Is the Network Turing-Complete? EPFL Technical Report 187131

Peter Perešíni
EPFL

Dejan Kostić
Institute IMDEA Networks

***Abstract*—Ensuring correct network behavior is hard. This is the case even for simple networks, and adding middleboxes only complicates this task. In this paper, we demonstrate a fundamental property of networks. Namely, we show a way of using a network to emulate the Rule 110 cellular automaton. We do so using just a set of network devices with simple features such as packet matching, header rewriting and round-robin loadbalancing. Therefore, we show that a network can emulate any Turing machine. This ultimately means that analyzing dynamic network behavior can be as hard as analyzing an arbitrary program. Analyzing a network containing middleboxes is already understood to be hard. Our result shows that using even only statically configured switches can make the problem intractable.**

I. Introduction

Correct network behavior is often taken for granted, but ensuring it is a daunting task. It was previously shown that statically analyzing the network and answering questions such as "can host A talk to host B" can be NP-hard [MKA+11].

The networks are increasing in complexity, as well as in the set of roles that are supposed to fulfill. Today's computer networks consist of equipment such as switches/routers[1], and various kinds of middleboxes (load balancers, intrusion detection systems, network address translators, etc). Each of these devices exposes traffic to a variety of features (packet matching, packet rewriting, *etc.*). These features may, however, interact in very complex ways. This means that analyzing dynamic network behavior (which cannot be checked statically) is also becoming more and more difficult.

For example, in security, protocol interactions can pose significant security risks amongst otherwise secure protocols [KSW98]. Currently, there is a lack of similar research on the interaction of basic features of today's networks. This gap is becoming more important as the software-defined networking, mainly the OpenFlow protocol, is growing up in its popularity.

In this paper, we show that the complexity of the interactions is inherent even in simple scenarios. We investigate the interaction between three simple network mechanisms – a packet *header matching*, a packet *header rewriting* and unicast/multicast *forwarding and/or round robin load-balancing*. On its own, each of these mechanisms is fairly simple but their combination might easily become hard to analyze.

[1]In this paper, we will use the terms *switch* and *router* interchangeably, both meaning a device capable of matching packets according to some criteria and forwarding them to one or more destinations.

A. A starter example – emulating a binary counter

To illustrate how simple switch features might interact, we start by constructing a binary counter, *i.e.* a device which can go through states 0, 1, 10, 11, 100, 101, ...We use a single switch with a loopback link (*e.g.* 2 switch ports connected to the same link) supporting following features: (i) matching packets on (a combination of) header fields (*i.e.*, either exact match or wildcard for each field); (ii) support of different match priorities (for overlapping match rules); and (iii) rewriting packet headers and/or forwarding to a specific port based on the matched rule.

We represent the value of the counter as a binary number where each bit is stored in its own packet header field.[2] When packet enters on the ingress port, we clear the packet by rewriting all applicable header fields to zeroes. Increment operation is performed by rewriting header field suffix "0,1,1,1,...,1,1,1" into "1,0,0,0,...,0,0,0" and looping the packet. There are two special cases: (i) all header fields are 0 – we rewrite the last field to 1 and loop; (ii) all header fields are 1 – we forward the packet to the output and finish the counting. The rules are summarized in Figure 1.

To summarize our example, if the packet header contains n independent fields, we can loop the packet 2^n times in the network using a switch with just $n + 2$ rules. Moreover, the previous example can simply be extended to k-ary counters. In that case, one can count up to k^n by using $\Theta(n * k)$ rules.

rule	match		rewrite	fwd
	port	$H_n \dots H_1$	$H_n \dots H_1$	
init	in	*******	0000000	loop
finish	loop	1111111	-------	out
digit n	loop	0111111	1000000	loop
digit $n-1$	loop	*011111	-100000	loop
digit $n-2$	loop	**01111	--10000	loop
digit $n-3$	loop	***0111	---1000	loop
digits $n-4$ to 3	...		...	
digit 2	loop	*****01	-----10	loop
digit 1	loop	******0	------1	loop

Fig. 1: Binary n-digit counter. Rules are in the order of deacreasing priority.

[2]It might not be possible to use all available header fields – for example by clearing protocol field one invalidates all IPv4 fields

B. Contributions

In this paper, we demonstrate a way of emulating the Rule 110 cellular automaton [Wol86] just using a set of network devices with simple features such as packet matching, header rewriting and round-robin loadbalancing. As the Rule 110 automaton is Turing-complete, we are therefore able to emulate any Turing machine. Thus, as a corollary, we show that analyzing computer network configurations can be as hard as analyzing arbitrary computer programs.

Therefore, the main contribution of this paper does not entail a new approach for ensuring network correctness. Instead, it is a fundamental result.

II. Modelling the network & basic building blocks

In this section informally introduce a network model used through this paper. For the formal model, please refer to the appendix. Additionally, we present basic building blocks we use in the rest of the paper, each building block corresponding to a statically configured switch.

Our goal is to model an *asynchronous* computer network with unbounded propagation time. The model of the network is represented as a directed graph with nodes representing switches and edges representing directed links. For the modelling purpose, we abstract out the packet delivery over links. Instead, we assume that each packet in the network is located either at an ingress or at an egress queue of some switch. We assume only simple FIFO queuing disciplines and atomic packet processing, that is, the network state can change by (i) atomically moving the first packet from a switch egress queue to the connected switch's ingress queue (*i.e.* sending packet over link); (ii) atomically moving the first packet from a switch ingress queue, enqueuing it (with possible modifications) to all relevant egress queues and updating switch state (*i.e.* switch packet processing).

From the switches, we require support for following packet processing capabilities:

Forward/multicast: The switch can be configured such that any packet on one of its ports will be forwarded to one or several other ports.

Header matching: The switch is able to match packet on a combination of header fields and forward packet to the port associated with matching rule. We require the support for overlapping rules with different priorities.

Header rewriting: The switch is able to rewrite a subset of header fields and then forward the packet.

Round-robin loadbalancing: The switch provides a load-balancing mechanism where the packet on the ingress port is forwarded to one of the m egress ports. The output port is chosen in a round-robin fashion, where the first packet is forwarded to the first egress port, the second packet forwarded to the second egress port, ... the $m+1$-th packet forwarded again on the first egress port.

Although real switches might support any combination of these features, it is enough for us to have just a single capability per switch. We use the switches with such capabilities to model basic building blocks for our Turing machine concstruction. We will call these building blocks *gates* (summarized in Fig. 2) and each basic gate is easily implemented by a single switch. We leave possible optimization of merging several of these elements into a single switch as a future work.

`split:` Copies a packet on one ingress link (also denoted as input *wire*) into multiple output wires by using the multicast capability.

`merge:` Accepts a packet on any input wire and forwards it to the output.

`rrobin:` Packets on any input port should be forwarded to one of several output wires in round-robin fashion. The element is implemented by a round-robin loadbalancer feature.

`rewrite:` Rewrite packet header field(s) of any incoming packet with the new values and forward it to the output.

`conditional:` Emulate a simple decision making with the packets by outputting the packet to one of the outputs depending on the match. The element is realized as a switch with two rules: (i) a high-priority yes rule matching the condition on the header field; (ii) a low-priority no rule that matches everything else (default rule).

`join:` The `join`[3] is an element which filters a sequence of packets and passes through only each m-th packet on the input. It is implemented as `rrobin` with first $m-1$ egress ports "dead" (not connected to any link).

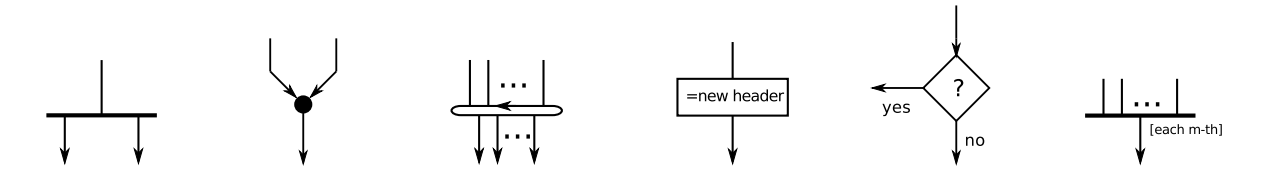
III. Boolean functions

As a first step towards Turing machine emulator, we show a way to emulate a boolean circuit [AB07] – an acyclic directed graph with n nodes with not incoming edges called *inputs*, m nodes with no outgoing edges called *outputs* and several other nodes called *logical gates*. We will call all edges of a boolean circuit *wires* because of their representation in hardware circuits. Each logical gate contains several incoming edges, a single outgoing edge and performs a boolean function over its inputs. In this paper, we will consider only logical gates `AND`, `OR` and `NOT` as they are functionally complete and one can construct any other logical gate out of them – in fact, `AND` and `NOT` would be sufficient [End].

A. Representing boolean circuits as computer networks

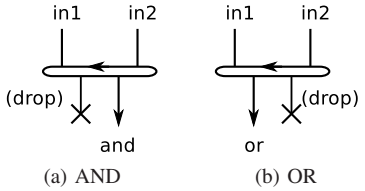
We represent each logical gate as a set of interconnected switches (*e.g.*, basic elements). Edges of the boolean circuit graph are represented by network links. Although boolean circuits use bits as input values, we cannot represent bits by a packet header field – both `AND` and `OR` need in certain situations to wait for both inputs before they can produce the output. However, such "packet save" functionality is not present in the switches. Instead, we take an approach similar to the hardware implementation of boolean circuits and we represent boolean values as the presence/absence of *a single* packet on the link.

[3] similar to *join* in UML diagrams



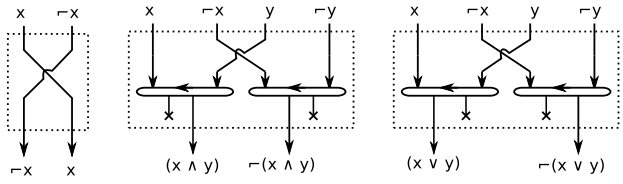
(a) split (copy to multiple destinations) (b) merge multiple sources (c) round robin (loops through m outputs) (d) header rewrite (sets new packet header) (e) conditional (f) join (pass through every m-th packet)

Fig. 2: Pictograms of basic building blocks, each one can be implemented by a single switch.



(a) AND (b) OR

Fig. 3: Basic boolean operators (single-wire inputs)



(a) NOT (b) AND (c) OR

Fig. 4: Boolean operators using the two-wire representation.

B. Simple `and` and `or` gates

For both `and` and `or`, we exploit the fact that `rrobin` element cycles through its outputs. The key idea is simple – `and` needs to produce a single packet on the output iff (if and only if) it receives packets on both of its inputs. That is, we need to drop the first packet (if any) and let through the second packet (if any). This is a perfect match for `rrobin` gate with first output link dead[4] and the second output representing the `and` output as illustrated in Fig. 3. Note that while `and` gate has the same hardware implementation as `join` ($m = 2$) gate, we will use two gates to make the semantic distinction. The `or` can be realized in a similar manner as `and`, the only difference is that we exchange the live and dead links.

C. Negations

The approach, as is, cannot handle negations. The `not` gate would need to produce a packet iff there is no (and nor will not be in the future!) packet on the input. This is simply impossible to achieve in asynchronous networks with unbounded propagation time. To address this issue, we develop a *two-wire* bit representation. In the representation, each input of a boolean circuit consists of two network links, each link corresponding to one of the two possible bit values. The boolean circuit input is represented as *a single packet on exactly one* of the wires. If the value of input x is one, the input consists of a single packet on wire denoted "x". Otherwise (the value is zero) the input consists of a single packet on wire denoted "$\neg x$".

The `not` gate in this new representation is simply an exchange of the wires (Fig. 4a). For the `and` and `or` gates, we reuse the previously introduced single-wire `and`/`or` gates and combine them using De Morgan identities $\neg(x \wedge y) = (\neg x) \vee (\neg y)$, $\neg(x \vee y) = (\neg x) \wedge (\neg y)$ to produce the two-wire gates (Fig. 4b, 4c) [CC90].

[4]We assume that round-robin loadbalancing algorithm in the switch does not skip dead over dead links. Otherwise, we may utilize an additional switch instructed to drop any incoming packet

IV. Repeating computations – reusing gates multiple times

If we want to emulate a (possibly exponentially long) computation of a Turing machine in polynomial size of the network, we need to reuse the same gates several times with different inputs. This, however, poses a new challenge – we need to clear/reset the internal state of these gates. In particular, the elements needed to be reset are all `rrobin` elements as they end up in different states depending on the number of input packets. Additionally, we need a mechanism for slowing down some packets – sending second set of inputs to the gates too fast can end up with some of the new packets interfering with previous, still-running, computation. In this section, we address these two challenges by designing *clearable* gates and a new special element which can "*buffer*" packets.

A. The art of clearing

In order to reuse gates, we need to reset any changes to the internal state of our gates. This is trivial for a `not` gate, but poses a significant challenge to `and` and `or` gates, which contain state of the round-robin algorithm inside. To address this, we introduce a special *clearing* packet(s) which can be distinguished from information packets by matching on some header field.

To illustrate the mechanism of clearing, consider the `and` gate from Fig. 4b – we extend this gate to a self-clearing `and` gate in Fig. 5. In the new gate, the clearing packet needs to be created after the original `rrobin` gates processed all packets on their inputs – this moment can by identified as a moment when the total number of packets processed by both `rrobin` gates is two (we know that there will be exactly two input packets on four input wires). Therefore, we collect copies of all packets output by `rrobin` elements and wait for the last one using the `join` element. Note that the newly introduced `join` internally contains a new `rrobin` which potentially

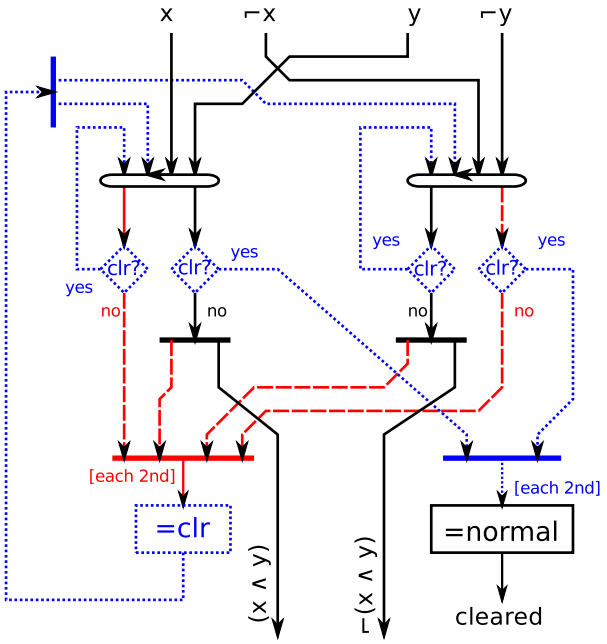


Fig. 5: Clearable AND gate. Original AND is in black, red part (dashed) waits until the circuit finishes computing and blue (dotted) part is responsible for clearing.

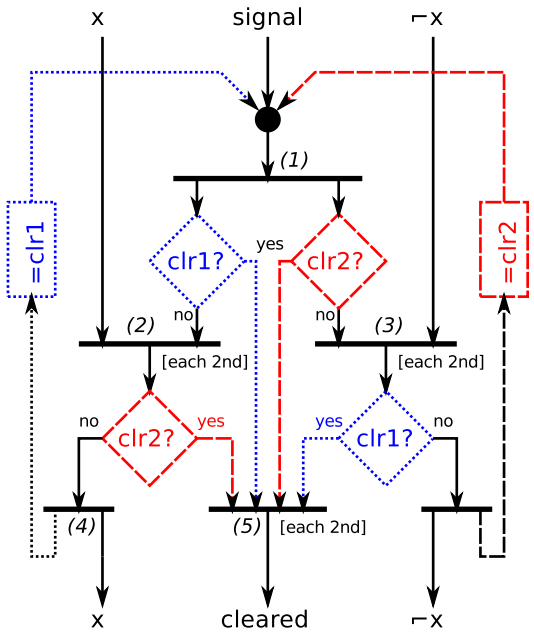


Fig. 6: Buffer (holds value until a signal is received).

must be reset. This `rrobin` is, however, reset to its initial state after receiving the second packet and we do not need to clear it explicitly.

The output of the `join` is then converted to a clearing packet by header rewriting. Further, we split this clearing packet and use it to clear both `rrobin` elements separately. A single `rrobin` element can be easily cleared by a looping trick – all it takes is to loop the clearing packet enough times that after the last iteration, the element will be in the initial state. We may do this by intercepting `rrobin` outputs and sending the packet back to loop on all but the last `rrobin` output link.

Finally, after both clearing packets for both `rrobin`s finish looping, we wait for the last one by using `join`. The results is a single packet indicating that the circuit was cleared. Again note that newly introduced `join` does not need an explicit reset.

Lemma: Clearing does not interfere with the computation. Clearing starts after both `rrobin` elements finish processing their inputs. The rest of the computation (*i.e.* forwarding packets to the output) is unaffected by parallel cleaning as there are no more elements with the internal state.

Lemma: Clearing eventually finishes and when the circuit outputs a signal, it is ready to be reused. After receiving all (two) input packets, they will be eventually forwarded to `join` and thus eventually reach the clearing `rewrite`. Subsequently, the newly split packets will both loop finite number of times before being forwarded to the final `join`. By looping, these packets will reset both `rrobin` elements and `join` elements self-reset right after they place a packet to their output queue (and before the packet is forwarded further).

B. *Waiting – because clearing is not enough!*

To correctly repeat calculations, clearing the state is not enough because the circuit is quite brittle to race-conditions – if we send new inputs while the circuit is clearing, the clearing can interfere with the ongoing computation. Therefore, we need an element which can hold the inputs till we know it is safe to send them further. To address this, we create a new `buffer` element. The challenge is to perform the buffering task without any direct control over switch buffers – the only way to prevent a packet from being processed by the switch and sent on the output link is to drop it.

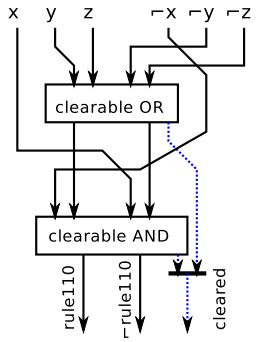
The key idea behind the `buffer` element (Fir. 6) is that we actually *drop* packets and then "recreate" them when needed. Of course, we are not able to fully recover the original packet with all headers and data. However, this is not needed – we can simply copy any other existing packet as in our case only the presence or absence of a packet matters[5]. The building block of the `buffer` element are two `join` elements, one for each input wire $(x, \neg x)$. Both of these `join`s wait for two incoming packets – the input and the *signal* telling that the buffer can be relased. After both the input and the signal packet arrive (in any order), one of the `join`s will pass through to the output. Moreover, it is necessary to reset the other join. Here we again utilize the special clearing packet as displayed in blue/red color in Figure 6.

Lemma: Buffer does not pass any packet until it receives both the input and the signal. Trivially, a single packet (either the input or the signal) is not able to pass any of the `join` elements.

[5]Actually, it is also important that the packet is created as information and not clearing packet. We copy only information packets.

x_{i-1}	x_i	x_{i+1}	new x_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(a) Rule 110 update table



(b) `rule110` gate

Fig. 7: Rule 110 - a simple linear automaton

Lemma: Buffer is cleared properly and signals its clearing only when it is safe to reuse it. We will use the fact that *packets following the same path cannot be reordered* (they cannot be reordered on links and we assume switches with FIFO input/output port queues).

Without a loss of generality, assume an input packet on wire x. After receiving packets on both x and *signal*, the `join` *(2)* will pass through the later packet while `join` *(3)* will drop the signal. Subsequently, the passed packet continues to `split` *(4)* where it is forwarded to the output and looped back as a clearing packet. The clearing packet is then split again at *(1)* with one part (left branch) being sent to `join` *(5)* and the second part (right branch) going resetting `join` *(3)* and further looping before finally reaching `join` *(5)*.

Notice that for `buffer` functioning properly, the clearing packet (right branch) must reach input `join` *(3) after* the signal packet. If we simply looped directly from *(4)* to *(3)* and bypassed *(1)*, the signal could take too long to travel from *1* to *(3)* and the clearing packet would be there first. Instead, our trick is to use `merge` and then `split` to guarantee that at the time we created the clearing packet, the original signal packet must have reached `split` *(1)* and the clearing packet will be placed behind it on all FIFO queues towards `join` *(3)*.

V. From circuits to a Turing machine

Final step is to show that clearable logical gates and `buffer` are sufficient to emulate a Turing machine. To simplify our design and fit into a short paper, we will construct a Rule 110 automaton instead of a full Turing machine.

Rule 110 Rule 110 [Wol86] is a simple linear cellular automaton (an array of cells) with each cell holding a binary value. Rule 110 computation consist of discrete steps, each step synchronously updating values of all cells. In each step, each cell looks at its own value and the values of its neighbors and update its value according to the function from Fig. 7a. Although Rule 110 is a simple automaton, it has been shown that it is capable of emulating the first n steps of a Turing machine in a polynomial number of cells [NW06]. Thus, by emulating Rule110 in polynomial number of network elements, we will be able to emulate first n steps of Turing machine in polynomial network size. This will prove that not only networks are capable of emulating Turing machines but that the emulation is also semi-efficient.

Rule 110 emulator We observe that the Rule 110 function in Fig. 7a can be translated into a boolean circuits as as $f(x, y, z) = \neg x \wedge (y \vee z)$. Thus, by using clearable logical gates, we can create a clearable Rule 110 gate (Fig. 7b. We create Rule 110 automaton as an array of interconnected circuits for cells. Each cell circuit is composed of a buffer holding the input values, the Rule 110 gate and the output value buffer. The Rule 110 gate is the connected to input buffers of the current and the neighboring cells. Finally, to enable the multi-step computation, we send the values of output buffers back to the input buffers and thus create a loop. The overall structure of the construction is shown in Fig. 8.

Lemma: The emulator construction in Fig. 8 emulates the Rule 110 cellular automaton in synchronous steps.

It is easy to see that the circuit is iteratively computing Rule 110 cell values from the previous values. What is more challenging is the proof that cells are updated synchronously and that we do not feed values to the circuit before it acknowledged clearing. We use the natural split of the circuit into three stages: (i) input buffers; (ii) Rule 110; and (iii) output buffers. When the computation begins, all packets are in stage (i). After receiving the "start" packet, the computation can proceed to stages (ii) and (iii). However, the output buffers in stage (iii) will hold all packets until all input buffers and all `rule110` circuits acknowledge clearing. At this moment there are no packets in stage (i) and (ii)– the input buffers and `rule110`s are cleared. Thus, the output buffers can forward the packets back to the input (stage (i)). Here, the packets will be held until all output buffers acknowledge clearing and there is not packet in stage (iii). Only after this, the new round of computation may begin. Thus, we proved that the circuit is updated synchronously and that there is no unwanted interference between the packets in different stages.

VI. Future work

There is certainly room for improving our results. In particular, we leave the direct construction of a Turing machine emulator for our future work. Such emulator, if constructed, should be capable of emulating the Turing machine with bounded tape length in limited network size. Thus, the emulator would be capable of computing exponential number of steps in polynomial network size, achieving efficient emulation.

Another area of potential future work is optimization of the created network – in the current construction one switch for each single element. This unnecessarily wastes resources because switches typically have many ports and complicated packet processing pipelines. It is therefore interesting to explore this area to provide much smaller building blocks (*e.g.*, by packing whole Rule 110 cell into a single switch with several loopback links).

Finally, the constructs we show in this paper are quite brittle – they heavily depend on no other packets being forwarded by the switch. It is an interesting future direction to see if we can add some robustness to the design.

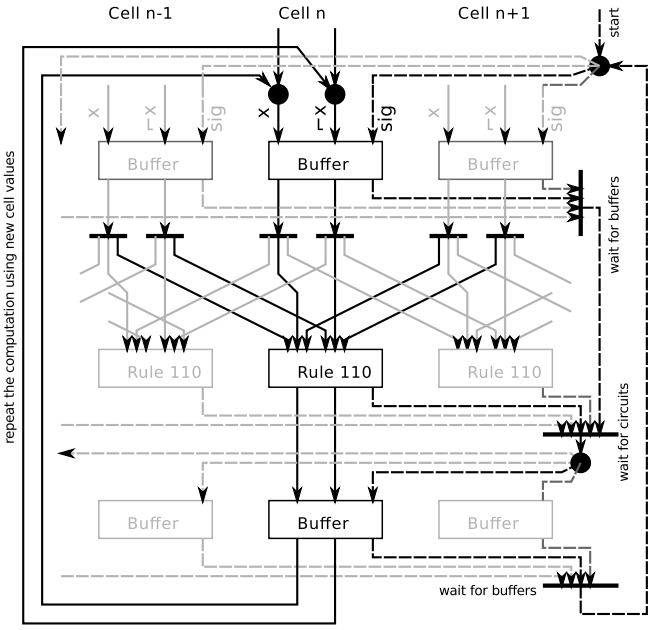


Fig. 8: Rule 110 emulator (showing only cells $n-1$, n, and $n+1$). Black part is responsible for one step of computation of cell n, grey parts represents different cells. Dashed part is responsible for synchronizing and advancing the computation.

VII. Related work

We are certainly not the first to suggest that systems with simple features might interact in a complex way. Protocol interaction from a security viewpoint was analysed in [KSW98]. Feature interaction was also studied in telecommunication services [BDC+89], [CKMRM03]. Unlike these studies, we are not trying to come up with a solution to the feature interaction problem. Indeed, we take an opposite way – we are trying to exploit the interaction of simple features and show that in general it might be very hard to analyze.

Parasitic computing [BFJB01] demonstrates the potential of solving SAT by network checksumming. Implicit simulations using messaging protocols [Koh03] demonstrate the use of the ICMP protocol as another way to employ simple computations using the network. Both [BFJB01] and [Koh03], however, need a host which creates the packet and the network works just as a simple filter of results. In contrast, we do not require intelligent endhosts, and still perform the full Turing machine emulation in the network.

The problem of analysing the network behavior was tackled by [XZM+05], [MKA+11]. These tools, however, perform only a static analysis of the network. They are not able to analyze the dynamic behavior of round-robin loadbalancers. Instead, for the analysis, they overapproximate the loadbalancing primitive simply as a multicast. Our result implies that this overapproximation trick enables them to predict the network behavior much faster than a dynamic analyzer could do, albeit at the cost of precision.

VIII. Conclusions

It is common wisdom that debugging networks is hard – analysing a network containing middleboxes is commonly understood to be hard. In fact, even analyzing network consisting of switches is hard – it was previously shown that statically analyzing the network and answering questions such as "can host A talk to host B" can be NP-hard [MKA+11]. In this paper, we further move the boundary and show that the ability to analyze dynamic behavior of statically configured networks consisting only of simple switches is equivalent to the ability of analyzing an arbitrary Turing machine. This implies that a dynamic analysis of networks is indeed as hard as it can be – it is an intractable problem.

Appendix
I. Modelling the network

Here we present a formal model of the network used in the paper. Although the model could be described in the spirit of formal languages, we believe that a less formal overview is sufficient.

A. Model of the network topology

The network topology is a directed graph in which nodes represent switches and edges represent network links. More formally, a network topology consists of (i) a set of switches, each modeled as a node with a set of input/output ports; (ii) a set of network links, each modeled as a directed edge connecting an output port of some switch with an input port of another switch. In addition to the traditional switches, we have a special "ingress" and "egress" switches each containing a set of ports. These switches model endpoints of links external to our network. In the rest of the paper, we will not explicitly include these special switches in our models but the reader should be aware of their (implicit) presence.

Switches and their behavior. The integral part of our model is the representation of the switch. A switch is a device with a set of input ports P_{in} and a set of output ports P_{out}. To model switch behavior, we borrow the idea of switch *transfer function* from [KVM12]. The idea is that the whole switch can be represented as a function Φ that takes a packet and an input port as its arguments, and produces a set of packets with corresponding output ports. If the size of the output set is zero, the packet is dropped, otherwise it is forwarded (with possible modifications) to one or multiple destinations. However, we need to extend this definition to also take the internal state of the switch into account. Therefore, each switch is associated with a set S of possible states and the transfer function is defined as

$$\Phi(pkt_{in}, port_{in}, state) \mapsto (\{(pkt_{out1}, port_{out1}), \dots\}, state')$$

where $pkt_{in}, pkt_{out} \in PKT$ (a set of all possible packets), $port_{in} \in P_{in}$, $port_{out} \in P_{out}$ and $state, state' \in S$. We denote the first part of the transfer function as *forwarding function* F and the second part as *state change functions* Ψ. We postpone the modeling of packets (definition of PKT) and possible transfer functions Φ to the next subsections.

B. Model of the network behavior

To model the network behavior, we need to describe (i) what is the state of the network; (ii) how network state can change and to which state it will change.

Network state The state of the network consists of state of all the switches and location of all packets in flight. To model the queuing behavior of the switch, we assign a first-in-first-out (FIFO) queue to each switch port. We abstract out packets that are currently being transmitted over the network links by assuming that network link transitions are instantaneous, and that the packets are directly placed from queue of the output port of one switch to queue of the input port of the corresponding switch.

The state thus can be summarized as a tuple $(sw1\_state, sw2\_state, \ldots, port1\_queue, port2\_queue, \ldots)$ where we include the state of all switches and ports. We note that we do not include network topology in the state – that is, in our model we assume that the network topology is fixed during the whole time.

Network state changes (transitions) There are two possibilities as to how a state of the network can change: (i) transmitting a packet over a network link; and (ii) processing a packet at a switch. We will call all such possible state changes *network state transitions* and we will denote a possible transition as $old\_state \rightarrow new\_state$. Because our model is asynchronous, if the network is in state net_state and there are several possible transitions, the new state net_state' can be the result of any of the possible transitions; however, only one transition can be applied at any time (no concurrent transitions).

We note that as defined, network state transitions do not take into the account the possibility of spontaneous switch state change, *e.g.*, because of timers. Again, one might include this possibility in the definition if there is a need for it.

Transmitting packets over links: Let net_state be a network state in which an output port $port_{out}$ contains a non-empty queue. Let $port\_in$ be an input port connected by a link to $port_{out}$. Then there exists a possible network transition $net\_state \rightarrow net\_state'$ with new state net_state' obtained from net_state by

1) extracting the *first* packet pkt from queue of port $port_{out}$, and
2) pushing the packet into the queue of port $port_{in}$.

Processing packets at a switch: Similarly to packet forwarding, let net_state be the network state and let $port_{in}$ be the input port of the switch sw. Moreover, let s be the state of the switch sw, pkt be the first packet from the queue of $port_{in}$ and $\Phi = (F, \Psi)$ be the transfer function of the switch sw. Then, there exists a transition $net\_state \rightarrow net\_state'$ to the new network state net_state' obtained from net_state by

1) extracting the packet pkt from queue of port prt_{in},
2) for each tuple $(pkt_{new}, port_{out}) \in F(pkt, port\_in, s)$ pushing the packet pkt_{new} to the queue of output port $port_{out}$, and
3) updating switch state to $s' = \Psi(pkt, port\_in, s)$

Modeling packets The last part of the puzzle is how to model the packets, *i.e.*, the set PKT. Because packet processing depends only on the packet header, we will abstract out the rest of the packet. Moreover, we abstract the packet header as a tuple $(h_1, h_2, \ldots, h_n)$ of n *independent* header fields. Although this is not true for all header fields combination (*e.g.*, IP_SRC field does not make sense unless the ethernet protocol is equal to 0x800 (IP)), nevertheless there are combinations of fields that are independent (*e.g.*, for IP protocol there are source IP, destination IP, source port, destination port, type of service, ...). Each header field h_i can hold a value from all possible values of that field, $h_i \in H_i$. Except for the packet counting example in Figure 1, we assume that $n = 1$ and that $H_1 = \{\texttt{normal, clr, clr2}\}$.

C. Formal model of basic elements

Here we describe the transfer function for each element.

`split:` There is no internal switch state (the state change function is trivially $\Psi(*, *, *) = \emptyset$ and the forwarding function is $F(pkt, port_{in}, \emptyset) = \{(pkt, port_{out1}), (pkt, port_{out2}, \ldots), \}$

`merge:` There is no internal switch state and the transfer function is $F(pkt, *, \emptyset) = \{(pkt, port_{out})\}$.

`rrobin:` The set of possible states is $S = \{1, 2, \ldots, m\}$ with initial state $s = 1$ and state change function $\Psi(*, *, s) = s + 1$ iff $s < m$ and $\Psi(*, *, m) = 1$. The forwarding function is $F(pkt, *, s) = \{(pkt, port_{out_s})\}$.

`rewrite:` There is no internal state and the forwarding function is $F(pkt, port_{in}, \emptyset) = \{(rewrite(pkt), port_{out})\}$.

`conditional:` There is not internal switch state. The forwarding function is

$$F(pkt, port_{in}, \emptyset) = \begin{cases} \{(pkt, port_{yes})\} & \text{if } pkt \text{ matches} \\ \{(pkt, port_{no})\} & \text{otherwise} \end{cases}$$

REFERENCES

[AB07] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 2007. http://www.cs.princeton.edu/theory/complexity/circuitschap.pdf.

[BDC+89] T.F. Bowen, F.S. Dworack, C.H. Chow, N. Griffeth, G.E. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *SETSS*, 1989.

[BFJB01] Albert-Laszlo Barabasi, Vincent W. Freeh, Hawoong Jeong, and Jay B. Brockman. Parasitic computing. *Nature*, 412(6850):894–897, August 2001.

[CC90] Irving M. Copi and Carl Cohen. *Introduction to logic*. MacMillan, 1990.

[CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115 – 141, 2003.

[End] Herbert Enderton. *A Mathematical Introduction to Logic, Second Edition*.

[Koh03] G.A. Kohring. Implicit simulations using messaging protocols. *International Journal of Modern Physics C*, 14(2):203–213, 2003.

[KSW98] John Kelsey, Bruce Schneier, and David Wagner. Protocol interactions and the chosen protocol attack. In *Proceedings of the 5th International Workshop on Security Protocols*, 1998.

[KVM12] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[MKA+11] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *SIGCOMM*, SIGCOMM '11, 2011.

[NW06] Turlough Neary and Damien Woods. P-completeness of cellular automaton rule 110. In *ICALP*, pages 132–143. Springer, 2006.

[Wol86] Stephen Wolfram. *Theory and Applications of Cellular Automata*. World Scientific, 1986.

[XZM+05] G.G. Xie, Jibin Zhan, D.A. Maltz, Hui Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *INFOCOM*, 2005.