Internship Report
A Software Defined Rule System for WiFire

Markus Engel

March 24, 2011

Contents

1 Introduction

A firewall is conceptually a device which is placed at the borders of networks and which controls the crosstraffic between the two (or more) networks. In wired systems, a firewall is equipped with two or more network devices which are connected to the different networks. Packets crossing the network boundaries are stored and either forwarded or dropped, depending on administrator defined rules.

This traditional firewall concept is no more applicable in wireless systems. In wireless systems there is no physical wired connection, so there is no such network border. Moreover, everyone in a specific transmission range has access to the medium and can intercept every packet or even inject own packets. Therefore, if we want to have a firewall for wireless systems, we have to redefine the traditional understanding of a firewall and its behaviour.

There is little to do about the fact that everyone has access to the medium, so the idea of a wireless firewall is to jam all unwanted traffic, such that it is no more received by the clients which should be protected. The inherent challenge is that this system must come to a verdict while the packet is still on the air, thus a primitive store-and-forward principle is not applicable in this setting.

The WiFire System is such a wireless firewall considered and realized by the distributed computer systems lab (DISCO). Its implementation utilizes the USRP [3], a low cost software defined radio. The devices captures all traffic, inspects it and if it comes to the end that the traffic is not legitimate, it is jammed, so the endpoints of the network will discard it since the checksum won't match anymore.

At the time I started my work in the project, a lot of work was already done by others. The implementation was able to receive and decode IEEE 802.15.4 [6] frames [2, 1], as well as generating and sending a jamming signal. It was even possible to classify the packets and do the jamming [7] selectively by rules [2].

But, one thing which was missing was reconfigurability. Clearly, the rules could be changed, but since this was implemented in hardware (or more specific in synthetisized hardware), a change in the rules needed a run of the synthesis tool—which takes about half an hour. So it was my task to have a rule system which was more flexible and easier to administrate.

In the following report, I will explain the concepts and some implementation aspects of the rule system. Furthermore I will provide some performance measurements and conclude with an outlook.

As a last note in the introduction, I will now clarify some terminology to circumvent misunderstandings. Even if I wrote about "packets" in the above paragraphs, I will now use the term "frame" since most sections will refer to IEEE 802.15.4 frames. Another trap could be the word "reconfigurability". This term often refers to a runtime reprogramming of FPGA chips and therefore a modification of hardware behaviour. Being a very interresting topic, this is not meant in the context of this report. Reconfiguration essentially means only to alter configuration variables of the software or configuration registers of the hardware which control the behaviour of the otherwise fixed code or hardware.

2 Towards a reconfigurable Rule System

At the time I began my work and got comfortable with all the hardware and technical details, a paper was about to being published by the working group. For that paper, experimental results were needed and I was requested to setup some scenarios which were considered for the paper. These scenarios requested to inspect multiple fields of the header of an IEEE 802.15.4 frame as well as matching physical values such as signal strength and time durations.

To get all this working, I considered a rule system which should meet some requests. Most of the work was done in hardware, because at that time it wasn't clear whether evaluating rules in software was fast enough and we had little time to the deadline for the paper. I will now discuss the features which were needed and how they were realized in the modules.

2.1 Implementation

As already mentioned, the implementation was mostly done in hardware, or more specific in Verilog. Hardware Description Languages such as Verilog offer the possibility to group related behaviour into components, called "Modules" in Verilog. Modules can instanciate other modules and their interfaces are connected by user defined wires, therefore modules provide an hierarchical structure.

For simplicity (and to follow some conventions), all inner complexity was hidden in modules, refining the overall behaviour. Figure 1 shows the inner structure of the new top level module WiFire.

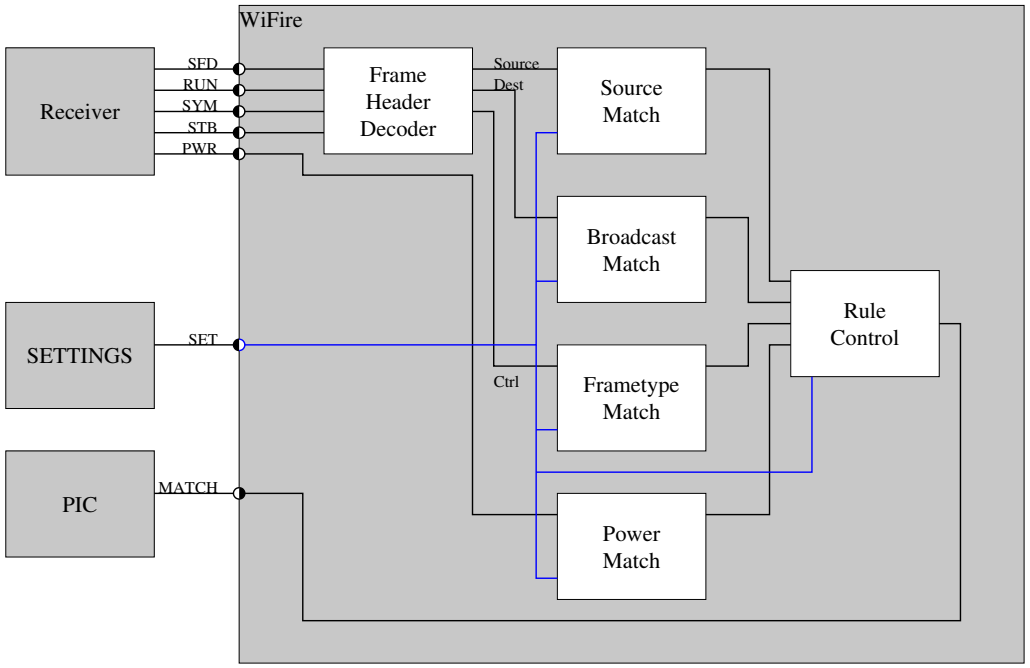

Figure 1: WiFire Module

The individual modules will be explained in more or less detail in the following. As most of these modules were not considered to be part of a long time solution (and were discarded after the paper was done), I will not go very much into detail here.

2.1.1 Header Decoding

A lot of decisions whether a frame is considered malicios or not depend on the contents of the header. Thus decoding the individual fields is a central component and was therefore realized in a Verilog module.

This was relatively easy to accomplish, since the frame decoding procedure can be described by a finite state machine. Hardware Description Languages are somewhat predestinated to realize such finite state machines. In fact, programming in these languages is just about describing data and control paths (on a register-transfer level to be more precise). For decoding a header, consider you are in some state (e.g. `STATE_LENGTH`) and you get some data byte now. Then you can just save these data in a corresponding register (`length` in that example) and advance to the next state, according to the frame header format defined by the IEEE 802.15.4 standard.

At the end of the header the decoder just signalizes that the header is decoded (and therefore the register values are valid and up to date) and waits for another frame, which is signalized by the receiver with a Start-Of-Frame signal.

2.1.2 Reconfigurable Matches

For the scenarios, several matches were needed. A match essentially describes a property and yields a boolean value whether that property holds for a frame in question or not.

Most matches had in common, that their corresponding property had to be parameterized. Consider for example a source address match, then it would be nice not to just match one specific hard wired address but an address which should be reconfigurable at runtime.

The realization of that reconfigurabiltiy was pretty much straight forward since the USRP Core provides a settings bus, which is connected to the wishbone bus. The purpose of this bus is to set hardware register values from the firmware.

For the scenarios, different matches were realized. Some of them, such as a power threshold match did also have to measure time. Since time cannot be expressed by purely combinatorical logic, these matches had a sequential behaviour.

The matches which were implemented, are

- Source Address Match
- Destination Broadcast Match
- Frame Type Match (Data, Ack, Beacon, Control)
- Power Threshold Match
- Power Hysteresis Match

The matches were realized in modules each. Their input were parts of the decoded frame header and the settings bus itself. The output was just a signal of a boolean type indicating whether the match did actually match the current data of the decoded frame header.

Using this setup, it was possible to just send some UDP packet containing all the configuration data, extracting these data on the USRP side and feeding them to the corresponding match via the settings bus.

2.1.3 Reconfigurable Rule

Since some scenarios needed multiple matches for a frame, the ability to enable and disable each match independely and at runtime was inevitable. Again, it should be possible to just send some message to the USRP to reconfigure the rule. At that time it was sufficient to have just one rule, such that all enabled matches could be linked together by one big logical `AND`.

For the implementation, a new module was introduced. The inputs of this module were the outcomes of the matches, e.g. some wires yielding a boolean value indicating whether the corresponding match matched or failed. The module also had a settings register, connected to the settings bus. The value of that settings register just indicated whether the outcome of a match should be taken into account or not. Essentially this can be done by a logical implication.

Thus the outcome of the whole rule was just combinatorical logic, using a function of essentially the following style:

```
assign output =
  (!sourceEnabled | sourceMatches) &
  (!broadcastEnabled | broadcastMatches) &
  ...
```

2.1.4 Putting it all together

The missing piece of the rule system was some glue to bring all these modules together. For that reason, another top level module was added, called WiFire. This module connects to the output of the receiver which include the Start-Of-Frame signal as well as the decoded symbols.

On the inside, the module reassembled the symbols to bytes and instanciated the frame header decoder module, passing the data bytes in sequence. The output side of the frame header decoder was passed to the matches, which were also instanciated in the WiFire module. In the end, the rule system module got all the outcomes of the matches and provided just one output wire, telling whether the rule as a whole applied or not. This output was propagated as an interrupt to the processor and therefore into the firmware which was initiating the jam.

On the host side, the existing wifireadm script was heavily modified to support all the reconfigurability of the matches and the rule. Clearly, the firmware had to be adjusted as well to get the values from the UDP packet and forward them to the hardware.

2.2 Revision

In fact, a lot more work was done. First, there were a lot of setbacks due to the nature of the receiving hardware or self introduced bugs. In order to keep the number of bugs (and therefore the number of very time-consuming resyntheses of FPGA code) as low as possible, every module was added its testbench to validate its behaviour by simulation. Of course, by just giving some input stimuli to the module and examing the output, no complete test could be done, so unfortunately some modules still produced unwanted behaviour which occured for the first time when running on the hardware. A formal verification by property checking would have been my preferred way here, but since the corresponding tools are extremely expensive and it would have been somewhat overkill for our needs, a mixture between simulation and try-an-error was enough.

Second, we needed the ability to see what was really going on, especially regarding the power level. On one hand, we had to get some values for tuning an appropriate threshold and on the other hand, we also could find out some hardware flaws in the receiver circuit of the USRP hardware. For that purpose, the WiFire module was also connected to the wishbone bus, so we could easily talk back values originated by the hardware into the firmware. The firmware itself could process these data or just relay them to the host using UDP datagrams.

The design of all the modules allowed to have some degree of freedom considering the reconfigurability of the firewall. At least it was no more necessary to resynthesize all the FPGA Code to alter matches or the interaction of matches.

The overall system could be used to perform the experiments needed for the paper. Unfortunately, the success of these were satisfying to some extend only, but failing experiments could often be ascribed to the non high-end hardware we used here.

But in the end, I was very unfortunate with that solution, since it was still a very long way to go to my personal target. To face it, the WiFire system had only one more or less configurable rule. Due to that fact, it was not possible to e.g. match two independend source addresses or more complex patterns for rules.

3 A Software Defined Rule System for WiFire

To get towards reconfigurability, one has to abstract from hard wired hardware designs (or hard coded FPGA code) and move to more general purpose hardware which can be used by a controller in a sequential way, processesing a list of instructions. The most general pecularity of that abstraction is a processor, which processes a sequence of machine instructions. The USRP includes such a processor—even if it is not a single piece of hardware but is implemented itself in Verilog, thus forming a so called "softcore".

My initial idea for the WiFire system was to have a rule system which had all its configuration and functionality residing in the firmware which runs on the processor.

In the following, I will define the capabilities and concepts of the rule system and how it is actually implemented.

3.1 The Rule System

Most concepts of the rule system are adopted from the netfilter rule system of the Linux Operating System, better known as iptables. This system defines entities such as tables, chains, rules, matches and targets. Each of these entities are given a semantics. I will now introduce these concepts in a (mostly) bottom-up direction.

Match A match could be described as a function, taking a frame and returning a boolean value, indicating whether a specific property of the frame holds or not. The property in question should be as atomic as possible, since the individual match is more reusable then.

If we were programming in a functional language, we could just say that we had one (lambda) function to match address 1, one function to match address 2, and so on. Since this would become very unhandy in e.g. the language C, another parameter is given to the function, which points to a match-specific structure containing match-specific data. Using that mechanism, examples for matches would be the source address match, destination address match, frame type match and so on.

Target A target is either a final verdict or some special action.

A final verdict is the final outcome of an evaluation of a frame in the whole rule system. It is either `ACCEPT` or `JAM` (in our case). The meaning of these verdicts should be inherently clear. Whenever such a final verdict is made, the whole evaluation process is done and can immedately return.

Note, that a final verdict is just some kind of value, not an action itself. This value is returned by the evaluation function and then the overall system will jam the signal if appropriate. However, this behaviour could be changed if needed.

Other special cases are `JMP <target>` to jump into another chain (see below) or `RET` to return from a subchain.

It is also possible to specify functions as targets, which defer the actual target. These functions get the packet and return another target or they can do special actions (such as logging) and tell the rule system that the packet is handled completely.

Rule A rule consists of an ordered list of instances of matches and an instance of a target.

At least in our case, every match of a rule can be inverted independently, essentially negating the boolean outcome of the match function. Negation can also be achived by using subchains and applying De Morgan's law to the language of the rule system. Sometimes this method is more appropriate, sometimes it is just overkill—Your mileage may vary.

The semantics of a rule is described like this: when evaluating, the rule is given a frame. Then each match of the rule is evaluated with the specific frame. If all matches apply, then the whole

rule applies and the outcome is the target of the rule. If at least one match does not apply, the rule does not apply and thus there is no outcome.

From another point of view, we could say that all matches of a rule are combined by logical `AND`. From techniques which are applied in compiler construction, we can reuse the concept of early evaluation exit here: if the matches are evaluated in sequence and we find the first match which does not apply, we can directly deduce that the whole rule won't apply and therefore we can stop at that point and go on to the next rule. For that reason it is reasonable to spend some time on the order of the matches: the more frames will be failing on a specific match, the sooner that match should be evaluated. This general rule is also true for iptables, but in our case another criterion has to be thought of which will be explained later in section 5.2.

Chain A chain consists of an ordered list of rules and a policy. Conceptually we could classify two types of chains: the toplevel chains and subchains.

The semantics of a toplevel chain is just a final verdict: when evaluating a chain for a given frame, the chain will evaluate all its rules in order until it finds the first one which applies and returns a final verdict. Then the outcome of the overall evaluation is just that verdict. If no rule applies at all, the policy (which is actually a final verdict) will be the outcome.

A subchain in turn can be called by any rule through a special jumping target. Evaluating a subchain can yield a final verdict (then the overall evaluation is done) or can return to the parent chain if a rule matches and its target tells to return. The default policy of subchains is always to return, so if no rule of a subchain matches at all, evaluating the parent chain is continued at that point.

It should be clear, that despite the fact that the syntax allows for loops (jumping into a parent chain), this has to be disallowed by static semantic checks. A loop in a chain could potentially lead to an endless loop in the evaluation function and is anyway never useful. With that assumption in mind, we can always represent the whole configuration of chains, rules and matches as an acyclic graph. We can also conclude, that each chain is at most once part of an evaluation and therefore each chain is in use at most once at each point in time. Note, that this is no longer true for multicore systems—however, the USRP is still a singlecore system.

Table Since iptables can classify incoming, outgoing and forwarded packets, it has a toplevel chain for each of these packet flows. These toplevel chains are encapsulated in a table. There are several tables, which all have special meanings and which are hooked at different points into the TCP/IP stack into the kernel, e.g. before and after routing a packet.

Since the WiFire system only receives frames, there is only one toplevel chain and no need for tables.

3.2 Implementation

In every rule system, performance is the most important topic. In traditional firewalls, the throughput has to be very high while keeping the load on the processing hardware low to provide some scalability. Especially when moving to 10 GBit/s links, there a lot of packets to be processed per time unit.

The very central difference for our Wireless Firewall is the fact that we have to get to a verdict while receving the frame. In contrast, traditional firewalls could store the whole packet, examine it and forward it if necessary. Indeed, commercial systems also try to evaluate as much as possible while the packet is in transit, but in our case this is mandatory.

That means, that all implementation design, such as involved data structures and algorithms have to be done in a way that traversing all the chains, rules and matches is highly optimized and takes as little resources as possible.

Another desirable property for getting towards higher speeds would be to do optimization on the rules. Ideas for that can be found in section 5.6.

3.2.1 Firmware

The actual implementation of the WiFire rule system follows again pretty much the implementation of iptables. The data structures are desinged to have as little pointer indirections as possible. Therefore all match instances are embedded into the parenting rule and all rules are embedded into the parenting chain. From the view of memory, "embedding" means to put e.g. the list of matches into the rule instead of having a pointer in the rule pointing to the list of matches. Since the number of matches can vary, this means of course that the size of a rule is no more constant. Likewise the size of a chain is not constant, because rules are enclosed in the chain itself.

However, the configuration data of match instances are not embedded into the match instance, but there is a pointer to the corresponding data. This indirection is reasonable, because for evaluation the match, there is already a call to the corresponding match function involved, so a pointer access will not add that much delay here.

The evaluation function itself can be done in different ways. Since we have the concept of chains and we can jump into chains in an (conceptually) arbitrary deep nesting level, we have to reflect that concept somehow in the implementation. In the following, three different implementation styles are described.

Recursion The easiest way to implement this is to use recursion. In that case, all local variables (such as the current chain and the current rule) is stored on the program stack implicitly by using function calls. The drawback is that really every local variable is stored, even if this is not needed in our case. Another drawback is the function call itself, which takes quite some time because of running function prologues when calling and epilogues on return.

Explicit Stack The second idea to implement this would be to have an explicit stack – some data structure we can push a pointer to the current chain and rule and pop this information when returning from a subchain.

The obvious advantage is, that this will save a lot of function calls, but it also has the advantage of a very fast function exit if a subchain comes to a final verdict. The drawback, however, is that a stack is either bound to an upper limit (which means a hard coded maximum nesting level here) or uses an underlying dynamic data structure needing some dynamic memory management such as malloc/realloc/free, which is not available on the USRP.

The performance advantage of some test scenarios is about 25%.

Embedding A third implementation idea is to use the fact, that each chain can be invoked at most once. Thus, we could reserve some space in the data structure of chains and store backlinks inside the chain itself when doing a jump into another chain.

The performance could be improved by another 5% and this alternative allows for an arbitrary nesting deepness of chains (at least there is only a constant amount of memory used for evaluation).

As a sidenode it should be mentioned, that iptables uses the second alternative, because it runs on multiprocessor systems and cannot do the third optimization here.

The implementation of the evaluation function is actually very small and mostly understandable by the interrested reader.

3.2.2 FPGA

As already mentioned in section 2.2, the modules of the previous rule system were not meant as a long term solution and therefore most of them were discarded, because their functionality had

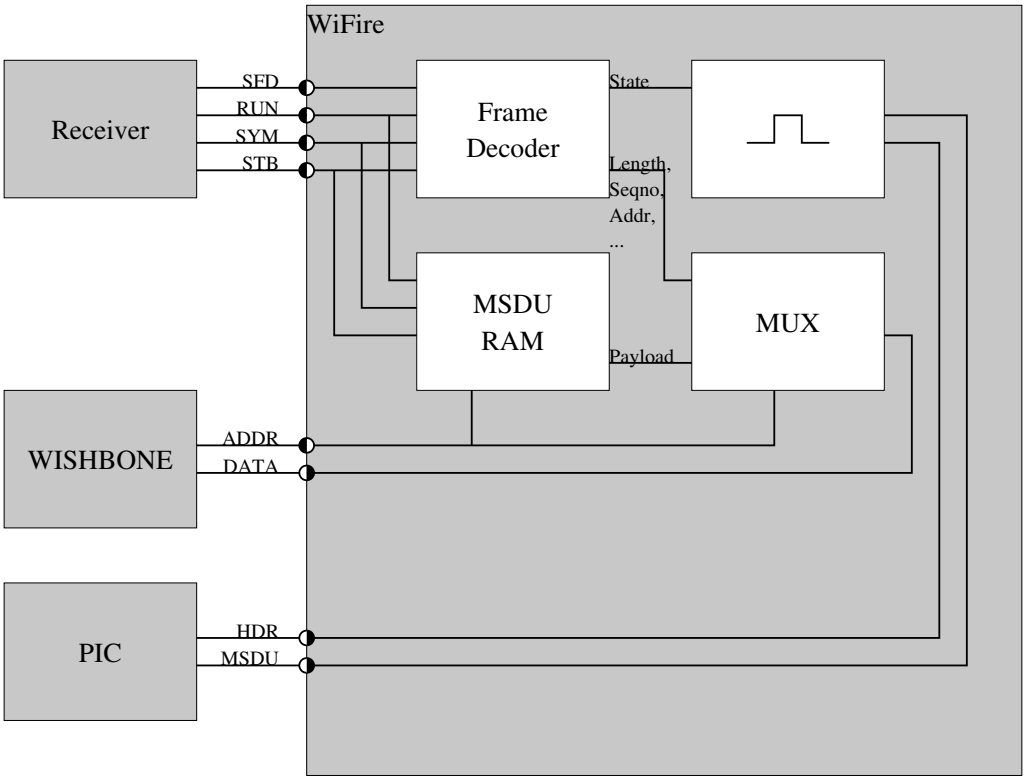

Figure 2: Component Diagram of WiFire Module

been shifted to software in the new approach. But, some of the modules proved to be very useful. Especially the gap between the receiver and the firmware had to be closed, because obviously the data of the frames is needed in order to get to a verdict.

Thus the frame (header) decoder which was described in section 2.1.1 is now reused in a slightly modified way. The task of the WiFire component was redesigned. Figure 2 shows a component diagram of the WiFire module. Essentially, it receives the symbols of the receiver on the one side, passes them to the frame decoder and hands the outputs of the frame decoder and the payload to the wishbone bus and therefore to the firmware. Furthermore it communicates state changes by raising interrupts, so the firmware can be alerted in case of a new frame.

3.2.3 The Host Side

As described earlier, the data structures are embedded to allow a fast traversal of the rules in the needed direction. Therefore, on the other hand, these data structures are hard to maintain.

For that reason, the task of building the whole configuration data was outsourced to the host. Another advantage is to have an easy to use user interface. The idea behind the program was therefore to have as little as possible to do on the USRP and do all the stuff on the PC.

The usual usage pattern with the concrete implementation is as follows:

Learning the USRP Side The USRP reserves a fixed amount of memory for the configuration of the firewall. When the host connects, the USRP tells it about the size and the beginning address of that memory. Furthermore, all available matches are communicated, using an identifier (which is known by the USRP and the Host side) and the function address on the USRP side.

Maintaining the System The rule system can be altered using commands such as `add`, `insert`, `delete` and so on, and examined by commands such as `list`. For more information about the commands, see section 5.8 in the appendix.

In the program, the GNU Readline Library [5] was used and a new parser framework was built from scratch. This allows for comfortable command line editing (such as familiar features of the Bash [4], even the very useful history features) and command completion using the `TAB` key.

Clearly, each match on the USRP side must have its partner on the host side. The task of the host side match is essentially parsing the data when the match is instanciated, keeping the data for representation and writing the data into the target memory. The most critical point here is writing the data because this must exactly match the data layout of the match on the USRP side. In order to disarm some of the obvious traps, a framework was built to do most conversions and alignments automatically.

Committing the configuration When configuration is done, the command `commit` will pack all the matches into their rules, the rules into their chains and the configuration data of the individual match instances into the configuration memory. Thereby an automatic pointer relocation and data alignment is done and data formats are converted (since the USRP uses big endian encoding and the x86_64 host uses little endian). In the end the memory is just sent to the USRP which overwrites its reserved configuration memory with the new data. The USRP does not have to postprocess any of this data but can just use it together with the evaluation function when the next frame arrives.

4 Performance

As already mentioned, performance is a very important topic in firewalls. Therefore I did some measurements and calculations to show the relations between a configuration and its memory usage and evaluation delay.

4.1 Memory usage

Clearly, configuration takes space. The overall size of the configuration is the sum of the size of all chains and subchains. These sizes in turn depend on the particular rules and their matches. Additionally, each entity adds static sizes and a bunch of alignment rules must be taken care of.

- The size of a **(sub)chain** is the size of a verdict (4 byte), the name of the chain (20 byte), the backlink pointers ($2 * 4$ bytes), the end pointer (4 bytes) and the sum of the rule sizes. That means, that a chain takes at least 36 bytes.
- The size of a **rule** is the size of its target (12 bytes), the size information (4 bytes), an end pointer (4 bytes) and the sum of the match sizes. A rule takes therefore at least 20 bytes.
- The size of a **target** is the size of its verdict (4 bytes), and the size of the jump target (4 bytes) or the size of the user defined target function (4 bytes) plus a pointer to target specific data for that function (4 bytes). A target takes therefore 12 bytes plus some target specific data when user defined target functions are used.
- The size of a **match** is the size of the function pointer (4 bytes), some flags (4 bytes), a data pointer (4 bytes) plus the match specific configuration data size. Therefore, a match takes at least 12 bytes.
- The size of the match or target specific data can vary greatly and must be aligned correctly. Therefore it's not a trivial task to calculate the overall size, but the static sizes give a good starting point.

  Fortunately the size is already computed on the host, and checked against the reported memory size, so no errors can occur regarding non-fitting configurations.

You can see some examples in the end of this section.

4.2 Evaluation Delay

The evaluation delay is the time the evaluation function needs to come to a verdict. Clearly, this delay depends on the configuration and the actual frame. In our case, only the worst case is important so we can abstract from an actual frame and take only the longest execution paths for a given configuration into account.

For these measurements, time was taken before and after the evaluation and the difference was calculated. On the USRP, time can be expressed in number of clock cycles. For that purpose, a register is incremented each clock cycle and can be read by the firmware. The clock speed is 100 MHz, so one tic corresponds to 10 ns.

Note, that the Processor on the USRP runs with the half clock speed, i.e. 50 MHz and that a single instruction could take more than one cycle. Fortunately simple calculations and even memory accesses need very little time, but you should have in mind that a cycle time of 20 ns is very high, compared to speeds we see every day when working with desktop computers.

Another thing to be mentioned is, that the Processor on the USRP does not have any caches or address manipulating units, but this is not really bad. First, this would not gain any better access time, since memory access is already at minimum clock cycle time. Second, caches always

bring some non-determinism when evaluating execution times. Since we have no caches, you can repeat the measurements and you will always get the very same results.

To get representative numbers, I tested a bunch of predefined configurations, so I could take the differences to know the delays of the individual entities of the rule system.

- A **chain** takes 100 tics to be evaluated which corresponds to 1 µs. This is the time to jump into the function, to load some working set of pointers, to see that there is no (more) rule and to finally return the policy.
- A **rule** takes 26 tics to be evaluated, which corresponds to 0.26 µs. This value was a bit tricky to get since you can define empty rules, but then this rule will apply always, which will end evaluation and therefore you will have another execution path through the evaluation code. Therefore another measurement was done which yields the difference for these execution paths, so this difference could be added to the rule delay.
- A **match** takes at least 34 tics to be evaluated, which corresponds to 0.34 µs. This is the time to call the match function (by pointer) and return from that function, then maybe negate the outcome. For that purpose a match was added which does absolutely nothing than return. An actual match will add the time to do the test on the frame. For example the address match takes additionally 152 tics, therefore an overall of 186 tics, which is 1.86 µs.
- A **subchain** takes 70 tics to be traversed, which corresponds to 0.7 µs. You can see that this is less than the chain time, since there is no further function call involved, thanks to the implementation of the rule system as described in section 3.2.1.

Having these numbers, you can easily add delays for a given configuration in a recursive way. It would be appropriate to have this calculation done by the wftables application as these delays are a very limiting factor of what is possible.

4.3 Overall delay

As a last step, I measured some more delays using the previously described cycle-counting method. These measurements should clarify the delay of the interrupt and the overall delay of WiFire.

The test data which were sent had an header length of 10 bytes, which is 20 symbols and takes therefore 320 µs to be transmitted. When this header is transmitted and decoded, an interrupt is generated which will eventually trigger the evaluation function. The decoding itself takes a negligible constant delay, so we discard it.

The time to trigger an interrupt was taken by storing the current tics in a hardware register when flagging the interrupt and by reading the elapsed tics from software when the interrupt routine was eventually fired. The time varies greatly between 3.31 µs and 4.03 µs, which is a difference of a 0.72 µs. Between this interval, the actual times seem to follow an equal distribution. The source of the variation is a part of the main loop of the firmware, where interrupts are disabled. The difference corresponds to 36 processor cycles which seems quite reasonable for the code in that interrupt disabled block. Besides the variation, the minimum measured time of 3.31 µs is quite high. 0.66 µs are needed for calling the interrupt handler and executing the prologue of the handler. The handler itself runs a loop which executes the appropriate handler of the triggered interrupt. This takes the largest part of the time, but can be optimized in several ways. By just unrolling the loop (which can be done by just setting a compiler flag), the minimum time can be reduced to 1.83 µs which is 1.48 µs faster than the untouched code. Another optimization would be to change the implementation of the interrupt controller (resp. use parts of the PIC which are unused in the current implementation) such that the loop can be removed completely.

After the interrupt is triggered, the evaluation begins, so the evaluation delay is added. After that the jamming routine is called (if the verdict told us to jam) which adds another delay of 0.41 µs.

Thus the WiFire worst case delay adds up to

$$\begin{aligned} t_{\text{delay}} &= t_{\text{intr\_worst}} + t_{\text{eval}} + t_{\text{react}} \\ &= 4.03\,\mu\text{s} + t_{\text{eval}} + 0.41\,\mu\text{s} \\ &= 4.44\,\mu\text{s} + t_{\text{eval}} \end{aligned}$$

This is only the delay which could be measured by counting cycles. There are additional delays, given by the receiver and the transmitting hardware (such as the DSP and other components in the TX-chain) which are not taken into account here.

4.4 Examples

Table 1 gives an overview of three scenarios with their corresponding worst case evaluation delay, worst case WiFire delay and the size of the configuration in memory. Figure 3 shows the worst case delay for different rule and match configurations. Note that in that figure only the minimum match delay was taken, so for actual matches the slopes are much steeper.

Scenario	t_{eval}	t_{delay}	s_{config}
No Rules	1 µs	5.44 µs	36 byte
1 Rule, 1 Address Match	3.12 µs	7.56 µs	80 byte
10 Rules, 1 Address Match each	22.2 µs	26.64 µs	476 byte

Table 1: Delays

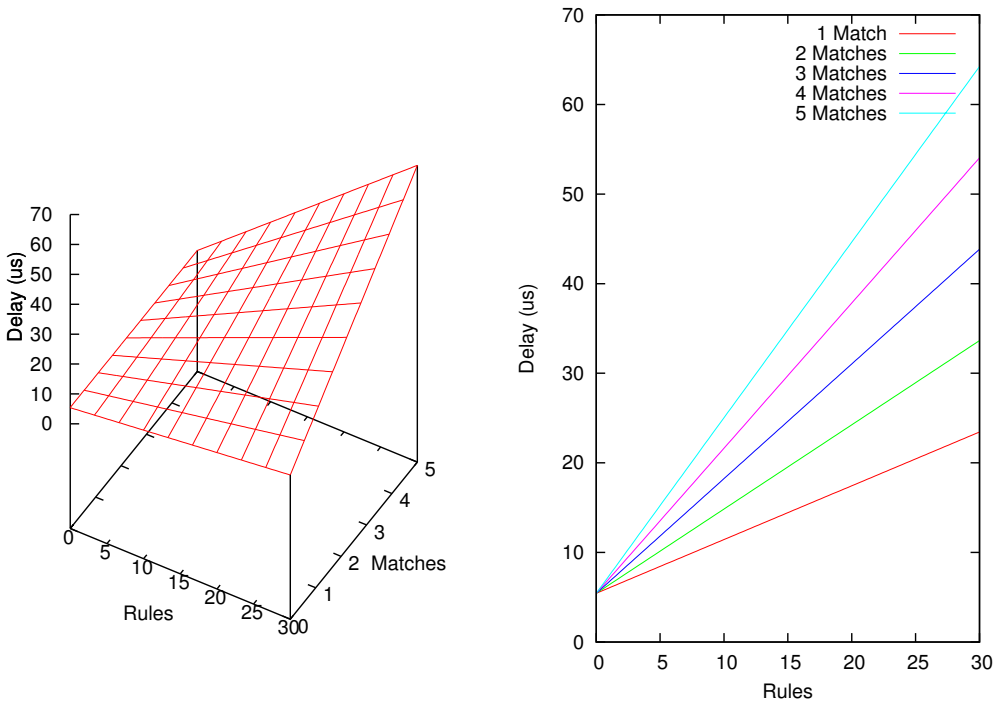

Figure 3: Delays

5 Future Work

The implemented rule system is far from being perfect. In fact, it can be considered as some generic framework where matches can be embedded. That said, a variety of dimensions arises, in which the rule system can be expanded.

The following sections will discuss some shortcommings of the current implementation and will give an overview of possible extensions.

5.1 More Matches

By the time of writing, only a few matches such as the address and frame type were implemented. In order to have some more sophisticated rules, more matches are needed. Some of them can be implemented without changing the framework, but a lot of them also need the rule system to be changed.

- **Length** This match formulates a property on the length of the frame. One can think of expressions like "at least X bytes" or "at most X bytes". Another thought should be spend on which length to match actually. We could always take the frame length or the payload length. The payload length cannot be calculated by just subtracting a constant number from the frame length, since the header length varies greatly. Considering performance, it would be best practise to calculate the payload length in the frame decoder in hardware.

- **Security** The current frame decoder is not able to deal with the security extension header of IEEE 802.15.4. Thus we cannot formulate any match concerning security (except for a match which could tell whether security is enabled). For these matches, an extension of the frame decoder should be considered.

- **Address Range** The range match is maybe known from iptables. With such a match, we could restrict the allowed address range in a PAN.

- **Broadcast** This match could be used to prohibit broadcast communication. In IEEE 802.15.4 you should also distinct broadcast traffic inside a PAN and broadcast traffic crossing PAN boundaries. Both types should be matchable.

- **Statistic** We can also think of statistical matches which could try to keep a certain data rate or even match according to random distributions.

- **Unclean** This match will identify all frames which do not adhere to the IEEE 802.15.4 specification. This is whenever reserved bits do not have the defined values or invalid bit combinations are set. As best practise, the frame decoder should be able to identify those frames and set a value which is readable in the firmware. On one hand, these frames also confuse the current implementation of the frame decoder, so header fields could be misinterpreted. On the other hand, these frames could also confuse the legitimate clients, so it would be appropriate to jam these frames.

  So, the aim of this is twofold: make WiFire more robust against malformed frames and give WiFire the ability to jam these.

- **Payload** By now, only matches concerning the header of a frame are possible. Inspecting payload is more challanging since the evaluation time point would be shifted then. Section 5.2 will discuss some aspects for these matches.

- **Physical** A frame is more than the concatenation of its bits. In the wired world we could for example distinguish the network device which received the frame. In the wireless world, we can measure physical values such as signal strength. If it should be possible to receive

from two antennae (maybe with another hardware or with multiple USRPs), we could also distinguish which antenna received the signal. For these things, one has to define a evaluation time point, since most of these values refer to a infinitesimal time range, so we could for example take the signal strength which was measured at SFD or we could take the integral of a longer time range when doing the evaluation of a rule.

5.2 Matching Payload

Deep Packet Inspection (DPI) is a very important topic for firewalls. Especially in IEEE 802.15.4 there are a lot of frames for controlling the network. All these frames are marked as "control frame" in the header, but their individual (and very different) meaning is encoded in the payload. Furthermore if we assume ZigBee being used in the next layer, there are a lot more different messages exchanged which can only be distinguished when looking into the payload of the frame.

This has a direct impact on the evaluation time point. Keep in mind, that we have to come to a verdict while the frame is still on the air. Thus, if we need for example the 30^{st} byte of payload for a match to get to its verdict, we would have to wait all the time until this byte arrives.

Things get worse if you consider two rules: the first one needs 30 bytes to evaluate, the second one just 5 bytes. Fortunately we know the length of a frame, so if in the above case a frame with 15 bytes payload arrives, we know that the match which needs 30 bytes will not apply, so we can jump over that rule. If another frame with 30 bytes arrives and the first rule will not match but the second does, we know that we have still enough time to jam it (since the payload is followed by the checksum). To be honest, things are not that easy, because we have to consider that each match can be inverted, so we have to take that into account when jumping over rules.

It is obvious that the evaluation time point will depend greatly on the ordering of the rules and the frame itself.

Another possiblity would be to define two (more or less) fixed timepoints to do evaluation: first one after the header is received, second one at the end of the payload (right before the checksum). If we could not get to a verdict after the header, we would defer the verdict to the second evaluation time and get to an end there.

5.3 Move Parts into Hardware

The main aspect why the rule system was moved to software was reconfigurability. The drawback, as seen in section 4, was the delay which was introduced. There are several ways to improve performance again by porting some software parts into hardware again. Some of these possibilities will partly give up reconfigurability.

5.3.1 Specialized Co-Processor

A very sophisticated solution would be to reimplement the whole rule system in hardware, that means to have some special-purpose processor which can evaluate rules which are kept in some memory. Since the processor is only used for that task, its atomic operations can be much more specialized than the operations for general-purpose processors, which yields an enormous performance advantage.

The matches would be translated to modules which get a relevant part of the frame and a configuration. This configuration is fed by the rule system, very much the way it is implemented now in software. The advantage would be that a hardware match could do a lot more things in parallel than the software counterpart and thus gain a big performance advantage.

On the other hand, this is very challenging. Implementing such a module in hardware requires a lot of knowledge in the field of FPGA programming and the overall outcome would surely be somewhat worse to maintain than the existing software code.

5.3.2 Hybrid Solutions

Another idea would be to bridge the both extremes of evaluating the rules in either hardware or software. We could for example limit the number of instances for the source address match to 10 instances. Then we could code the source address match in Verilog and instanciate it 10 times. The firmware part of the match would load the appropriate configuration values into these instances when being reconfigured. When the firmware part of the match is finally evaluated, the function would just read a single value from the hardware which tells whether the match applied or not. Since the actual check is done in hardware, it would be much faster, essentially bringing down the evaluation time for the source match from 1.86 µs to the minimum match time of 0.34 µs.

Clearly, this solution would give up some reconfigurability, but it would be a great trade-off between reconfigurability and performance. The challenges here could be the transfer method for the configuration of the matches, since the address space is very limited. Of course, this solution also requires a good understanding of hardware and hardware description languages.

5.4 Stateful matches

Another interesting topic are stateful matches. These take history into account, such as properties or rate of previous frames. A good example for such a match is iptables' limit match, which matches for example 10 times per minute. The rate is of course configurable. This is not meant as a kind of traffic shaping but should be seen as a security feature, e.g. to limit connection attempts. Another good example are all frames, which involve the concept of a connection. So WiFire could keep a list of active connections and associate individual frames to the connections, thereby e.g. prohibiting new connections to a specific node when combined with other matches.

Stateful matches come along with a very challenging task: The early evaluation exit paradigm which we used when evaluating rules affects the behaviour of the stateful matches, because these have side effects. Therefore, the order of matches in a rule is now very important.

5.5 User defined targets

Currently, there are only the standard targets implemented. These are the default verdicts for accepting or jamming a packet and the subchain handling. Another possibility would be to call a user-implemented function which is configurable by parameters, much like the matches.

Using this mechanism, special jamming targets could be considered. For example if there is an unencrypted key exchange, we would want to destroy the whole key while it is on air, not only some bytes of the frame. Note, however, that this idea could conflict with deferring matches as described in section 5.2.

Another feature would be logging. Logging can be used to analyse traffic in order to optimize rules or to be alerted by spurious traffic behaviour.

5.6 Optimization

Optimization is also a very important issue for firewalls. Since the evaluation code is already highly optimized, there is little to do to get any faster without getting rid of functionality. But, there can be a lot of static optimizations concerning the order of matches in a rule and of rules in a chain. Both have an impact on the overall response time of the evaluation. Some reorderings could be automatically, others depend on the particular traffic.

Automatic reordering of matches and rules could be done statically on the host. The challenge is to not alter the overall behaviour of the firewall with and without reordering.

Deferring evaluation, as explained in section 5.2, could be optimized by keeping an evaluation state. For example, consider a match which needs 30 bytes and there is a frame with 60 byte payload in the air. If the match should be evaluated after the header is received, it cannot get to a

result and would tell WiFire to wait until there are 30 bytes available. This circumstance could be stored in some structure, which would allow WiFire to jump exactly to that state when the 30^{st} byte arrives. The alternative would be to evaluate everything from the beginning, which would take a lot more time.

5.7 Porting

During my work, the people at Ettus Research also developed their code for the USRP further. As we setup our own things on their code, we have also to keep up with their state. Unfortunately they made a big change, to be more precise: they replaced the whole processor on the FPGA by another architecture. Since that change did not seem to be finished, I did not updat but used an older version of their code.

In the future however, one should try to keep up with their most recent versions as there are lots of bugfixes involved.

5.8 Allowing for several communication technologies

By now, only IEEE 802.15.4 is supported by WiFire. Another future direction is to implement other receivers and therefore other communication technologies. This will also involve the design of the rule system, since a single match which deals with address formats of different communication systems does not make any sense.

Hence it would be more reasonable to have something like namespaces for the communication technologies, so all matches and targets could be associated with one specific technology. On the configuration side, different toplevel chains could be used or even the concept of tables could be considered for that purpose.

References

[1] G. Dschung, "Efficient Wireless Traffic Classification using Software- defined Radio," Bachelor's thesis, TU Kaiserslatuern, Department of Computer Science, 2010.

[2] S. Dyckmans, "Real-time Classification of Wireless Communication Using Software Defined Radios," Master's thesis, TU Kaiserslautern, Department of Computer Science, 2010.

[3] Ettus Research LLC, "USRP2 data sheet," accessed on 11/01/2010. [Online]. Available: http://www.ettus.com/downloads/ettus_ds_usrp2_v5.pdf

[4] B. Fox, "The GNU Bourne-Again Shell." [Online]. Available: http://tiswww.case.edu/php/chet/bash/bashtop.html

[5] B. Fox and C. Ramsey, "The GNU Readline Library." [Online]. Available: http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html

[6] IEEE, *Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs).* New York, NY, USA: IEEE Computer Society, Oct. 2003. [Online]. Available: http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf

[7] M. Schäfer, "Wireless Firewall – Architecture and Implementation," Distributed Computer Systems Lab, University of Kaiserslautern, November 2010.

A Using wftables

The `wftables` program is an interactive console application. When started, it just gives a shell prompt, such as

```
main>
```

The shell is based on GNU Readline, a library which is also used in most command interpreter shells of unix-like systems, such as the Bash. Therefore a lot of known and beloved features are already possible for free, such has having a comfortable line editor with a command history. Other features known from shells such as the TAB-completion were programmed explicitly in wftables. Since there were no parser frameworks around which were applicable for the completion-feature, I built my own parser framework to get around that. There are still some issues to be solved, but at least parsing works without any flaws.

In the following subsections, I will introduce some commands and their syntax. Keep in mind, that you can always use the TAB key to get completition suggestions.

A.1 Adding Rules

To append a rule to a chain, use the `add` command. The command is followed by zero, one or more matches and a target specification.

A match specification begins with `-m` followed by the match name and the appropriate match parameters, each beginning with two minus signs. An example would be the source address match:

```
-m src --mode 2 --pan 0x22 --addr 5
```

This instanciation would match each frame having Source Address Mode 2 (16 bit addresses), Source PAN-ID 0x22 and Source Address 5.

A target specification begins with `-j` followed by the target name and the appropriate target parameters. An example is the JAM target:

```
-j JAM
```

A.2 Matches

At that time, there a only a few matches implemented. Nevertheless here is some information for documentation purpose.

Each match can be inverted individually by placing an exclamation mark in front of the match name

```
-m ! frametype --ack
```

A.2.1 Address Match

The address match comes in two flavours: source address and destination address match. The names of the matches are `src` and `dst` respectively. The match takes exactly three arguments (in any order).

mode This numeric parameter specifies the mode of the address. According to the standard, there are three possibilities:

- 0: no address (used for “anonymous” source and broadcast destination)
- 2: 16 bit addresses
- 3: 64 bit addresses

pan This numeric parameter specifies the PAN-ID to be matched. If the address mode is 0, there is also no pan. However, in that case you will have to provide a dummy value since the parser always expects a pan.

addr This numeric parameter speicifies the address to be matched. It is either a 16 or 64 bit value, depending on the mode.

A.2.2 Frame Type Match

This match takes exactly one parameter which corresponds to the frame type to be matched. The parameter is preceeded with two minus signs and the name, either `ack`, `data`, `beacon` or `ctrl` to match an ACK frame, data frame, beacon frame or control frame, accordingly.

A.3 Targets

The target always specifies what to do if all matches of a rule apply. There are more obvious targets which actually come to a verdict for a current frame and there are some special targets to handle subchains. As a third type, there could be special targets which execute user-implemented functions. Examples for this type could be a configurable jam duration or a logging target. Since these targets are currently only supported on the USRP side, i.e. there is no infrastructure on the host side wftables, I will not discuss this type of targets.

ACCEPT This target accepts the current frame, which means that the evaluation ends at this very point and there is no further action taken.

JAM This target also stops evaluation but will eventually trigger a the jam routine (at a later point in time) which finally destroys the frame.

JMP This target additionally takes the name of a subchain to jump into. The subchain can either come to a final verdict (such as `ACCEPT` or `JAM`) or return to this point and continue evaluation. Clearly you must not create any loops when jumping around—Well, the software wouldn't let you do that anyway.

RET When being in a subchain you can return at any point to the parent chain using this target. This is also the default behaviour if no rule of a subchain applies. Clearly, you cannot return from a toplevel chain.

A.4 Putting it together

Let's say we want to accept all data frames comming from node 5 and all non-control frames going to node 10 and all ack frames.

```
add -m src --mode 1 --pan 0x22 --addr 5 -m frametype --data -j ACCEPT
add -m dst --mode 1 --pan 0x22 --addr 10 -m ! frametype --ctrl -j ACCEPT
add -m frametype --ack -j ACCEPT
add -j JAM
```

A.5 Listing rules

You can list all rules of a chain using the command

```
list chain
```

You could also provide the name of a chain after that command.

Each rule is prefixed by a number which can be used as an identifier when deleting or inserting rules.

A.6 Deleting rules

To delete a specific rule, you can use

```
del rule NUM
```

where `NUM` is replaced by the rule number. Use the `list` command to get the rule number.

You can also delete all rules of a chain, using the command

```
flush
```

A.7 Inserting rules

To insert a new rule, you can use

```
insert NUM MATCHES TARGET
```

where `NUM` is replaced by the position of the new rule. The matches and target specifications are the same as for the `add` command.

A.8 Chains

To create a new subchain, use

```
chain NAME
```

where `NAME` is replaced by the name of the new chain. You can also use this command to change between existing chains. The currently selected chain is always printed as the shell prompt, e.g.

```
main> chain newChain
newChain>
```

Of course, chains can be listed (`list chains`) and deleted (`del chain NAME`). Note, that deleting a chain is prohibited if it is still in use, i.e. if there is a target pointing to it.

A.9 Default Policies

The policy of a chain can be described as the default target to be taken if no rule of the chain comes to a verdict.

The default policy of subchains is to return to the parent chain and this behaviour is not changable (well, you could add a rule without any matches at the end of the subchain to achive the behaviour of a default verdict).

For a toplevel chain, the policy is one of the verdicts, `ACCEPT` or `JAM`. The default is `ACCEPT`. You can change it to `JAM` using the command

```
policy JAM
```

when being in a toplevel chain.

A.10 Committing

In order to push the current configuration to the USRP, you have to connect and commit the rules

```
connect 192.168.10.2
commit
```

Note, that you have to connect only once when you do some changes. However, don't forget to commit.

B Adding new matches

In order to add new matches, you will have to add code on both sides: the USRP and the host side. On the USRP side, the behaviour of the match is implemented. On the host side, there has to be code for parsing the match specification, for keeping a specification and for packing the data into a data structure which can be interpreted by the code on the USRP.

In the following I will discuss an example implementation which checks the frame length. It should be possible to formulate matches like "Less than or equal to X bytes" and "More than or equal to X bytes".

B.1 USRP Side

On the USRP side, there is a subdirectory `matches` for this purpose. You can just add a new C file to this directory. In that C file, you have to specify at least two things:

The match function is the function which actually tells whether a given frame fulfils a certain property. The signature of the function is

```
static WF_bool lengthMatch(const struct WF_Packet * packet,
  const struct WF_Match * match);
```

Since we need some configuration, we also have to add a C data structure which will provide the data:

```
struct LengthConfig {
  uint8_t mode;
  uint16_t len;
};
```

In the match function, the configuration data is available in the `data` member of the parameter `match`. It is untyped, so we have to reinterprete it to our structure type.

```
  struct LengthConfig * cfg = (struct LengthConfig*)match->data;
  switch (cfg->mode) {
  case 0: // less-or-equal
    return packet->len <= cfg->len;
  case 1: // greater-or-equal
    return packet->len >= cfg->len;
  default: // should never happen but there is little we can do about it
    return false;
```

The match description is an instance of the `WF_MatchDesc` structure and exports the whole match, such that is known by the rule system.

```
struct WF_MatchDesc WF_lengthMatch = {
  .name = "Frame␣Length",
  .fun = &lengthMatch
};
```

Note, that you have only 19 bytes for the name of the match.

Next, the match description must be made available to the rule system. As a first step, add the description to the header file `wfmatches.h`

```
extern struct WF_MatchDesc WF_lengthMatch;
```

At this point you can recompile the wftable subsystem, which will update `libwftable_uhd.a` if everything was ok.

As a last step, you have to register the description in the rule system. For that purpose, edit `wifire.c` of the firmware and add the appropriate line in the function `void wifire_init()`:

```
  WF_registerMatch(&WF_lengthMatch);
```

B.2 Host Side

As already mentioned, there are some more things to do on the host side. We have to extend three subsystems for our match. This will be explained in more detail in the next subsections.

B.2.1 Parser

As a first step you have to consider an input format for the parameters of your match. For example we could consider the following three alternatives to express "frame length is less or equal 65 bytes".

```
 -m length --mode le --length 65
 -m length --le 65
 -m length <= 65
```

The first one is not very intuitive. The last alternative is maybe the most intuitive, but we sould consider the second one, since it follows the convention that all parameters begin with two minus signs.

The Extended-Backus-Naur-Form of the syntax for the parameters would thus be

```
LengthParam ::= ('--le' | '--ge') Number
```

where `Number` is already a known symbol.

This specification can be translated into the parser framework of wftables.

```
LengthParser::LengthParser() : MatchParser(LengthDesc::Instance())
{
  Builder le = Builder("--le");
  Builder ge = Builder("--ge");

  optLe = le;
  optGe = ge;

  sequenceAdd(le | ge);
  sequenceAdd(lenParser = Parser::NumberTerminal<uint16_t>::New());
}
```

It should be mentioned, that each `MatchParser` is a sequential parser of the framework, so we can add parameters using the `sequenceAdd` method.

The `Builder` class is just for shortcuts. As seen in the example, you can just use the expression (`le` | `ge`) to create a parser which matches either the `le`-Parser or `ge`-Parser. There are some more things you can do with the Builder, such as matching parts optionally or arbitrarily often. You can see examples in other matches and all over the code.

By now, we only specified the syntax of the parameters. But if the match is instanciated in a rule, we have to actually get the syntax tree and the semantic values which were specified. This is done in the `compile` method of the class.

```
  Entity::ptr c = Parser::as<Parser::Choice>(at(1))->get();
  uint16_t len = lenParser->get();
```

```
  if (c == optLe)
    return Length::NewLe(negate, len);
  else if (c == optGe)
    return Length::NewGe(negate, len);
  else
   return Match::ptr();
```

Parsing alternatives is a bit annoying. We use the `at` method to get to the first entity of the sequence, which is (according to our syntax definition above) the choice. Since the syntax is specified dynamically at runtime, we do not know the type here, thus we have to explicitly cast it to a choice using `Parser::as`. The rest should be obvious.

B.2.2 Translation

After the parameters are parsed, an instance of the class `Length` keeps the information. It is responsible for re-printing the parameters if e.g. rules are listed. But the most important work of the class is to translate the parameters in that format, the length match on the USRP side will understand.

There are two methods which are tightly coupled to the definition of the configuration data structure on the USRP side. The first one calculates the size of the structure from its layout. The second one encodes data in the same way the structure will do. This may seem easy, but under the hood, there is a lot of alignment and endianness stuff. Fortunately this complexity was hidden by some helper classes.

```
void Length::_size(Platform::Size& config) const
{
  // this is the structure layout
  config = config + SizeOf<uint8_t>() + SizeOf<uint16_t>();
}

void Length::_write(Stream& config) const
{
  config.write<uint8_t>(mode);
  config.write<uint16_t>(length);
}
```

Note that this is one of the most crucial parts! If data structures are not encoded the same on USRP and host side, the data will be useless in any way.

B.2.3 Description

As on the USRP side, the host side has also some description of the matches. This is done by inheriting from the class `MatchDesc`.

```
LengthDesc::LengthDesc() : MatchDesc("length", "Frame␣Length") {}
```

The first parameter of the `MatchDesc` Constructor is the name of the match, as specified on the command line. The second parameter is the identifier of the match. This is another crucial part: this identifier must exactly match the one on the USRP side, since matching the USRP part and the host part of a match is done via this identifier.

C Building

The following chapter will give some notes on how to build and setup WiFire.

Note: The paths which will be used here may differ from your actual setting, so be sure to adapt them. In the examples, the uhd repository is cloned into `~/uhd`, and the wifire repository is in `~/wifire`.

C.1 Preparation

First, checkout the right git revision

```
git clone --bare git://code.ettus.com/ettus/uhd.git
git checkout b4d58f3501596fdddf240d576d0b1b2cb5862892
```

Next, apply the patches for the firmware and the FPGA Code

```
cd uhd
patch -p1 < ~/wifire/patches/firmware.patch
patch -p1 < ~/wifire/patches/fpga.patch
```

In order to build the FPGA Bitfile and the firmware you will need Xilinx ISE 12.2 and the Microblaze GCC[1].

C.2 UHD

The (patched) UHD Firmware, Host Library and FPGA Bitfile is generated. The Firmware is the basis for the WiFire firmware, the Host Library is needed in order to tune the USRP to the right frequency and the FPGA Bitfile is the synthetisized hardware needed for WiFire.

C.2.1 FPGA

You will need to have the Xilinx ISE tools in your path.

```
source /opt/Xilinx/12.2/ISE_DS/settings64.sh
```

Note: You should run the above command only in the shell where the next commands are executed. You should not import the ISE Settings for anything else.

Then create the project file

```
cd uhd/fpga/usrp2/top/u2_rev3
make proj
```

This will create `build/u2_rev3.xise`. Open that file in the ISE Project Navigator.

- Add the receiver (using `Project → Add Source`): change into the receiver directory and add `receiver_overall_reset_cw.sgp`. The receiver can be found in `wifire/receiver` (you'll have to extract the zip first).
- Add the MSDU RAM: change into `wifire/coregen` and add `xlnx_bram_128_8_to_32_32.xco`
- Add the WiFire components: change into `wifire/verilog` and add `wifire.v` and `wifire_frame_decoder.v`

After that you can select the top-level module (`u2_rev3`) and double-click on `Generate Programming File` in the right window.

This will eventually create `build/u2_rev3.bin` which is the FPGA Bitfile. This file will be copied to the SD-Card later.

[1] http://gnuradio.org/tools/mb-gcc-4.1.1.gr2.i386.tar.gz

C.2.2 UHD Host Library

```
cd uhd/host
cmake .
make -j8
```

Note: you can also use `cmake -G‘‘Eclipse CDT4 - Unix Makefiles’’` in order to create an Eclipse CDT Project.

C.2.3 UHD Firmware

```
cd uhd/firmware/microblaze
./bootstrap
./configure --host=mb
make
```

C.3 WiFire

C.3.1 WFTables for USRP

```
cd wifire/wftable/target
make
```

This is a dependency for the WiFire Firmware, so it must be built before the firmware.

C.3.2 Firmware

```
cd wifire/firmware
make GR_UHD=~/uhd
```

At this point, you can install the firmware (and maybe FPGA Bitfile) on SD-Card:

```
make GR_UHD=~/uhd SD_DEV=/dev/sdb fpga
make GR_UHD=~/uhd SD_DEV=/dev/sdb all
```

The first command will copy the FPGA bitfile to the SD-Card, the second one will copy the firmware. You should make sure to pass a valid path to your SD-Card Device and you need reasonable rights to write directly to that device. Doublecheck that because you could overwrite your data on your hard disk.

C.3.3 Setup Tool

```
cd wifire/tools
make GR_UHD=~/uhd
```

C.3.4 WFTables for Host

```
cd wifire/wftable/host
make -j8
```

C.4 Setup

After burning the SD-Card, you can power up the USRP. There are several steps to finally get WiFire running:

1. Setup Tool:

```
cd wifire/tools
./setup_tool
```

   This will tune the USRP to the right frequency and set some other parameters like antenna gain and so on. You can use the `help` switch to get an idea of it. If your USRP is reprogrammed to another ip (e.g. 192.168.10.55), use

```
./setup_tool --args addr=192.168.10.55
```

2. WiFire Administration Tool:

```
cd wifire/tools
./wifireadm --opmode wifire --jam_type wifire
```

   This will send commands to the firmware over UDP in order to bring the firmware into WiFire mode, setup interrupts and so on. You can use the `help` switch to get an overview of some other options.

3. WFTables:

```
cd wifire/wftable/host
./wftables
```

   See section 5.8 to get used to the wftables application.

C.5 Simulation

The WiFire components have their testbenches which apply some important input stimuli to the components and check their outputs for correctness. In order to use that, you will need to have Icarus Verilog[2] in your path.

```
cd wifire/verilog
make CODEBASE=~/uhd/fpga/usrp2 ISE=/opt/Xilinx/12.2
```

This will create and run the test benches, which in turn creates `vcd` files which can be opened in gtkwave[3], a Wave-Viewer Application. Using gtkwave you can trace the level of signals over time. The `sav` files in the directory can be opened together with the `vcd` files in gtkwave, then the important signals are loaded.

[2] http://www.icarus.com/eda/verilog/
[3] http://gtkwave.sourceforge.net/