SDN controllers can “go wrong”

Software-defined networking (SDN) offers unprecedented control over network operation, allowing network operators programmatic control over switches’ forwarding behavior. In the compass-rose metaphor for networks, an SDN controller sends commands that modify switches’ forwarding tables (so-called flowmods), queues, counters, etc., by means of the southbound API. Several different southbound APIs exist. OpenFlow [9] is surely the most popular of the SDN APIs, but others exist [1, 5].

SDN replaces old but working legacy distributed algorithms like OSPF with new controller code—which, like all code, will have bugs. Some of these bugs will manifest as bad forwarding behaviors, but others will manifest as southbound protocol errors. As SDN has matured, controllers’ capabilities have expanded— and the southbound APIs have expanded with them. For example, OpenFlow 1.0 specifies that a switch consists of a single match/action table; by OpenFlow 1.1, a switch can have any acyclic topology of tables, and by OpenFlow 1.3 each table can have varying capabilities. Expanding the capabilities of SDN is all to the good, but these new features aren’t free: they introduce new failure modes. OpenFlow 1.0 controllers can only send rules matching a fixed set of headers to a single table, so there are only a few kinds of bad rules that can be sent (e.g., matching on IP fields before checking that the Ethertype is 0x0800 ). In OpenFlow 1.3, there are new kinds of mistakes to be made: sending a rule to the wrong table; sending a match, action, or instruction to a table that doesn’t support it; or sending an instruction to a table indicating an invalid next table in the Goto-Table instruction.

Type systems for SDN controllers

What can we do about bugs in SDN controllers? We propose to apply a classic bug prevention technique: type systems. Given a way that a programming language can “go wrong”, type systems identify a conservative set of programs that don’t go wrong [10]. In Milner’s classic paper, “wrong” means applying a boolean instead of a function or conditioning on a function instead of a boolean. Writing SDN controllers, the notion of wrong is quite different. Flipping through the OpenFlow specifications, we can see many ways SDN controllers might go wrong:

(OFPFFMFC_TABLE_BADDAY) Sending rules to non-existent tables.

(OFFPET_BAD_INSTRUCTION) Sending rules (matches, instructions, or actions) to tables that don’t support them, e.g., sending L2 routing rules to the L3 ACL table.

(OFPBAC_MATCH_INCONSISTENT) Sending rules that don’t respect protocol invariants, e.g., matching on the source IP address before checking that the packet is an IP packet—optionally.

1 “The effect of any inconsistent actions on matched packets is undefined. Controllers are strongly encouraged to avoid generating combinations of table entries that may yield inconsistent actions.” [2], p.38

There are other forms of “soft” failure, which generally produce dropped packets instead of error messages, and can be perniciously hard to detect:

– Exhausting switch resources by, e.g., never responding to Packet-In message with a corresponding Packet-Out message.
– Failing to ensure in-order processing, e.g. forgetting to send a Barrier message between sending Flow-Med and Packet-Out messages.

Finally, controllers can be subject to denial-of-service attacks: it may be that certain packets can trigger expensive computations on the controller or cause the controller to issue a large number of rules very quickly.

Among all the different ways SDN controllers can go wrong there are two particular cases where type systems can be particularly effective. First, there is the case where the controller sends rules that are too “wide”, matching on unavailable fields or using unavailable instructions. Second, there is the case where the controller sends rules that are too “deep”, exceeding the match/action tables’ capacity. These two cases correspond to two traditional ways of using type systems: to ensure that programs communicate with their components respecting a predetermined grammar, and to ensure that programs do not use more resources than are available.

Width correctness

We propose a type-based approach for ensuring that SDN controllers send rules to tables that support them—in particular we ensure that a controller match only on fields that are available and that uses only instructions that are available. We assign to each table a type containing some abstract information about the set of rules that is currently available in the table and ensures that the controller program is well typed with respect to these information.

More precisely, the type of a table is annotated as follows:

(id1 \mapsto M, id2 \mapsto A, \ldots, idn \mapsto MA)

where each id is an identifier for a field or instruction, and every identifier is paired with the operation it is associated with: “Match”, “Action”, or “Match and Action”. A type checker can then check that flowmods are sent to appropriately typed tables: that is, the type checker ensures “width correctness”.1

We have implemented this idea as a Haskell library based on recent extensions of the Haskell’s typechecker that allow for type-level sets [12]. These extensions allow us to use the typechecker to ensure the correctness of the annotations. We sketch here a simple example:

rule1 = Rule (Predicate srcMAC 1234)

2 A mistake seen in the wild by Jennifer Rexford.
without races or violations of the protocol dependencies we de-

scribed above. They don’t consider more dynamic, reactive con-
trollers, synthesizing a whole network-wide policy at a time.

Acknowledgments
This work was supported in part by the NSF under grant CNS
1111520.

References
[1] Network configuration protocol (NETCONF), June 2011. URL http:
org/images/stories/downloads/sdn-resources/
onf-specifications/openflow/openflow-spec-v1.4.0.
pdf.
M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller
programs in software-defined networks. In Programming Language
Design and Implementation (PLDI), 2014.
Higher-order approximate relational refinement types for mechanism
design and differential privacy. In Principles of Programming Lan-
guages (POPL), 2015.
[5] Broadcom. Openflow data path abstraction. URL http:
//www.broadcom.com/products/Switching/
Software-Defined-Networking-Solutions/
OF-DPA-Software.
controllers. In Programming Language Design and Implementation
(PLDI), 2013.
in software-defined networks. In Hot Topics in Software Defined
Networking (HotSDN), 2014.
spacetime leaks. In International Conference on Functional Prog-
gramming (ICFP), 2013.
J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation
in campus networks. SIGCOMM Computing Communications Review,
run-time system for network programming languages. In Principles of
Programming Languages (POPL), 2012.
Haskell Symposium, 2014.
S. Shoham. Decentralizing SDN policies. In Principles of Programming
Languages (POPL), 2015.

Related work
Ball et al. [3] verify connectivity controllers for a simplified table
model where there can’t be any errors in the southbound proto-
col from controllers to switches. Padon et al. [13] generate sound
controllers in a similarly simplified model. Guha et al. [6] verify
that a compiler for NetCore correctly issues commands to switches
without races or violations of the protocol dependencies we de-

\[
\begin{align*}
&\text{prog} :: \text{Controller } ['"SrcMAC" :-> 'M,} \\
&\text{  "DstMAC" :-> 'MA,} \\
&\text{  "Ethtype" :-> 'A]}
\end{align*}
\]

\[
\begin{align*}
\text{prog} = \text{do} \\
\text{  flowMod rule1} \\
\text{  flowMod rule2}
\end{align*}
\]

The rules are defined in a NetCore-like AST [11], and then
used to issue flowmods to a switch in our Controller monad.
Controller is essentially a writer monad, but the key point is
that Haskell’s type checker can identify which fields are used for
matches (M), actions (A), or both (MA). The Controller monad
then tracks which fields are used, allowing a programmer to match
these with the capabilities of a switch.

Depth correctness
Katta et al. [7] use a CPU-cache setup to automatically prevent
a controller from sending too many rules to a switch: when an
overflow is about to occur, they ‘evict’ some rules from the switch
and send them to software switches, just like an eviction from L0
into L1 cache. A type system for depth correctness could not only verify Katta et al.’s controller, but it would allow controllers to
make different compromises between network topology—not all
networks can accommodate a layer of L1 software switches—and
depth correctness.

Depth correctness poses challenges that are very different from
those posed by width correctness. Indeed, to ensure depth correct-
ness we need some technique for tracking the number of rules that
are sent to a table; depth correctness is a form of resource anal-
alysis. The standard tools here are then linearity and dependent or
refinement types. Recent advances in these areas [4] are promising,
but don’t go all the way: there are some novel problems here. Con-
trollers tend to work in two phases: an initial, proactive configura-
tion phase, followed by a reactive population phase. In the former,
proactive phase, the controller immediately issues some number of
rules and configuration directives to switches. After this configu-
ratio phase, the controller becomes reactive, waiting for network
events to incite it action. We believe that Barthe et al. offer a ba-
is for setting up invariants (using refinement types) and enforcing
them with linearity. Even so, previous work on linearity has focused
on non-trivial reactive behavior. Krishnaswami [8] may offer insights
into this.

A type system with robust support for depth correctness also
serves as a tool for analyzing how sensitive controllers are to pack-
}