Automated Network Repair with Meta Provenance

Yang Wu
University of Pennsylvania

Ang Chen
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

Wenchao Zhou
Georgetown University

Boon Thau Loo
University of Pennsylvania

ABSTRACT

When debugging an SDN application, diagnosing the problem is merely the first step – the operator must still implement a solution that works, and that does not cause new problems elsewhere. However, most existing SDN debuggers focus exclusively on identifying the problem and offer the network operator little or no help with finding an effective fix. Finding a fix is challenging because the number of potential repairs can be enormous.

In this paper, we propose a first step towards automated repair for SDN applications. Our approach consists of two elements. The first is a data structure we call *meta provenance*, which can be used to efficiently find good candidate repairs. Meta provenance is inspired by the provenance concept from the database community. However, whereas standard provenance can only reason about changes to data, meta-provenance can also reason about changes to programs. The second element is a system that can efficiently back-test a set of candidate repairs using historical data from the network. This is used to eliminate candidate repairs that do not work well, or that cause other problems. We present initial results from a case study, which suggest our approach is able to efficiently find high-quality repairs.

Categories and Subject Descriptors

D.2.5 [**Testing and debugging**]: Diagnostics

Keywords

Software-defined Networks, Debugging, Provenance

1. INTRODUCTION

Debugging networks is notoriously hard. The advent of software-defined networking (SDN) has added a new dimension to the problem: networks can now be controlled by programs written in software, and, like all other programs, these programs can have bugs.

There is a substantial literature on network debugging and root cause analysis [5, 9, 10, 28, 18, 26, 11]. These tools can offer network operators a lot of help with debugging. For instance, systems like NetSight [9] and negative provenance [26] provide a kind of “backtrace”, analogous to a stack trace in a conventional debugger, that can link an observed effect of a bug (say, packets being dropped in the network) to its root causes (say, an incorrect flow table entry).

However, in practice, diagnosing the problem is only the first step. Once the root cause of a problem is known, the operator must find an effective fix (repair) that not only solves the problem at hand, but also avoids creating *new* problems elsewhere in the network. Given the complexity of modern controller programs and configuration files, finding a good fix can be as challenging as – or perhaps even more challenging than – diagnostics, and it often requires considerable expertise on the part of the operator. However, current tools offer far less help with this second step than with the first. In this paper, we present a first step towards automated bug fixing in SDN applications. Our (slightly idealistic) long-term vision is a “Fix it!” button that will automatically find and fix the root cause of an observed problem. However, we initially aim for something less ambitious, which is to provide the operator with a list of suggested patches.

We believe that it may be possible to reach this goal by leveraging and enhancing some concepts that have been developed in the database community. For some time, this community has been studying the question how to explain the presence or absence of certain data tuples in the result of a database query, and whether and how the query can be adjusted to make certain tuples appear or disappear [3, 25]. By seeing SDN applications as “queries” that operate on a “database” of incoming packets and produce a “result” of delivered or dropped packets, it should be possible to ask similar queries – e.g., why a given packet was absent (misrouted/dropped) from an observed “result”.

The key concept in this line of work is that of *data provenance* [2]. In essence, provenance tracks causality: the provenance of a tuple (or packet, or data item) consists of the tuples from which it was directly derived. By applying this idea recursively, it is possible to trace the provenance of a tuple in the output of a query all the way to the “base tuples” in the underlying databases. The result is a comprehensive causal explanation of how the tuple came to exist. This idea has previously been adapted for the SDN setting

HotNets-XIV, November 16–17, 2015, Philadelphia, PA, USA.

ACM 978-1-4503-4047-2/15/11 ...$15.00.
DOI: http://dx.doi.org/10.1145/2834050.2834112.

as *network provenance*, and it has been used, e.g., in debuggers and forensic tools such as ExSPAN [30], SNP [28] and Y! [26]. However, *so far this work has considered provenance only in terms of packets and configuration data – the SDN controller program was assumed to be immutable*. This is sufficient for diagnosis, but not for repair: we must also be able to infer which parts of the controller program were responsible for an observed event, and how the event might be affected by changes to that program.

In this paper, we take the next step and extend network provenance to *both* programs *and* data. At a high level, we accomplish this with a combination of two ideas. First, we treat programs as just another kind of data; this allows us to reason about the provenance of data not only in terms of the data it was computed from, but also in terms of the parts of the program it was computed *with*. Second, we use a form of counterfactual reasoning to enable a form of negative provenance [26], so that operators can ask why some condition did *not* hold (Example: "Why didn't any DNS requests arrive at the DNS server?"). This is a natural way to phrase a diagnostic query, and the resulting meta provenance is, in essence, a tree of changes (to the program and/or to configuration data) that could make the condition true.

We discuss three key challenges. First, there are infinitely many possible repairs to a given program (including, e.g., a complete rewrite), and not all of them will make the condition hold. We demonstrate ways to find suitable repairs efficiently using properties of the provenance itself. Second, even if we consider only suitable changes, there are still infinitely many possibilities. To overcome this problem, we leverage the fact that most bugs affect only a small part of the program, and that programmers tend to make certain errors more often than others [12, 21]. This allows us to rank the possible changes according to plausibility, and to explore only the most plausible ones. Finally, even a small change that fixes the problem at hand might still cause problems elsewhere in the network. To avoid such fixes, we backtest them using historical information that was collected in the network. In combination, this approach should allow us to return to the operator a list of suggested fixes that 1) are small and plausible, 2) fix the problem at hand, and 3) are unlikely to affect unrelated parts of the network.

We proceed with an overview in Section 2 and then sketch a simple meta-provenance model and the corresponding provenance engine in Section 3. In Section 4, we report results from an initial case study, which suggest that meta provenance can indeed identify the root cause and find useful fixes for realistic network problems automatically.

2. OVERVIEW

We illustrate the problem with a simple scenario (Figure 1). A network operator manages an SDN that connects two web servers and a DNS server to the Internet. To balance the load, incoming web requests are forwarded to different servers based on their source IP. At some point, the administrator notices that web server H2 is not receiving any requests from the Internet.

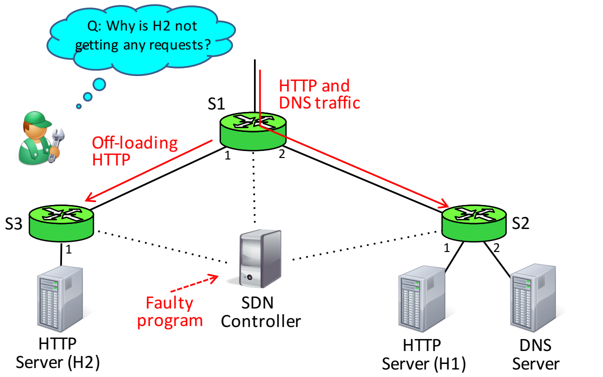


Figure 1: Example scenario. The primary web server (H1) is too busy, so the network is supposed to offload some HTTP requests to a backup web server (H2). However, offloading does not work because of a bug in the controller program.

Our goal is to build a debugger that accepts a simple specification of the observed problem (e.g. "H2 is not receiving any traffic on TCP port 80") and returns a) a detailed causal explanation of the problem, and b) a ranked list of suggested fixes. We consider a suggested fix to be useful if it a) fixes the specified problem and b) has few or no side-effects on the rest of the network.

2.1 Background: Network Datalog

For ease of exposition, we will assume here that the controller programs are written in *network datalog (NDlog)* [16]. The reason is that our approach is based on provenance, and, in a declarative language, provenance is particularly easy to see. (However, our approach is not specific to NDlog or to declarative languages and should work with any other language.) Before we proceed, we briefly review some key features of NDlog.

In NDlog, the state of a node (switch, controller, or server) is modeled as a set of *tables*, which each contain a number of *tuples*. For instance, an SDN switch might contain a table called `FlowTable`, and each tuple in this table might represent a flow entry; an SDN controller might have a table called `PacketIn` that contains the packets it has received from the switches. Tuples can be manually inserted, or they can be programmatically derived from other tuples; the former are called *base tuples*, and the latter are *derived tuples*.

NDlog programs consist of *rules* that describe how tuples should be derived from each other. For example, the rule `A(@X,P):-B(@X,Q),Q=2*P` says that a tuple `A(@X,P)` should be derived on node `X` whenever there is also a tuple `B(@X,Q)` on that node, and `Q=2*P`. Here, `P` and `Q` are variables that must be instantiated with values when the rule is applied: a tuple `B(@X,10)` would create a tuple `A(@X,5)`. The `@` operator specifies the node on which the tuple resides. Rules may include tuples from different nodes; for instance, `C(@X,P):- C(@Y,P)` says that tuples in table `C` on node `Y` should be sent to node `X` and inserted into table `C` there.

```
r1: FlowTable(@C,Swi,Sip,Dpt,Act) :- PacketIn(@C,Swi,Sip,Dpt),Swi==1,Sip in 10.0.1.0/24,Dpt==80,Act:=’Output-1’.
r2: PacketInMSG(@C,Swi,Sip,Dpt) :- PacketIn(@C,Swi,Sip,Dpt).
r3: FlowTableCNT(@C,Swi,Sip,Dpt,Count<*>) :- PacketInMSG(@C,Swi,Sip,Dpt),FlowTable(@C,Swi,Sip,Dpt,Act).
r4: FlowTable(@C,Swi,Sip,Dpt,Act) :- FlowTableCNT(@C,Swi,Sip,Dpt,Cnt),Swi==1,Dpt==80,Cnt==0,Act:=’Output-2’.
r5: FlowTable(@C,Swi,Sip,Dpt,Act) :- PacketIn(@C,Swi,Sip,Dpt),Swi==2,Sip in 10.0.0.0/16,Dpt==80,Act:=’Output-1’.
r6: FlowTable(@C,Swi,Sip,Dpt,Act) :- PacketIn(@C,Swi,Sip,Dpt),Swi==2,Sip in 10.0.0.0/16,Dpt==22,Act:=’Output-2’.
r7: FlowTable(@C,Swi,Sip,Dpt,Act) :- PacketIn(@C,Swi,Sip,Dpt),Swi==2,Sip in 10.0.0.0/16,Dpt==80,Act:=’Output-1’.
```

Figure 2: An SDN controller program written in NDlog. At the aggregation switch S1, certain HTTP requests are offloaded to the left top-of-rack switch S3 (rule `r1`). The remaining traffic is sent to the right top-of-rack switch S2 (rules `r2`+`r3`+`r4`). At S2 and S3, traffic is forwarded to the responsible server based on the destination port (rules `r5`+`r6`+`r7`).

2.2 Classical provenance

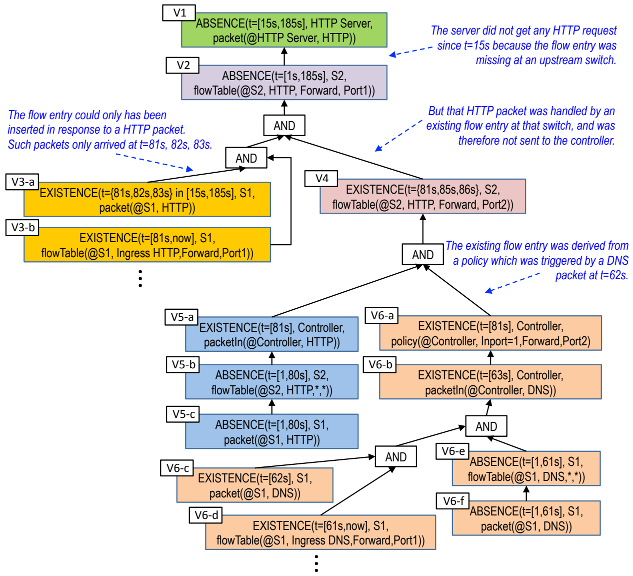
In NDlog, it is easy to see why a given tuple exists: if the tuple was derived using some rule `R`, then it must be the case that all the predicates in `R` were true, and all the constraints in `R` were satisfied. For instance, if a tuple `A(@X,5)` was derived using the rule above, then the reason for the derivation was that the tuple `B(@X,10)` existed, and that `10=2*5`. This concept can be applied recursively (e.g., to explain the existence of `B(@X,10)`) until a set of base tuples is reached that cannot be explained further (e.g., configuration data or packet transmissions). The resulting explanation can be visualized as a *provenance graph*, in which each vertex represents a tuple and in which edges represent direct causality; the tuple at the root is the one that is being explained, and the base tuples are at the leaves.

A similar kind of reasoning can explain why a given tuple *does not* exist [26]: we simply ask, counterfactually, how the tuple *could* be derived, and then explain why each precondition did not hold. The result is, again, a (negative) provenance graph, as shown in Figure 3.

2.3 Case study: Faulty program

We now return to the scenario in Figure 1. One possible reason for this situation is that the administrator has made a copy-and-paste error when writing the controller program. Such errors are common in large software because programmers prefer reusing code via copy-and-paste in order to reduce programming effort [14]. Concretely, suppose that the controller program initially contained the first six rules shown in Figure 2. When the second web server H2 was added, the configuration for switch S3 had to be updated to forward HTTP requests to H2. Perhaps the administrator saw a similar rule that was already in the program (`r5`), which is used for sending HTTP requests from S2 to H1, copied and pasted it to create rule `r7`, but forgot to change the condition that matches the switch (`Swi==2`) to match S3 instead of S2.

When the operator notices that no requests are arriving at H2, she can use a provenance-based debugger to get a causal explanation. Provenance trees are more useful than large packet traces or the system-wide configuration files because they only contain information that is causally related to the observed problem. But they can still be quite complex and require some expertise to interpret – and the operator is still on her own when fixing the bug.


V1
ABSENCE(t=[15s,185s], HTTP Server, packet(@HTTP Server, HTTP))
V2
ABSENCE(t=[1s,185s], S2, flowTable(@S2, HTTP, Forward, Port1))
AND
AND
V3-a
EXISTENCE(t={81s,82s,83s} in [15s,185s], S1, packet(@S1, HTTP))
V3-b
EXISTENCE(t=[81s,now], S1, flowTable(@S1, Ingress HTTP,Forward,Port1))
V4
EXISTENCE(t={81s,85s,86s}, S2, flowTable(@S2, HTTP, Forward, Port2))
AND
V5-a
EXISTENCE(t=[81s], Controller, packetIn(@Controller, HTTP))
V5-b
ABSENCE(t=[1,80s], S2, flowTable(@S2, HTTP,*,*))
V5-c
ABSENCE(t=[1,80s], S1, packet(@S1, HTTP))
V6-a
EXISTENCE(t=[81s], Controller, policy(@Controller, Inport=1,Forward,Port2)
V6-b
EXISTENCE(t=[63s], Controller, packetIn(@Controller, DNS))
AND
AND
V6-c
EXISTENCE(t=[62s], S1, packet(@S1, DNS))
V6-d
EXISTENCE(t=[61s,now], S1, flowTable(@S1, Ingress DNS,Forward,Port1))
V6-e
ABSENCE(t=[1,61s], S1, flowTable(@S1, DNS,*,*))
V6-f
ABSENCE(t=[1,61s], S1, packet(@S1, DNS))

The server did not get any HTTP request since t=15s because the flow entry was missing at an upstream switch.

The flow entry could only has been inserted in response to a HTTP packet. Such packets only arrived at t=81s, 82s, 83s.

But that HTTP packet was handled by an existing flow entry at that switch, and was therefore not sent to the controller.

The existing flow entry was derived from a policy which was triggered by a DNS packet at t=62s.

Figure 3: Excerpt from a negative provenance graph that explains why the HTTP server is no longer receiving traffic.

2.4 Meta provenance

Classical provenance is inherently unable to generate fixes because it reasons about the provenance of data that was generated *by a given program*. To find a fix, we also need the ability to reason about program changes.

We propose to add this capability, in essence, *by treating the program as just another kind of data*. Thus, the provenance of a tuple that was derived via a certain rule `R` does not only consist of the tuples that triggered `R`, but also of the syntactic components of `R` itself. For instance, when generating the provenance that explains why, in the scenario from Figure 1, no HTTP requests are arriving at H2, we eventually reach a point where we must explain the absence of a flow table entry in switch S3 that would send HTTP packets to port #1 on that switch. At this point, we can observe that rule `r7` would *almost* have generated such a flow entry, were it not for the predicate `Swi==2`, which did not hold. We can then, analogous to negative provenance, use counterfactual reasoning to determine that the rule *would* have the desired behavior if the constant were 3 instead of 2. Thus, the fact that the constant in the predicate is 2 and not 3 should become part of the missing flow entry's meta provenance.

2.5 Challenges

An obvious challenge with this approach is that there are infinitely many possible changes to a given program: constants can be changed, predicates can be added or removed from existing rules, rules can be added or deleted, and so on. However, in practice, only a tiny subset of these changes is actually relevant. Observe that, at any point in the provenance tree, we know exactly what it is that we need to explain – e.g., the absence of a particular flow entry for HTTP traffic. Thus, we need not consider changes to `r2` or `r3` (because they do not generate flow entries) or to the destination port (`Dpt`) in `r7` (because that predicate is already true).

Of course, the number of relevant changes, and thus the size of any meta-provenance graph, is still infinite. This does mean that we can never fully draw or materialize it – but there is also no need for that. Studies have shown that "real" bugs are often small [21], such as off-by-one errors or missing predicates. Thus, it seems useful to define a cost metric for changes (perhaps based on the number of syntactic elements they touch), and to explore only the "cheapest" changes based on the defined metric.

Third, it is not always obvious what to change in order to achieve a desired effect. For instance, when changing `Swi==2` in the above example, why did we change the constant to 3 and not, say, 4? Here, the reason is easy to see, but for more complex predicates, it can be harder. Fortunately, we can use existing tools, such as SMT solvers, that can enumerate possibilities quickly for the more difficult cases.

Finally, even if a change fixes the problem at hand, we cannot be sure that it will not cause new problems elsewhere. Such side-effects are difficult to capture in the meta provenance itself, but we show that they can be estimated in another way, namely by back-testing changes with historical information from the network.

3. TOWARDS META PROVENANCE

In this section, we describe how to generate meta provenance for controller programs written in NDlog.

3.1 Provenance graph

Recall from Section 2.2 that the provenance of an event – such as the presence or absence of a tuple – can be described as a graph in which each vertex 1) represents an event, and 2) has children that represent the direct causes that contribute to this event. Thus, a useful way to define a (meta-)provenance graph is to specify an algorithm that can enumerate the children of a given vertex.

We start with an existing algorithm from [26] that can generate positive and negative provenance, but not meta provenance (because it assumes that the rules in the NDlog program are immutable), and we extend this algorithm in two ways. First, we add *meta tuples* that represent the syntactic elements of the program itself, such as conditions and predicates; and second, we add *meta rules* that describe the operational semantics of NDlog. As a first approximation, the meta tuples/rules behave just like normal tuples and rules; thus, the algorithm from [26] essentially works out of the box, with just a few changes we discuss below.

Since NDlog is a relatively simple language, it requires few meta rules. Figure 4 shows the most important ones. Writing these rules is a one-time investment for each given language; once they exist, they can be applied to any program written in that language. For instance, rule `r1` from Figure 2 would be represented as a `HeadMeta` tuple with name `FlowTable`, and a `PredicateMeta` tuple with name `PacketIn`; constraint predicates like `Swi==1` would be represented by an `Expression` tuple as well as an `IsConstraint` tuple.

Notice that the meta provenance is not static; it depends on the state of the network *and* of the controller program. For instance, in our example from Section 2, the bug could have been introduced at some time *t*, and the program could have been fine before that. Thus, a practical debugger should index the provenance by time, and it should have the ability to remember or reconstruct earlier states. This issue is somewhat orthogonal to meta provenance and has been the subject of earlier work [29], so we do not discuss it here.

3.2 Reasoning about program changes

The key difference between meta provenance and classical provenance is the need to additionally consider the possibility that an unexpectedly present (absent) tuple could have disappeared (appeared) if the program were modified in some way. For instance, in our scenario from Section 2, we must eventually explain the absence of a flow entry on switch S3 that would forward HTTP traffic to server H2. Classical provenance would end here – such a flow entry simply does not exist! – but meta provenance must explain the absence by enumerating facts about the program that are responsible for it (e.g., "The second predicate in rule `r7` is `Swi==2` and not `Swi==3`").

We can enumerate such facts using counterfactual reasoning. For instance, suppose the following flow entry is missing: `FlowTable(@C,3,10.0.0.5, 80,Output-1).` What *could have* caused this tuple to exist? It could have been inserted as a base tuple, it could have been generated by some rule that is currently missing, or it could have been generated by a variant of an existing rule. Candidates for the latter are `r1` and `r4-7`. We can then investigate why these rules were not triggered in their current form; for instance, in the case of `r7`, we would probably find incoming packets (i.e., `PacketIn` tuples) with suitable destination IPs and port numbers, but the condition `Swi==2` is inconsistent with the fact that we are looking for a `FlowTable` tuple whose `Swi` element is 3. Thus, the presence of `Swi==2` in rule `r7` is one (of many) reasons for the absence of the missing `FlowTable` tuple.

Of course, there are more complicated cases. Suppose, for instance, that we have a rule `A(X):-B(Y,Z),X:=Y+3, X>5` and a tuple `A(X)` with `X<10` is missing. If a tuple `B(Y,Z)` with `Y=2` already exists, we can derive the miss-

```
h1: Head(Rul,Tab,Vals[]) :- Value(Rul,JID,Arg,Val),HeadMeta(Rul,Tab,Args[]),Arg==Args[0],Vals[0]:=Val,
       Constraint(Rul,JID,ID,Val==True),ConstraintCount(Rul,N==1).
h2: Head(Rul,Tab,Vals[]) :- Value(Rul,JID,Arg0,Val0),Value(Rul,JID,Arg1,Val1),HeadMeta(Rul,Tab,Args[]),
       Arg0==Args[0],Arg1==Args[1],Vals[0]:=Val0,Vals[1]:=Val1,
       Constraint(Rul,JID,ID,Val==True),Constraint(Rul,JID,ID’,Val==True),ID!=ID’,ConstraintCount(Rul,N==2).
h3: Head(Rul,Tab,Vals[]) :- Value(Rul,JID,Arg0,Val0),Value(Rul,JID,Arg1,Val1),HeadMeta(Rul,Tab,Args[]),
       Arg0==Args[0],Arg1==Args[1],Vals[0]:=Val0,Vals[1]:=Val1,
       Constraint(Rul,JID,ID,Val==True),Constraint(Rul,JID,ID’,Val==True),Constraint(Rul,JID,ID”,Val==True)
       ID!=ID’,ID’!=ID”,ID!=ID”,ConstraintCount(Rul,N==3).
d1: Tuple(Tab,Vals[]) :- Head(Rul,Tab,Vals[]),IsDerived(Tab).
d2: Tuple(Tab, Vals[]) :- Base(Tab, Vals[]),IsBase(Tab).
j1: Predicate(Rul,Tab,Args[],Vals[]) :- Tuple(Tab,Vals[]),PredicateMeta(Rul,Tab,Args[]).
j2: Join(Rul,JID,Args[],Vals[]) :- Predicate(Rul,Tab’,Args’[],Vals’[]),Predicate(Rul,Tab”,Args”[],Vals”[]),
       Tab’!=Tab”,Args:=Args’+Args”,Vals:=Vals’+Vals”,JID:=f_unique(),PredicateCount(Rul,N==2).
j3: Value(Rul,JID,Arg,Val) :- Join(Rul,JID,Args[],Vals[]),Arg:=Args[0],Val:=Vals[0].
a1: Value(Rul,JID,Arg,Val) :- Assignment(Rul,Arg,ID),Expression(Rul,JID,ID,Val).
c1: Constraint(Rul,JID,ID) :- Expression(Rul,JID,ID,Val),IsConstraint(Rul,ID).
e1: Expression(Rul,JID,ID,Val) :- Constant(Rul,ID,Val),JID:=*.
e2: Expression(Rul,JID,ID,Val) :- Value(Rul,JID,Arg,Val),ID:=f_string(Arg).
e3: Expression(Rul,JID,ID,Val) :- Expression(Rul,JID,ID’,Val’),Expression(Rul,JID,ID”,Val”),ID’!=ID”,
       Edge(Rul,ID,ID’),Edge(Rul,ID,ID”),Operator(Rul,ID”’,Opr),Edge(Rul,ID,ID”’),Opr==’==’,Val:=(Val’==Val”).
e4: Expression(Rul,JID,ID,Val) :- Expression(Rul,JID,ID’,Val’),Expression(Rul,JID,ID”,Val”),ID’!=ID”,
       Edge(Rul,ID,ID’),Edge(Rul,ID,ID”),Operator(Rul,ID”’,Opr),Edge(Rul,ID,ID”’),Opr==’>’,Val:=(Val’>Val”).
```

Figure 4: Some key meta rules for NDlog programs (base tuples are underlined). Head tuples derive from the variables in the head declaration with instantiated values, and only when constraints are satisfied (rules `h1`–`h3`). Tuples are either derived tuples or base tuples (rules `d1`+`d2`). Variables derive from joining predicate tuples (rules `j1`–`j3`) or arbitrary expressions. Some expressions are constraints (rule `c1`). Expressions derive from constants, variables, or composition (rules `e1`–`e4`).

ing tuple by changing the assignment from `X:=Y+3` to either `X:=Y+4` or `X:=Y+2`. However, only the former will actually work because the latter violates the other constraint, `X>5`. In such difficult cases, we could use a constraint solver, such as Z3 [4], to find constants that satisfy all conditions; however, in our experience, most constraints in actual programs are amenable to simpler methods.

3.3 Enumerating changes

Recall that, in general, there are infinitely many candidate repairs, so we cannot hope to explore all of them. Instead, we define a cost metric and explore the candidates in cost order, up to some reasonable cut-off (or until the operator's patience runs out). We can derive such a metric from prior work in software engineering, such as [21], that has studied the kinds of errors that programmers typically make. By assigning a low cost to common errors (such as changing a constant by one or changing a == to a `!=`) and a high cost to unlikely errors (such as writing an entirely new rule, or defining a new table), we can prioritize the search of fixes to software bugs that are more commonly observed in actual programming, and thus increase the chances that a working fix will be found.

3.4 Backtesting candidate repairs

By itself, meta provenance does not consider the side-effects of a candidate repair; it merely considers whether the repair would solve the problem at hand. To mitigate this problem, we can back-test the candidates using historical information from the network – perhaps a Netflow trace or a sample of packets, along with some statistics, such as throughput and network latency, from the various end-hosts. Since the problems we are aiming to repair are typically subtle (total network failures are comparatively easy to diagnose!), they should affect a relatively small fraction of the traffic. Hence, a "good" candidate repair should have little or no impact on metrics that are not related to the specified problem.

It is important for the backtesting to be fast: the less time it takes, the more candidate repairs we can afford to consider. Fortunately, we should be able to leverage another concept from the database literature for this purpose. Basically, each backtest simulates the behavior of the network with the repaired program, and measures some statistics at the end. Thus, we are effectively running many very similar "queries" (the repaired programs, which differ only in the fixes that were applied) over the same "database" (the historical network data), where we expect significant overlaps among the query computations. This is a classical instance of multi-query optimization, for which powerful solutions are available in the literature [17, 6].

4. CASE STUDY

We have implemented an early prototype of a meta provenance engine for SDNs, based on the algorithm and the heuristics above. To illustrate the results our engine produces, we have set up the scenario from Figure 1 and Figure 2 (our running example) in the RapidNet [1] simulator. We use the meta model shown in Figure 4 in combination with a cost model that is based on two software engineering studies [12, 21] about common programmer errors. For backtesting, we use a simple approach that uses packet counters at the DNS server and the primary web server; we discard changes that cause these counters to change substantially.

To test our prototype, we submitted the problem description from Section 2 ("H2 is not receiving any traffic on TCP port 80"). This caused our prototype to generate more than 80 initial repair candidates, of which ten are shown in Ta-

A	Changing operator in `r7`: `Swi==2` to `Swi!=2`	×
B	Changing constant in `r7`: `Swi==2` to `Swi==3`	✓
C	Changing operator in `r5`: `Swi==2` to `Swi>=2`	✓
D	Changing constant in `r1`: `Swi==1` to `Swi>=1`	✓
E	Changing operator in `r5`: `Swi==2` to `Swi!=2`	×
F	Changing constant in `r5`: `Swi==2` to `Swi==3`	✓
G	Deleting constraint `Swi==1` in `r1`	✓
H	Changing constraint in `r7`: `Swi==2` to `Swi<=3`	×
I	Deleting constraint `Swi==2` in `r5`	✓
J	Changing multiple constraints in `r6`: `Swi==2` to `Swi==3`, and `Dpt==22` to `Dpt==80`	×

Table 1: Candidate repairs listed in generation order. Some candidates are discarded after backtesting.

ble 1. Because the candidates were generated using meta provenance, each of them is effective, i.e., causes H2 to receive at least some HTTP traffic.

Next, the candidates were submitted to backtesting, and candidates that caused problems for the rest of the network were discarded (shown as a cross in the right column of Table 1). Table 2 shows the backtesting results for the discarded candidates. For instance, repair candidate A changed `Swi==2` in rule `r7` to `Swi!=2`. This causes the controller to generate a flow entry that forwards HTTP requests at S3; however, the modified `r7` *also* causes S1 to offload all HTTP requests to the backup web server. Similarly, candidate J's changes to `r6` do fix the problem at hand but also cause DNS queries to be dropped.

After backtesting, the remaining candidates are presented to the operator in complexity order, i.e., the simplest candidate is shown first. In this example, the first candidate on the list (B) is also the one that most human operators would intuitively have chosen – it fixes the copy-and-paste bug by changing the switch number in the faulty predicate from `Swi==2` to `Swi==3`.

This initial result is encouraging, particularly because our meta-provenance model is in no way specific to this particular scenario. This suggests that meta provenance can potentially find useful fixes for a many other problems as well. As ongoing work, we are validating this approach on larger case studies and more sophisticated back-testing techniques.

5. RELATED WORK

Related work on network debugging was already covered in the introduction, so we focus on other related work below.

Databases: There are several interesting connections between meta provenance and work from the database literature. For instance, backtesting could be made efficient with incremental view maintenance [8, 15]; Gupta et al. [7] describes a solution where the definition of the view (in our case, the controller program) can be modified. Multi-query optimization techniques would help as well; for instance, CACQ [17] uses a group filter index to share work between queries on streaming data, and SharedDB [6] aggregates small queries into large queries for batching computations.

A	The primary web server (H1) receives no traffic
E	The primary web server (H1) receives no traffic
H	The primary web server (H1) receives no traffic
J	The DNS server receives no traffic

Table 2: Backtesting results for the discarded candidates.

Meta programming: SecureBlox [19] leverages meta rules to reference the program for certain purposes, and to dynamically change the program during execution (e.g., by generating new rules at runtime to enforce access control). In ExSPAN [30], the provenance graph contains both tuple vertices and rule execution vertices.

Automated program fixing: The software engineering community has been developing techniques such as genetic programming [13] and symbolic execution [20] to adapt programs to a given test suite. We decided against these approaches because, unlike the database work, they aim for fully general programs, which networks do not typically need, and in return are restricted to relatively small programs [20] or certain kinds of patches [13], which may not be enough. Some approaches can work with larger programs: ClearView [22] mines invariants in programs, correlates violations with failures, and generates patches at runtime; ConfDiagnoser [27] compares correct and undesired executions to find suspicious predicates in the program; and Sidiroglou et al. [24] runs attack vectors on instrumented applications and then generates patches automatically. These systems primarily rely on heuristics, whereas our proposed approach uses provenance to track causality and can thus pinpoint specific root causes. Rx [23] recovers failed programs by modifying the environment; this can prevent bugs from manifesting but does not fix their root causes.

6. CONCLUSION

We believe that it is time for SDN debuggers to transition from bug diagnostics tools to automated program fixers. While the eventual goal is ambitious, our paper demonstrates early promise of feasibility. we hypothesize that techniques from the database community – such as data provenance – can be adapted to help us reach this goal. In this paper, we have proposed an approach that is based on data provenance, and we have sketched an extension, called meta provenance, that can reason not only about changes to data but also about changes to programs. Our initial results from a case study seem encouraging, and we hope that meta provenance can eventually diagnose and fix a variety of network problems, with very little help from the human operator.

Acknowledgments: We thank the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF grants CNS-1054229, CNS-1065130, CNS-1117052, CNS-1218066, CNS-1453392, and CNS-1513734; DARPA contract HR0011-15-C-0098; and the Intel-NSF Partnership for Cyber-Physical Systems Security and Privacy (NSF CNS-1505799).

7. REFERENCES

[1] http://netdb.cis.upenn.edu/rapidnet/.

[2] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In *Proc. ICDT*. Jan. 2001.

[3] A. Chapman and H. Jagadish. Why not? In *Proc. SIGMOD*, June 2009.

[4] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, Apr. 2008.

[5] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proc. SIGCOMM*, Aug. 2004.

[6] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. *Proc. VLDB Endowment*, 5(6):526–537, 2012.

[7] A. Gupta, I. S. Mumick, J. Rao, and K. A. Ross. Adapting materialized views after redefinitions: Techniques and a performance study. *Information Systems*, 26(5):323 – 362, 2001.

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD*, May 1993.

[9] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. NSDI*, Apr. 2014.

[10] E. Katz-Bassett, H. V. Madhyastha, J. P. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *Proc. NSDI*, Apr. 2008.

[11] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI*, Apr. 2012.

[12] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. ICSE*, May 2013.

[13] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. ICSE*, June 2012.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proc. OSDI*, Dec. 2004.

[15] M. Liu, N. E. Taylor, W. Zhou, Z. G. Ives, and B. T. Loo. Recursive computation of regions and connectivity in networks. In *Proc. ICDE*, Mar. 2009.

[16] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Comm. ACM*, 52(11):87–95, Nov. 2009.

[17] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. SIGMOD*, June 2002.

[18] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Proc. SIGCOMM*, Aug. 2011.

[19] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. SecureBlox: Customizable secure distributed data processing. In *Proc. SIGMOD*, June 2010.

[20] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proc. ICSE*, May 2013.

[21] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[22] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *Proc. SOSP*, Nov. 2009.

[23] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies. In *Proc. SOSP*, Oct. 2005.

[24] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, Nov. 2005.

[25] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *Proc. SIGMOD*, June 2010.

[26] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems negative provenance. In *Proc. SIGCOMM*, Aug. 2014.

[27] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *Proc. ICSE*, May 2013.

[28] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proc. SOSP*, Oct. 2011.

[29] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *Proc. VLDB*, Aug. 2013.

[30] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proc. SIGMOD*, June 2010.