Advanced Topics in Communication Networks

Programming Network Data Planes

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

Sep 27 2018

Materials inspired from Jennifer Rexford, Changhoon Kim, and p4.org
Last week on
Advanced Topics in Communication Networks
Networking is on the verge of a paradigm shift towards *deep programmability*
Why? It's really a story in 3 stages
Stage 1

The network management crisis
“Human factors are responsible for 50% to 80% of network outages”

Ironically, this means that data networks work better during week-ends…

source: Job Snijders (NTT)
Stage 2

Software-Defined Networking
What is SDN and how does it help?

• SDN is a new approach to networking
  – Not about “architecture”: IP, TCP, etc.
  – But about design of network control (routing, TE,...)

• SDN is predicated around two simple concepts
  – Separates the control-plane from the data-plane
  – Provides open API to directly access the data-plane

• While SDN doesn’t do much, it enables a lot
Traditional Computer Networks

Data plane:
Packet processing & delivery

Forward, filter, buffer, mark, rate-limit, and measure packets
Traditional Computer Networks

Control plane:
Distributed algorithms, establish state in devices

Track topology changes, compute routes, install forwarding rules
Software Defined Networking (SDN)

Logically-centralized control

Smart, slow

API to the data plane (e.g., OpenFlow)

Dumb, fast

Switches
OpenFlow is an API to a switch flow table

- Simple packet-handling rules
  - Pattern: match packet header bits, i.e. flowspace
  - Actions: drop, forward, modify, send to controller
  - Priority: disambiguate overlapping patterns
  - Counters: #bytes and #packets

10. src=1.2.*.*, dest=3.4.5.* → drop
05. src = *.*.*.*, dest=3.4.*.* → forward(2)
01. src=10.1.2.3, dest=*.*.*.* → send to controller
Stage 3

Deep Network Programmability
OpenFlow is not all roses

The protocol is too complex  
12 fields in OF 1.0 to 41 in 1.5
switches must support complicated parsers and pipelines

The specification itself keeps getting more complex
extra features make the software agent more complicated

consequences  
Switches vendor end up implementing parts of the spec.
which breaks the abstraction of one API to rule-them-all
Enters… Protocol Independent Switch Architecture and P4

P4: Programming Protocol-Independent Packet Processors

* Barefoot Networks  Intel  Stanford University  **Princeton University  *Google  **Microsoft Research

ABSTRACT
P4 is a high-level language for programming protocol-independent packet processors. P4 works in conjunction with SDN control protocols like OpenFlow. In its current form, OpenFlow explicitly specifies protocol headers on which it operates. This set has grown from 12 to 41 fields in a few years, increasing the complexity of the specification while still not providing the flexibility to add new headers. In this paper we propose P4 as a strawman proposal for how OpenFlow should evolve in the future. We have three goals: (1) Reconfigurability in the field: Programmers should be able to change the way switches process packets once they are deployed. (2) Protocol independence: Switches should not be tied to any specific network protocols. (3) Target independence: Programmers should be able to describe packet-processing functionality independently of the specifics of the underlying hardware. As an example, we describe how to use P4 to configure a switch to add a new hierarchical label multiple stages of rule tables, to allow switches to expose more of their capabilities to the controller.

The proliferation of new header fields shows no signs of stopping. For example, data-center network operators increasingly want to apply new forms of packet encapsulation (e.g., NVGRE, VXLAN, and STT), for which they resort to deploying software switches that are easier to extend with new functionality. Rather than repeatedly extending the OpenFlow specification, we argue that future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through a common, open interface (i.e., a new “OpenFlow 2.0” API). Such a general, extensible approach would be simpler, more elegant, and more future-proof than today’s OpenFlow 1.x standard.
P4 is a high-level language for programming protocol-independent packet processors.
P4 is a high-level language for programming protocol-independent packet processors

P4 specifies packet forwarding behaviors
enables to redefine packet parsing and processing

P4 is protocol-independent
the programmer defines packet headers & processing logic

P4 is target-independent
data plane semantic and behavior can be adapted
IP forwarding in P4?

1.2.3.4  1.2.3.5  1.2.3.254
LAN 1

1.2.3.0/24
5.6.7.0/24
...
forwarding table

5.6.7.1  5.6.7.2  5.6.7.200
LAN 2
This week on

Advanced Topics in Communication Networks
We will start diving into the P4 ecosystem and look at our first practical usage.

Next week: Stateful data plane programming
Probabilistic data structures (beginning)
What is needed to program in P4?

- P4 environment
- P4 language
- P4 in practice
Quick historical recap

- **July 2014**: Initial paper
- **September 2014**: \(P4_{14}\) v1.0.1, v1.0.2, v1.0.3, v1.0.4
- **December 2016**: \(P4_{16}\) specification (draft)
- **May 2017**: \(P4_{16}\) specification
- **2018**:
P4 introduces the concept of an architecture

- **P4 Target**
  - a model of a specific hardware implementation

- **P4 Architecture**
  - an API to program a target
Programming a P4 target involves a few key elements
P4 Program

Compiler

Target

Architecture Model

Data Plane

Tables

Externs

CPU port

target-specific binary

User supplied

Vendor supplied
We'll rely on a simple P4$_{16}$ switch architecture (v1model) which is roughly equivalent to "PISA"
Each architecture defines the metadata it supports, including both standard and intrinsic ones.

```c
v1model struct standard_metadata_t {
    bit<9> ingress_port;
    bit<9> egress_spec;
    bit<9> egress_port;
    bit<32> clone_spec;
    bit<32> instance_type;
    bit<1> drop;
    bit<16> recirculate_port;
    bit<32> packet_length;
    bit<32> enq_timestamp;
    bit<19> enq_qdepth;
    bit<32> deq_timedelta;
    bit<19> deq_qdepth;
    error parser_error;
} bit<48> ingress_global_timestamp;
bit<48> egress_global_timestamp;
bit<32> If_field_list;
bit<16> mcast_grp;
bit<32> resubmit_flag;
bit<16> egress_rid;
bit<1> checksum_error;
bit<32> recirculate_flag;
}
```

https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

more info
Each architecture also defines a list of "externs", i.e. blackbox functions whose interface is known.

Most targets contain specialized components which cannot be expressed in P4 (e.g. complex computations).

At the same time, P4\textsubscript{16} should be target-independent. In P4\textsubscript{14} almost 1/3 of the constructs were target-dependent.

Think of externs as Java interfaces only the signature is known, not the implementation.
v1model

extern register<T> {
    register(bit<32> size);
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}

extern void random<T>(out T result, in T lo, in T hi);

extern void hash<O, T, D, M>(out O result,
    in HashAlgorithm algo, in T base, in D data, in M max);

extern void update_checksum<T, O>(in bool condition,
    in T data, inout O checksum, HashAlgorithm algo);

+ many others (see below)
≠ architectures → ≠ metadata & ≠ externs

NetFPGA-SUME

P4→NetFPGA Compilation Overview

P4 Program
\[\text{Xilinx \(P4_{16}\) Compiler}\]
\[\text{Xilinx SDNet Tools}\]

SimpleSumeSwitch Architecture

NetFPGA Reference Switch

more info
/* standard sume switch metadata */

struct sume_metadata_t {
    bit<16> dma_q_size;
    bit<16> nf3_q_size;
    bit<16> nf2_q_size;
    bit<16> nf1_q_size;
    bit<16> nf0_q_size;
    bit<8> sendDig_to_cpu; // send digest_data to CPU
    bit<8> dst_port; // one-hot encoded
    bit<8> src_port; // one-hot encoded
    bit<16> pkt_len; // unsigned int
}

• _q_size – size of each output queue, measured in terms of 32-byte words, when packet starts being processed by the P4 program
• src_port/dst_port – one-hot encoded, easy to do multicast
• user_metadata/digest_data – structs defined by the user

P4→NetFPGA Extern Function library

- Implement platform specific functions
  - Black box to P4 program
- Implemented in HDL
- Stateless – reinitialized for each packet
- Stateful – keep state between packets
- Xilinx Annotations
  - @Xilinx_MaxLatency() – maximum number of clock cycles an extern function needs to complete
  - @Xilinx_ControlWidth() – size in bits of the address space to allocate to an extern function

Deeper dive into the language constructs (*)

(*) full info  https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html
```
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;
typedef bit<32> ip4Addr_t;
header ipv4_t {...}
struct headers {...}

parser MyParser(...) {
    state start {...}
    state parse_ethernet {...}
    state parse_ipv4 {...}
}

control MyIngress(...) {
    action ipv4_forward(...) {...}
    table ipv4_lpm {...}
    apply {
        if (...) {...}
    }
}

control MyDeparser(...) {...}

V1Switch(
    MyParser(),
    MyVerifyChecksum(),
    MyIngress(),
    MyEgress(),
    MyComputeChecksum(),
    MyDeparser()
) main;
```
But first, the basics:

data types, operations, and statements
**P4_{16}** is a statically-typed language with base types and operators to derive composed ones.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Boolean value</td>
</tr>
<tr>
<td>bit&lt;\text{w}&gt;</td>
<td>Bit-string of width \text{W}</td>
</tr>
<tr>
<td>int&lt;\text{w}&gt;</td>
<td>Signed integer of width \text{W}</td>
</tr>
<tr>
<td>varbit&lt;\text{w}&gt;</td>
<td>Bit-string of dynamic length \leq \text{W}</td>
</tr>
<tr>
<td>match_kind</td>
<td>describes ways to match table keys</td>
</tr>
<tr>
<td>error</td>
<td>used to signal errors</td>
</tr>
<tr>
<td>void</td>
<td>no values, used in few restricted circumstances</td>
</tr>
<tr>
<td>float</td>
<td>not supported</td>
</tr>
<tr>
<td>string</td>
<td>not supported</td>
</tr>
</tbody>
</table>
P4_{16} is a statically-typed language with base types and operators to derive composed ones

Header

```p4
header Ethernet_h {
  bit<48> dstAddr;
  bit<48> srcAddr;
  bit<16> etherType;
}
```

```p4
Ethernet_h ethernetHeader;
```
Think of a header as a struct in C containing the different fields plus a hidden "validity" field

```c
header Ethernet_h {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}
```

Parsing a packet using `extract()` fills in the fields of the header from a network packet.

A successful `extract()` sets to true the validity bit of the extracted header.
P4₁₆ is a statically-typed language with base types and operators to derive composed ones.
P4₁₆ is a statically-typed language with base types and operators to derive composed ones

Struct
Unordered collection of named members

```
struct standard_metadata_t {
    bit<9> ingress_port;
    bit<9> egress_spec;
    bit<9> egress_port;
    ...
}
```

Tuple
Unordered collection of unnamed members

```
tuple<bit<32>, bool> x;
x = { 10, false };
```
P4\textsubscript{16} is a statically-typed language with base types and **operators to derive composed ones**

- **enum**
  ```
  enum Priority {High, Low}
  ```

- **type specialization**
  ```
  typedef bit<48> macAddr_t;
  ```

- **extern**
  ```
  ...
  ```

- **parser**
  ```
  ...
  ```

- **control**
  ```
  ...
  ```

- **package**
  ```
  ...
  ```

**more info**  [https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html](https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html)
P4 operations are similar to C operations and vary depending on the types (unsigned/signed ints, ...)

- arithmetic operations: +, −, *
- logical operations: ~, &, |, ^, >>, <<
- non-standard operations: [m:l] Bit-slicing
  ++ Bit concatenation
- × no division and modulo (can be approximated)

more info: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html
Constants, variable declarations and instantiations are pretty much the same as in C too.

Variable

```c
bit<8> x = 123;

typedef bit<8> MyType;
MyType x;
x = 123;
```

Constant

```c
const bit<8> X = 123;

typedef bit<8> MyType;
const MyType x = 123;
```

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html
Variables have local scope and their values is not maintained across subsequent invocations.

**Important**

Variables *cannot* be used to maintain state between different network packets.

Instead, you can only use two stateful constructs:

- **tables**
  - modified by control plane
- **extern objects**
  - modified by control plane & data plane

More on this next week.
P4 statements are pretty classical too

Restrictions apply depending on the statement location

return terminates the execution of the action or control containing it

exit terminates the execution of all the blocks currently executing

Conditions if (x==123) {...} else {...} not in parsers

Switch switch (t.apply().action_run) {
  action1: {...}
  action2: {...}
}

only in control blocks

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html
Parser

Match-Action Pipeline

Deparser
The parser uses a state machine to map packets into headers and metadata.

<table>
<thead>
<tr>
<th>Packet</th>
<th>Headers and metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:b:c:d → 1:2:3:4</td>
<td>meta {ingress_port: 1, …}</td>
</tr>
<tr>
<td>1.2.3.4 → 5.6.7.8</td>
<td>ethernet {srcAddr: a:b:c:d, …}</td>
</tr>
<tr>
<td>1234 → 56789</td>
<td>ipv4 {srcAddr: 1.2.3.4, …}</td>
</tr>
<tr>
<td></td>
<td>tcp {srcPort: 12345, …}</td>
</tr>
</tbody>
</table>

Payload
parser MyParser(...) {
    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            0x800: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            6: parse_tcp;
            17: parse_udp;
            default: accept;
        }
    }

    state parse_tcp {
        packet.extract(hdr.tcp);
        transition accept;
    }

    state parse_udp {
        packet.extract(hdr.udp);
        transition accept;
    }
}

Recap
The last statement in a state is an (optional) transition, which transfers control to another state (inc. accept/reject).

```plaintext
state start {
    transition parse_ethernet;
}

state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        0x800: parse_ipv4;
        default: accept;
    }
}
```

Go directly to parse_ethernet

Next state depends on etherType
Defining (and parsing) custom headers allow you to implement your own protocols.
A simple example for tunneling
header myTunnel_t {
    bit<16> proto_id;
    bit<16> dst_id;
}

struct headers {
    ethernet_t   ethernet;
    myTunnel_t   myTunnel;
    ipv4_t       ipv4;
}

parser MyParser(...) {
    state start {...}

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            0x1212: parse_myTunnel;
            0x800: parse_ipv4;
            default: accept;
        }
    }

    state parse_myTunnel {
        packet.extract(hdr.myTunnel);
        transition select(hdr.myTunnel.proto_id) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {...}
}
P4 parser supports both fixed and variable-width header extraction

```
header IPv4_no_options_h {
    bit<32> srcAddr;
    bit<32> dstAddr;
}

header IPv4_options_h {
    varbit<320> options;
}
...
parser MyParser(...) {
    ...
    state parse_ipv4 {
        packet.extract(headers.ipv4);
        transition select (headers.ipv4.ihl) {
            5: dispatch_on_protocol;
            default: parse_ipv4_options;
        }
    }
    state parse_ipv4_options {
        packet.extract(headers.ipv4_options, (headers.ipv4.ihl - 5) << 2);
        transition dispatch_on_protocol;
    }
    ...
```
Parsing a header stack requires the parser to loop the only “loops” that are possible in P4
Header stacks for source routing
```c
header srcRoute_t {
    bit<1>    bos;
    bit<15>   port;
}

struct headers {
    ethernet_t              ethernet;
    srcRoute_t[MAX_HOPS]    srcRoutes;
    ipv4_t                  ipv4;
}

parser MyParser(...) {
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_SRCROUTING: parse_srcRouting;
            default: accept;
        }
    }

    state parse_srcRouting {
        packet.extract(hdr.srcRoutes.next);
        transition select(hdr.srcRoutes.last.bos) {
            1: parse_ipv4;
            default: parse_srcRouting;
        }
    }
    accept
    reject
}
```
The parser contains more advanced concepts
check them out!

- verify
- lookahead
- sub-parsers

error handling in the parser
access bits that are not parsed yet
like subroutines

more info: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html
Control

- Tables
  - match a key and return an action

- Actions
  - similar to functions in C

- Control flow
  - similar to C but without loops
Control

- **Tables**: match a key and return an action
- **Actions**: similar to functions in C
- **Control flow**: similar to C but without loops
<table>
<thead>
<tr>
<th>Field(s) to match</th>
<th>Table name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match type</td>
<td>Field(s) to match</td>
</tr>
<tr>
<td>Possible actions</td>
<td>Match type</td>
</tr>
<tr>
<td>Max. # entries in table</td>
<td>Possible actions</td>
</tr>
<tr>
<td>Default action</td>
<td>Max. # entries in table</td>
</tr>
</tbody>
</table>

```plaintext
table {
  key = {
    ;
  };
  size = ;
  default_action = ;
}
```
table ipv4_lpm {
  key = {
    hdr.ipv4.dstAddr: lpm;
  }
  actions = {
    ipv4_forward;
    drop;
  }
  size = 1024;
  default_action = drop();
}
Tables can match on one or multiple keys in different ways.

```table Fwd {
    key = {
        hdr.ipv4.dstAddr : ternary;
        hdr.ipv4.version : exact;
    }
    ...
}
Match types are specified in the P4 core library and in the architectures

- **exact**: exact comparison
  - `0x01020304`

- **ternary**: compare with mask
  - `0x01020304 & 0x0F0F0F0F`

- **lpm**: longest prefix match
  - `0x01020304/24`

- **range**: check if in range
  - `0x01020304 - 0x010203FF`

...
Table entries are added through the control plane

```
control_plane

Table entries are added through the control plane.

<table>
<thead>
<tr>
<th>Key</th>
<th>ID</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2.3.0/24</td>
<td>1</td>
<td>01:...</td>
</tr>
<tr>
<td>5.6.7.0/24</td>
<td>1</td>
<td>02:...</td>
</tr>
</tbody>
</table>

table_add ipv4_lpm ipv4_forward 1.2.3.0/24 => 01:01:01:01:01:01 1
table_add ipv4_lpm ipv4_forward 5.6.7.0/24 => 02:02:02:02:02:02 2
```
Control

- **Tables**: match a key and return an action
- **Actions**: similar to functions in C
- **Control flow**: similar to C but without loops
Actions are blocks of statements that possibly modify the packets
Actions usually take directional parameters indicating how the corresponding value is treated within the block.
Directions can be of three types

- **in**: read only inside the action like parameters to a function.
- **out**: uninitialized, write inside the action like return values.
- **inout**: combination of in and out like “call by reference.”
Let's reconsider a known example

```
action reflect_packet(
inout bit<48> src,
inout bit<48> dst,
in bit<9> inPort,
out bit<9> outPort
)
{
    bit<48> tmp = src;
    src = dst;
    dst = tmp;
    outPort = inPort;
}

reflect_packet(
    hdr.ethernet.srcAddr,
    hdr.ethernet.dstAddr,
    standard_metadata.ingress_port,
    standard_metadata.egress_spec
);
```
reflect_packet

inout bit<48> src
inout bit<48> dst
in bit<9> inPort
out bit<9> outPort
Actions parameters resulting from a table lookup do not take a direction as they come from the control plane

```
action set_egress_port(bit<9> port) {
    standard_metadata.egress_spec = port;
}
```
Control flow

- Tables
  - match a key and return an action

- Actions
  - similar to functions in C

- Control flow
  - similar to C but without loops
Interacting with tables from the control flow

- Applying a table
  
  ```python
  ipv4_lpm.apply()
  ```

- Checking if there was a hit
  
  ```python
  if (ipv4_lpm.apply().hit) {...}
  else {...}
  ```

- Check which action was executed
  
  ```python
  switch (ipv4_lpm.apply().action_run) {
      ipv4_forward: { ... }
  }
  ```
Example: L3 forwarding with multiple tables

IP Packet

<table>
<thead>
<tr>
<th>ipv4_lpm</th>
<th>forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1.0/24</td>
<td>1</td>
</tr>
<tr>
<td>2.2.2.0/24</td>
<td>10</td>
</tr>
<tr>
<td>3.3.3.0/24</td>
<td>12</td>
</tr>
<tr>
<td>4.4.4.0/24</td>
<td>30</td>
</tr>
</tbody>
</table>

Map a prefix to a next hop index

Map a next hop index to an egress port
Example: L3 forwarding with multiple tables

```plaintext
table ipv4_lpm {
  key = {
    hdr.ipv4.dstAddr: lpm;
  }
  actions = {
    set_nhops_index;
    drop;
    NoAction;
  }
  size = 1024;
  default_action = NoAction();
}

table forward {
  key = {
    meta.nhop_index: exact;
  }
  actions = {
    _forward;
    NoAction;
  }
  size = 64;
  default_action = NoAction();
}
```
Applying multiple tables in sequence and checking whether there was a hit

control MyIngress(...) {
    action drop() {...}
    action set_nhport_index(...) 
    action _forward(...) 
    table ipv4_lpm {...} 
    table forward {...} 

    apply {
        if (hdr.ipv4.isValid()){
            if (ipv4_lpm.apply().hit) {
                forward.apply();
            }
        }
    }
}
Validating and computing checksums

```c
extern void verify_checksum<T, O>( in bool condition,
    in T data,
    inout O checksum,
    HashAlgorithm algo);
```

```c
extern void update_checksum<T, O>( in bool condition,
    in T data,
    inout O checksum,
    HashAlgorithm algo);
```
Re-computing checksums
(e.g. after modifying the IP header)

control MyComputeChecksum(...) {
    apply {
        update_checksum(
            hdr.ipv4.isValid(),
            { hdr.ipv4.version,
                hdr.ipv4.ihl,
                hdr.ipv4.diffserv,
                hdr.ipv4.totalLen,
                hdr.ipv4.identification,
                hdr.ipv4.flags,
                hdr.ipv4.fragOffset,
                hdr.ipv4.ttl,
                hdr.ipv4.protocol,
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
        }
    }
}
Control flows contain more advanced concepts
check them out!

- cloning packets: create a clone of a packet
- sending packets to control plane: using dedicated Ethernet port, or target-specific mechanisms (e.g. digests)
- recirculating: send packet through pipeline multiple times

more info: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html
Packet Headers

- ethernet {srcAddr: a:b:c:d, ...}
- ipv4 {srcAddr: 1.2.3.4, ...}
- tcp {srcPort: 12345, ...}

Deparser

- a:b:c:d → 1:2:3:4
- 1.2.3.4 → 5.6.7.8
- 1234 → 56789

Payload

Recap

control MyDeparser(...) {
   apply {
      packet.emit(hdr.ethernet);
      packet.emit(hdr.ipv4);
      packet.emit(hdr.tcp);
   }
}

```
"Full circle"
P4 environment

P4 language

P4 in practice

in-network obfuscation

[USENIX Sec'18]
Advanced Topics in Communication Networks

Programming Network Data Planes

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

Sep 27 2018