever an IPv4 packet is sent by the host.

[3]Except the null character (code 0). This is a limitation of the assembler, not of the interpreter.

- `/proc/net/firmament/rules/ipv6_in`

  Receives and stores the faultlet which is run whenever an IPv6 packet is received by the host.

- `/proc/net/firmament/rules/ipv6_out`

  Receives and stores the faultlet which is run whenever an IPv6 packet is sent by the host.

- `/proc/net/firmament/control`

  Receives commands which control the fault injection tool. Commands can be passed to the tool through the `echo` utility. For example, to execute the `command` command, the user should type:

  ```
  echo "command" > /proc/net/firmament/control
  ```

  Commands currently supported by the tool are:

  - `startflow {`*`flow`*`|all}`
    Starts faultlet processing for *flow* (`ipv4_in`, `ipv4_out`, `ipv6_in`, `ipv6_out`) or for every flow (`all`).
  - `stopflow {`*`flow`*`|all}`
    Interrupts faultlet processing for *flow* (`ipv4_in`, `ipv4_out`, `ipv6_in`, `ipv6_out`) or for every flow (`all`).
  - `showregister` *`flow register`*
    Shows, through the event log, the contents of *register* associated with *flow*.
  - `settimeout` *`value`*
    Modifies the watchdog timer value of the packet processing function to *value*, in milliseconds. Setting it to 0 (zero) disables the watchdog timer[4].
  - `reset`
    Restarts the tool, stopping all message flows, clearing all registers and interrupting auto-increment, unloading configured faultlets and setting the watchdog timer value to the default value (20 ms).
  - `version`
    Displays, through the event log, the version of the FIRMVM virtual machine in execution.
  - `wdverbose {yes|no}`
    Configures wether the display of warnings from the watchdog mechanism should be detailed (`yes` argument) or short (`no` argument). The detailing includes the exhibition of register contents associated to the flow which triggered the watchdog.

[4]This setting must be used with care, the user assumes any responsability for the crash of the fault injection node caused by a faultlet which puts the FIRMVM virtual machine into an infinite loop.

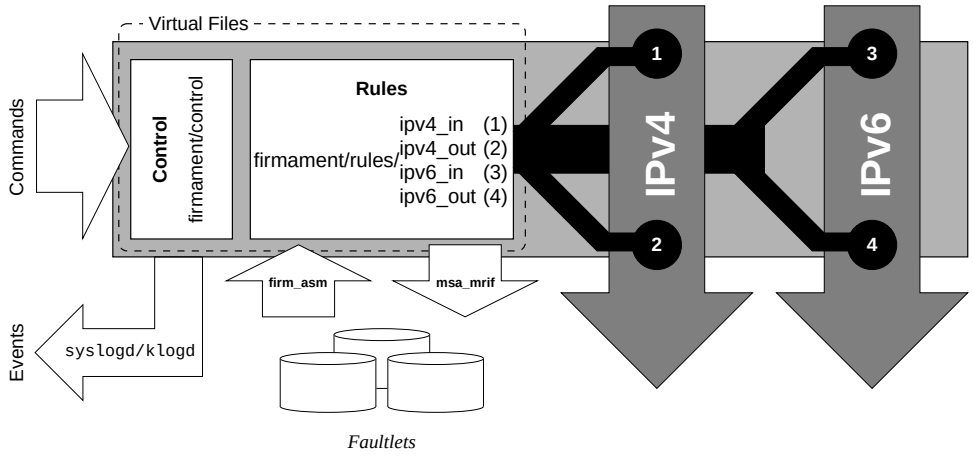

Figure 1: Conceptual relation of FIRMAMENT's components

7 Logging

A fault injection campaign is of limited utility if there is no possibility to record the events associated to fault injection and to the operation of the injection tool. This is necessary to the future analysis and to relate the generated events with its consequences in application behavior and fault tolerance mechanisms, so that conclusions regarding its dependability can be reached.

ComFIRM uses a virtual file at the `proc` filesystem to log events associated to the tool operation and the fault injection activities. This mechanism captures the events and formats them to exhibition when the file is open for reading by any utility, which becomes responsible by the experiment monitoring. However, the way this mechanism was implemented is somehow limited, since the number and size of the events logged is fixed, and events generated while the file is not open for reading are not logged.

FIRMAMENT uses a much simpler approach, using the standard `klogd(8)` mechanism for capturing system messages. The kernel's message buffers are read from the `/proc/kmsg` file by the `klogd` utility, which forwards the messages to `syslogd(8)`. It can store these messages localy in files or make a log in remote machines, which is usually useful in distributed experiment situations. Also, these utilities support event priorization. Informative messages can be distincted from critical or debugging messages, as the ones created by the `DBG` instruction of the injector. It is possible to capture only messages from above a configured level, avoiding the overhead of logging too many unimportant messages.

Figure 1 presents the conceptual relation between FIRMAMENT's virtual files, commands and *faultlets*, as well as the location where the injector intercepts the communication. The numbers associate the rule virtual files with the corresponding interception points.