



Automated Bug Removal for Software-Defined Networks

**Yang Wu, Ang Chen, and Andreas Haeberlen, *University of Pennsylvania*;
Wenchao Zhou, *Georgetown University*; Boon Thau Loo, *University of Pennsylvania***

<https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wu>

**This paper is included in the Proceedings of the
14th USENIX Symposium on Networked Systems
Design and Implementation (NSDI '17).**

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

**Open access to the Proceedings of the
14th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by USENIX.**

Automated Bug Removal for Software-Defined Networks

Yang Wu*, Ang Chen*, Andreas Haeberlen*, Wenchao Zhou†, Boon Thau Loo*

*University of Pennsylvania, †Georgetown University

Abstract

When debugging an SDN application, diagnosing the problem is merely the first step: the operator must still find a fix that solves the problem, without causing new problems elsewhere. However, most existing debuggers focus exclusively on diagnosis and offer the network operator little or no help with finding an effective fix. Finding a suitable fix is difficult because the number of candidates can be enormous.

In this paper, we propose a step towards automated repair for SDN applications. Our approach consists of two elements. The first is a data structure that 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 backtest 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 have implemented a system that maintains meta provenance for SDNs, as well as a prototype debugger that uses the meta provenance to automatically suggest repairs. Results from several case studies show that, for problems of moderate complexity, our debugger can find high-quality repairs within one minute.

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 software programs, and, like all other programs, these programs can have bugs.

There is a substantial literature on network debugging and root cause analysis [16, 21, 23, 25, 36, 55, 61]. These tools can offer network operators a lot of help with debugging. For instance, systems like NetSight [21] and negative provenance [55] provide a kind of “backtrace” to capture historical executions, 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 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 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. However, current tools offer far less help with this second step than with the first.

In this paper, we present a step towards automated bug fixing in SDN applications. Ideally, we would like to provide a “Fix it!” button that automatically finds and fixes the root cause of an observed problem. However, removing the human operator from the loop entirely seems risky, since an automated tool cannot know the operator’s intent. Therefore we opt for a slightly less ambitious goal, which is to provide the operator with a list of suggested repairs.

Our approach is to leverage and enhance 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 [9, 50]. By seeing SDN programs 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* [6]. 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 as *network provenance*, and it has been used, e.g., in debuggers and forensic tools such as ExSPAN [63], SNP [61] and Y! [55]. 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 respon-

sible 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 counterfactual reasoning to enable a form of negative provenance [55], 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.

Our approach presents 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. To address this challenge, we show how 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. 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 [27, 41]. 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 enables us to produce a list of suggested repairs that 1) are small and plausible, 2) fix the problem at hand, and 3) are unlikely to affect unrelated parts of the network.

We present a concrete algorithm that can generate meta provenance for arbitrary controller programs, as well as a prototype system that can collect the necessary data in SDNs and suggest repairs. We have applied our approach to three different controller languages, and we report results from several case studies; our results show that our system can generate high-quality repairs for realistic bugs, typically in less than one minute.

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 operator notices that web server H2 is not receiving any requests from the Internet.

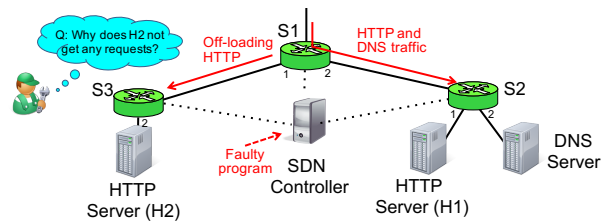


Figure 1: Example scenario. The primary web server (H1) is too busy, so the network offloads some traffic to a backup server (H2). The offloaded requests are never received because of a bug in the controller program.

Our goal is to build a debugger that accepts a simple specification of the observed problem (such as “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

Since our approach involves tracking causal dependencies, we will explain it using a declarative language, specifically *network datalog* (NDlog) [34], which makes these dependencies obvious. However, these dependencies are fundamental, and they exist in all the other languages that are used to program SDNs. To demonstrate this, we have applied our approach to three different languages, of which only one is declarative; for details, please see Section 5.8.

In NDlog, the state of a node (switch, controller, or server) is modeled as a set of *tables*, each of which contains a number of *tuples* (e.g., configuration state or network events). For instance, an SDN switch might contain a table called `FlowTable`, where each tuple represents a flow entry. Tuples can be manually inserted or programmatically derived from other tuples; the former are called *base tuples* and the latter *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 a) there is also a tuple $B(@X, Q)$ on that node, and b) $Q=2 * P$. The $@$ symbol specifies the node on which the tuple resides, and the $:-$ symbol is the derivation operator. 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 .

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 (e.g., $A(@X, 5)$), then it must be the case that all the predicates in r were

```

r1 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), WebLoadBalancer(@C,Hdr,Prt), Swi == 1.
r2 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 1, Hdr == 53, Prt := 2.
r3 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 1, Hdr != 53, Prt := -1.
r4 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 1, Hdr != 80, Prt := -1.
r5 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 80, Prt := 1.
r6 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 53, Prt := 2.
r7 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 80, Prt := 2.

```

Figure 2: Part of an SDN controller program written in NDlog: Switch S1 load-balances HTTP requests across S2 and S3 (rule r1), forwards DNS requests to S3 (rule r2); and drops all other traffic (rules r3–r4). S2 and S3 forward the traffic to the correct server based on the destination port (rules r5–r7). The bug from Section 2.3 is underlined.

true (e.g., $B(@X, 10)$), and all the constraints in r were satisfied (e.g., $10=2*5$). 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 packets at border routers). The result is as a *provenance tree*, in which each vertex represents a tuple and edges represent direct causality; the root tuple is the one that is being explained, and the base tuples are the leaves. Using negative provenance [55], we can also explain why a tuple *does not* exist, by reasoning counterfactually about how the tuple *could* have been derived.

2.3 Case study: Faulty program

We now return to the scenario in Figure 1. One possible reason for this situation is that the operator has made a copy-and-paste error when writing the program. Figure 2 shows part of the (buggy) controller program: when the operator added the second web server H2, she had to update the rules for switch S3 to forward HTTP requests to H2. Perhaps she saw that rule r5, which is used for sending HTTP requests from S2 to H1, seemed to do something similar, so she copied it to another rule r7 and changed the forwarding port, but forgot to change the condition $Swi==2$ to check for 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 the operator is still largely on her own when interpreting the provenance information and fixing the bug.

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 #2 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, predicates, and entire rules can be changed, added, or deleted. 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 we need to explain – e.g., the absence of a particular flow entry for HTTP traffic. Thus, we need not consider changes to the destination port in the header (Hdr) in r7 (because that predicate is already true) or to unrelated rules that do not generate flow entries.

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 [41], 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.

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? 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

```

program ← rule | rule program
rule ← id func ":" funcs "," sels "," assigns "."
id ← (0-9a-zA-Z)+
funcs ← func | func func
func ← table "(" location "," arg "," arg ")"
table ← (a-zA-Z)+
assigns ← assign | assign assigns
assign ← arg "!=" expr
sels ← sel "," sel
sel ← expr opr expr
opr ← == | < | > | !=
expr ← integer | arg

```

Figure 3: μ Dlog grammar

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 backtesting changes with historical information from the network.

3 Meta Provenance

In this section, we show how to derive a simple meta provenance graph for both positive and negative events. We begin with a basic provenance graph for declarative programs, and then extend it to obtain meta provenance.

For ease of exposition, we explain our approach using a toy language, which we call μ Dlog. In essence, μ Dlog is a heavily simplified variant of NDlog: all tables have exactly two columns; all rules have one or two predicates and exactly two selection predicates, all selection predicates must use one of four operators ($<$, $>$, $!=$, $==$), and there are no data types other than integers. The grammar of this simple language is shown in Figure 3. The controller program from our running example (in Figure 2) happens to already be a valid μ Dlog program.

3.1 The basic provenance graph

Recall from Section 2.2 that provenance can be represented as a DAG in which the vertices are events and the edges indicate direct causal relationships. Since NDlog’s declarative syntax directly encodes dependencies, we can define relatively simple provenance graphs for it. For convenience, we adopt a graph from our prior work [55], which contains the following *positive* vertices:

- $\text{EXIST}([t_1, t_2], N, \tau)$: Tuple τ existed on node N from time t_1 to t_2 ;
- $\text{INSERT}(t, N, \tau)$, $\text{DELETE}(t, N, \tau)$: Base tuple τ was inserted (deleted) on node N at time t ;
- $\text{DERIVE}(t, N, \tau)$, $\text{UNDERIVE}(t, N, \tau)$: Derived tuple τ acquired (lost) support on N at time t ;
- $\text{APPEAR}(t, N, \tau)$, $\text{DISAPPEAR}(t, N, \tau)$: Tuple τ appeared (disappeared) on node N at time t ; and
- $\text{SEND}(t, N \rightarrow N', \pm\tau)$, $\text{RECEIVE}(t, N \leftarrow N', \pm\tau)$: $\pm\tau$ was sent (received) by node N to/from N' at t .

Conceptually, the system builds the provenance graph incrementally at runtime: whenever a new base tuple is

inserted, the system adds an INSERT vertex, and whenever a rule is triggered and generates a new derived tuple, the system adds a DERIVE vertex. The APPEAR and EXIST vertexes are generated whenever a tuple is added to the database (after an insertion or derivation), and the interval in the EXIST vertex is updated once the tuple is deleted again. The rules for DELETE, UNDERIVE, and DISAPPEAR are analogous. The SEND and RECEIVE vertexes are used when a rule on one node has a tuple τ on another node as a precondition; in this case, the system sends a message from the latter to the former whenever τ appears ($+\tau$) or disappears ($-\tau$), and the two vertexes are generated when this message is sent and received, respectively. Notice that – at least conceptually – vertexes are never deleted; thus, the operator can inspect the provenance of past events.

The system inserts an edge (v_1, v_2) between two vertexes v_1 and v_2 whenever the event represented by v_1 is a direct cause of the event represented by v_2 . Derivations are caused by the appearance (if local) or reception (if remote) of the tuple that satisfies the last precondition of the corresponding rule, as well as by the existence of any other tuples that appear in preconditions; appearances are caused by derivations or insertions, message transmissions by appearances, and message arrivals by message transmissions. The rules for underivations and disappearances are analogous. Base tuple insertions are external events that have no cause within the system.

So far, we have described only the vertexes for positive provenance. The full graph also supports *negative* events [55] by introducing a negative “twin” for each vertex. For instance, the counterpart to APPEAR is NAPPEAR, which represents the fact that a certain tuple *failed* to appear. For a more detailed discussion of negative provenance, please see [55].

3.2 The meta provenance graph

The above provenance graph can only represent causality between data. We now extend the graph to track provenance of programs by introducing two elements: *meta tuples*, which represent the syntactic elements of the program itself (such as conditions and predicates) and *meta rules*, which describe the operational semantics of the language. For clarity, we describe the meta model for μ Dlog here; our meta model for the full NDlog language is more complex but follows the same approach.

Meta tuples: We distinguish between two kinds of meta tuples: program-based tuples and runtime-based tuples. Program-based tuples are the syntactic elements that are visible to the programmer: rule heads (HeadFunc), predicates (PredFunc), assignments (Assign), constants (Const), and operators (Oper). Runtime-based tuples describe data structures inside the NDlog runtime: base tuple insertions (Base), tuples (Tuple), sat-

```

h1 Tuple(@C,Tab,Val1,Val2) :- Base(@C,Tab,Val1,Val2).
h2 Tuple(@L,Tab,Val1,Val2) :- HeadFunc(@C,Rul,Tab,Loc,Arg1,Arg2), HeadVal(@C,Rul,JID,Loc,L), Val == True,
    HeadVal(@C,Rul,JID1,Arg1,Val1), HeadVal(@C,Rul,JID2,Arg2,Val2), Sel(@C,Rul,JID,SID,Val), Val' == True,
    Sel(@C,Rul,JID,SID',Val'), True == f_match(JID1,JID), True == f_match(JID2,JID), SID != SID'.
p1 TuplePred(@C,Rul,Tab,Arg1,Arg2,Val1,Val2) :- Tuple(@C,Tab,Val1,Val2), PredFunc(@C,Rul,Tab,Arg1,Arg2).
p2 PredFuncCount(@C,Rul,Count<N>) :- PredFunc(@C,Rul,Tab,Arg1,Arg2).
j1 Join4(@C,Rul,JID,Arg1,Arg2,Arg3,Arg4,Val1,Val2,Val3,Val4) :- TuplePred(@C,Rul,Tab,Arg1,Arg2,Val1,Val2),
    TuplePred(@C,Rul,Tab',Arg3,Arg4,Val3,Val4), PredFuncCount(@C,Rul,N), N==2, Tab != Tab', JID := f_unique().
j2 Join2(@C,Rul,JID,Arg1,Arg2,Val1,Val2) :- TuplePred(@C,Rul,Tab,Arg1,Arg2,Val1,Val2), PredFuncCount(@C,Rul,N),
    N == 1, JID := f_unique().
e1 Expr(@C,Rul,JID,ID,Val) :- Const(@C,Rul,ID,Val), JID := *.
e2 Expr(@C,Rul,JID,Arg1,Val1) :- Join2(@C,Rul,JID,Arg1,Arg2,Val1,Val2).
e3-e7 // analogous to e2 for Arg2/Val2 (Join2) and Arg1..4/Val1..4 (Join4)
a1 HeadVal(@C,Rul,JID,Arg,Val) :- Assign(@C,Rul,Arg,ID), Expr(@C,Rul,JID,ID,Val).
s1 Sel(@C,Rul,JID,SID,Val) :- Oper(@C,Rul,SID,ID',ID'',Opr), Expr(@C,Rul,JID',ID',Val'),
    Expr(@C,Rul,JID'',ID'',Val''), True == f_match(JID',JID''), JID := f_join(JID',JID''),
    Val := (Val' Opr Val''), ID' != ID''.

```

Figure 4: Meta rules for μ Dlog.

ified predicates (`TuplePred`), evaluated expressions (`Expr`), joins (`Join`), selections (`Sel`) and values in rule heads (`HeadVal`). Although concrete implementations may maintain additional data structures (e.g., for optimizations), these tuples are sufficient to describe the operational semantics.

Meta rules: Figure 4 shows the full set of meta rules for μ Dlog. Notice that these rules are written in NDlog, not in μ Dlog itself. We briefly explain each meta rule below.

Tuples can exist for two reasons: they can be inserted as base tuples (`h1`) or derived via rules (`h2`). Recall that, in μ Dlog’s simplified syntax, each rule joins at most two tables and has exactly two selection predicates to select tuples from these tables. A rule “fires” and produces a tuple $T(a, b)$ iff there is an assignment of values to a , and b that satisfies both predicates. (Notice that the two selection predicates are distinguished by a unique selection ID, or SID.) We will return to this rule again shortly.

The next four meta rules compute the actual joins. First, whenever a (syntactic) tuple appears as in a rule definition, each concrete tuple that exists at runtime generates one variable assignment for that tuple (`p1`). For instance, if a rule r contains `Foo(A, B)`, where A and B are variables, and at runtime there is a concrete tuple `Foo(5, 7)`, meta rule `p1` would generate a `TuplePred(@C, r, Foo, A, B, 5, 7)` meta tuple to indicate that 5 and 7 are assignments for A and B .

Depending on the number of tuples in the rule body (calculated in rule `p2`), meta rule `j1` or `j2` will be triggered: When it contains two tuples from different tables, meta rule `j1` computes a `Join4` tuple for each pair of tuples from these tables. Note that this is a full cross-product, from which another meta rule (`s1`) will then select the tuples that match the selection predicates in the rule. For this purpose, each tuple in the join is given a unique join ID (JID), so that the values of the selection predicates can later be matched up with the correct tuples. If a rule contains only a tuple from one table, we compute a `Join2` tuple instead (`j2`).

The next seven meta rules evaluate expressions. Expressions can appear in two different places – in a rule head and in a selection predicate – but since the evaluation logic is the same, we use a single set of meta rules for both cases. Values can come from integer constants (`e1`) or from any element of a `Join2` or `Join4` meta tuple (`e2–e7`). Notice that most of these values are specific to the join on which they were evaluated, so each `Expr` tuple contains a specific JID; the only exception are the constants, which are valid for all joins. To capture this, we use a special JID wildcard (`*`), and we test for JID equality using a special function `f_match(JID1, JID2)` that returns true iff `JID1==JID2` or if either of them is `*`.

The last two meta rules handle assignments (`a1`) and selections (`s1`). An assignment simply sets a variable in a rule head to the value of an expression. The `s1` rule determines, for each selection predicate in a rule (identified by SID) and for each join state (identified by JID) whether the check succeeds or fails. Function `f_join(JID1, JID2)` is introduced to handle JID wildcard: it returns `JID1` if `JID2` is `*`, or `JID2` otherwise. The result is recorded in a `Sel` meta tuple, which is used in `h2` to decide whether a head tuple is derived.

μ Dlog requires only 13 meta tuples and 15 meta rules; the full meta model for NDlog contains 23 meta tuples and 23 meta rules. We omit the details here; they are included in a technical report [54].

3.3 Meta provenance forests

So far, we have essentially transformed the original NDlog program into a new “meta program”. In principle, we could now generate meta provenance graphs by applying a normal provenance graph generation algorithm on the meta program – e.g., the one from [55]. However, this is not quite sufficient for our purposes. The reason is that there are cases where the same effect can be achieved in multiple ways. For instance, suppose that we are explaining the absence of an X tuple, and that there are two different rules, $r1$ and $r2$, that could derive X . If our goal

was to *explain why* X was absent, we would need to include explanations for both $r1$'s and $r2$'s failure to fire. However, our goal is instead to *make X appear*, which can be achieved by causing *either* $r1$ *or* $r2$ to fire. If we included both in the provenance tree, we would generate only repairs that cause both rules to fire, which is unnecessary and sometimes even impossible.

Our solution is to replace the meta provenance tree with a meta provenance *forest*. Whenever our algorithm encounters a situation with k possible choices that are each individually sufficient for repair, it replaces the current tree with k copies of itself and continues to explore only one choice in each tree.

3.4 From explanations to repairs

The above problem occurs in the context of disjunctions; next, we consider its “twin”, which occurs in the context of conjunctions. Sometimes, the meta provenance must explain why a rule with multiple preconditions did *not* derive a certain tuple. For diagnostic purposes, the absence of one missing precondition is already sufficient to explain the absence of the tuple. However, meta provenance is intended for repair, i.e., it must allow us to find a way to make the missing tuple appear. Thus, it is not enough to find a way to make a single precondition true, or even ways to make each precondition true individually. What we need is a way to satisfy *all* the preconditions *at the same time!*

For concreteness, consider the following simple example, which involves a meta rule $A(x, y) : \neg B(x), C(x, y), x+y>1, x>0$. Suppose that the operator would like to find out why there is no $A(x, y)$ with $y==2$. In this case, it would be sufficient to show that there is no $C(x, y)$ with $y==2$ and $x>0$; cross-predicate constraints, such as $x+y>1$, can be ignored. However, if we want to actually make a suitable $A(x, y)$ appear, we need to *jointly* consider the absence of both $B(x)$ and $C(x, y)$, and ensure that all branches of the provenance tree respect the cross-predicate constraints. In other words, we cannot explore the two branches separately; we must make sure that their contents “match”.

To accomplish this, our algorithm automatically generates a constraint pool for each tree. It encodes the attributes of tuples as variables, and it formulates constraints over these variables. For instance, given the missing tuple A_0 , we add two variables $A_0.x$ and $A_0.y$. To initialize the constraint pool, the root of the meta provenance graph must satisfy the operator's requirement: $A_0.y == 2$. While expanding any missing tuple, the algorithm adds constraints as necessary for a successful derivation. In this example, three constraints are needed: first, the predicates must join together, i.e., $B_0.x == C_0.x$. Second, the predi-

cates must satisfy the constraints, i.e., $B_0.x>0$ and $C_0.x+C_0.y>1$. Third, the predicates must derive the head, i.e., $A_0.x==C_0.x$ and $A_0.y==C_0.y$. In addition, tuples must satisfy primary key constraints. For instance, suppose deriving $B(x)$ requires $D_0(9, 1)$ while deriving $C(x, y)$ requires $D_1(9, 2)$. If x is the only primary key of table $D(x, y)$, $D_0(9, 1)$ and $D_1(9, 2)$ cannot co-exist at the same time. Therefore, the explanation is inconsistent for generating repairs. To address such cases, we encode additional constraints: $D.x == D_0.x$ implies $D.y == 1$ and $D.x == D_1.x$ implies $D.y == 2$.

3.5 Generating meta provenance

In general, meta provenance forests may consist of infinitely many trees, each with infinitely many vertices. Thus, we cannot hope to materialize the entire forest. Instead, we adopt a variant of the approach from [55] and use a step-by-step procedure that constructs the trees incrementally. We define a function $QUERY(v)$ that, when called on a vertex v from any (partial) tree in the meta provenance forest, returns the immediate children of v and/or “forks” the tree as described above. By calling this function repeatedly on the leaves of the trees, we can explore the trees incrementally. The two key differences to [55] are the procedures for expanding $NAPPEAR$ and $NDERIVE$ vertices: the former must now “fork” the tree when there are multiple children that are each individually sufficient to make the missing tuple appear (Section 3.3), and the latter must now explore a join across *all* preconditions of a missing derivation, while collecting any relevant constraints (Section 3.4).

To explore an infinite forest with finite memory, our algorithm maintains a set of partial trees. Initially, this set contains a single “tree” that consists of just one vertex – the vertex that describes the symptom that the operator has observed. Then, in each step, the algorithm picks one of the partial trees, randomly picks a vertex within that tree that does not have any children yet, and then invokes $QUERY$ on this vertex to find the children, which are then added to that tree. As discussed before, this step can cause the tree to fork, adding multiple copies to the set that differ only in the newly added children. Another possible outcome is that the chosen partial tree is completed, which yields a repair candidate.

Each tree – completed or partial – is associated with a *cost*, which intuitively represents the implausibility of the repair that the tree represents. (Lower-cost trees are more plausible.) Initially, the cost is zero. Whenever a base tuple is added that represents a program change, we increase the total cost of the corresponding tree by the cost of that change. In each step, our algorithm picks the partial tree with the lowest cost; if there are multiple trees with the same cost, our algorithm picks the one

```

function GENERATEREPAIRCANDIDATES( $P$ )
   $R \leftarrow \emptyset$ ,  $\tau_r \leftarrow \text{ROOTTUPLE}(P)$ 
  if MISSINGTUPLE( $\tau_r$ ) then
     $C \leftarrow \text{CONSTRAINTPOOL}(P)$ 
     $A \leftarrow \text{SATASSIGNMENT}(C)$ 
    for ( $\tau_i \in \text{BASETUPLES}(P)$ )
      if MISSINGTUPLE( $\tau_i$ ) then
         $R \leftarrow R \cup \text{CHANGETUPLE}(\tau_i, A)$ 
    else if EXISTINGTUPLE( $\tau_r$ ) then
      for ( $T_i \in \text{BASETUPLECOMBINATIONS}(P)$ )
         $R_{ci} \leftarrow \emptyset$ ,  $R_{di} \leftarrow \emptyset$ 
         $C_i \leftarrow \text{SYMBOLICPROPAGATE}(P, T_i)$ 
         $A_i \leftarrow \text{UNSATASSIGNMENT}(C_i)$ 
        for ( $\tau_i \in T_i$ )
           $R_{ci} \leftarrow R_{ci} \cup \text{CHANGETUPLE}(\tau_i, A_i)$ 
           $R_{di} \leftarrow R_{di} \cup \text{DELETETUPLE}(\tau_i)$ 
         $R \leftarrow R \cup R_{ci} \cup R_{di}$ 
  RETURN  $R$ 

```

Figure 5: Algorithm for extracting repair candidates from the meta provenance graph. For a description of the helper functions, please see [54].

with the smallest number of unexpanded vertexes. Repair candidates are output only once there are no trees with a lower cost. Thus, repair candidates are found in cost order, and the first one is optimal with respect to the chosen cost metric; if the algorithm runs long enough, it should eventually find a working repair. (For a more detailed discussion, please see [54].) In practice, the algorithm would be run until some reasonable cut-off cost is reached, or until the operator’s patience runs out.

The question remains how to assign costs to program changes. We assign 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). Thus, 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.

3.6 Limitations

The above approach is likely to find simple problems, such as incorrect constraints or copy-and-paste errors, but it is not likely to discover fundamental flaws in the program logic that require repairs in many different places and/or several new rules. However, software engineering studies have consistently shown that simple errors, such as copy-and-paste bugs, are very common: simple typos already account for 9.4-9.8% of all semantic bugs [32], and 70–90% of bugs can be fixed by changing only existing syntactic elements [41]. Because of this, we believe that an approach that can automatically fix “low-cost” bugs can still be useful in practice.

Our approach focuses exclusively on incorrect computations; there are classes of bugs, such as concurrency bugs or performance bugs, that it cannot repair. We speculate that such bugs can be found with a richer meta model, but this is beyond the scope of the present paper.

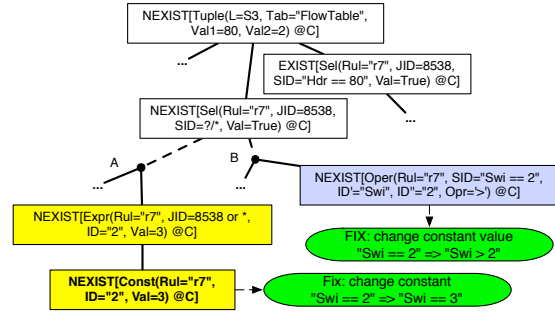


Figure 6: Meta provenance of a missing flow entry. It consists of two trees (white + yellow, white + blue), each of which can generate a repair candidate.

4 Generating repair candidates

As discussed in Section 3.5, our algorithm explores the meta provenance forest in cost order, adding vertexes one by one by invoking `QUERY` on a leaf of an existing partial tree. Thus, the algorithm slowly generates more and more trees; at the same time, some existing trees can be eventually completed because none of their leaves can be further expanded (i.e., `QUERY` returns \emptyset on them). Once a tree is completed, we invoke the algorithm in Figure 5 to extract a candidate repair.

The algorithm has two cases: one for trees that have an existing tuple at the root (e.g., a packet that reached a host it should not have reached), and one for trees that have a missing tuple at the root (e.g., a packet failed to reach its destination). We discuss each in turn. Furthermore, we note that the ensuing analysis is performed on the *meta* program, which is independent from the language that the *original* program is written in.

4.1 Handling negative symptoms

If the root of the tree is a missing tuple, its leaves will contain either missing tuples or missing meta tuples, which can be then created by inserting the corresponding tuples or program elements. However, some of these tuples may still contain variables – for instance, the tree might indicate that an `A(x)` tuple is missing, but without a concrete value for `x`. Hence, the algorithm first looks for a satisfying assignment of the tree’s constraint pool (Section 3.4). If such an assignment is found, it will supply concrete values for all remaining variables; if not, the tree cannot produce a working repair and is discarded.

As an example, Figure 6 shows part of the meta provenance of a missing event. It contains two meta provenance trees, which have some vertices in common (colored white), but do not share other vertices (colored yellow and blue). The constraint pool includes `Const0.Val = 3`, `Const0.Rul = r7`, and `Const0.ID = 2`. That is, the repair requires the exist-

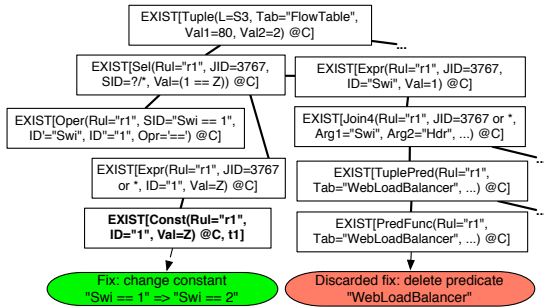


Figure 7: Meta provenance of a harmful flow entry. All repairs (e.g., green and red) can prevent this derivation, but the red one rederives the tuple via other meta rules.

tence of a constant of value 3 in rule $r7$. Therefore, we can change value of the original constant (identified by identical primary keys Rul and ID) to 3.

4.2 Handling positive symptoms

Meta provenance can also help with debugging scenarios with positive symptoms. Figure 7 shows the meta provenance graph of a tuple that exists, but should not exist. We can make this tuple disappear by deleting (or changing in the proper way) any of the base tuples or meta tuples on which the derivation is based.

However, neither base tuples nor meta tuples are always safe to change. In the case of meta tuples, we must ensure that the change does not violate the syntax of the underlying language (in this case, $\mu Dlog$). For instance, it would be safe to delete a `PredFunc` tuple to remove an entire predicate, but it may not be safe to delete a `Const` meta tuple, since this might result in an incomplete expression, such as `Swi >`.

In the case of changes to base tuples, the problem is to find changes that a) will make the current derivation disappear, and that b) will *not* cause an alternate derivation of the same tuple via different meta rules. To handle the first problem, we do not directly replace elements of a tuple with a different value. Rather, we initially replace the elements with symbolic constants and then re-execute the derivation of meta rules symbolically while collecting constraints over the symbolic constants that must hold for the derivation to happen. Finally, we can negate these constraints and use a constraint solver to find a satisfying assignment for the negation. If successful, this will yield concrete values we can substitute for the symbolic constant that will make the derivation disappear.

For concreteness, we consider the green repair in Figure 7. We initially replace `Const('r1', 1, 1)` with `Const('r1', 1, Z)` and then reexecute the derivation to collect constraints – in this case, `1==Z`. Since `Z=2` does not satisfy the constraints, we can make the tuple at

the top disappear by changing `Z` to 2 (which corresponds to changing `Swi==1` to `Swi==2` in the program).

This leaves the second problem from above: even if we make a change that disables one particular derivation of an undesired tuple, that very change could enable some other derivation that causes the undesired tuple to reappear. For instance, suppose we delete the tuple `PredFunc('r1', 'WebLoadBalancer', ...)`, which corresponds to deleting the `WebLoadBalancer` predicate from the $\mu Dlog$ rule $r1$ (shaded red in Figure 7). This deletion will cause the `Join4` tuple to disappear, and it will change the value of `PredFuncCount` from 2 to 1. As a result, the derivation through meta rule $j1$ will duly disappear; however, this will instead trigger meta rule $j2$, which leads to another derivation of the same flow entry.

Solving this for arbitrary programs is equivalent to solving the halting problem, which is NP-hard. However, we do not need a perfect solution because this case is rare, and because we can either use heuristics to track certain rederivations or we can easily eliminate the corresponding repair candidates during backtesting.

4.3 Backtesting a single repair candidate

Although the generated repairs will (usually) solve the problem immediately at hand, by making the desired tuple appear or the undesired tuple disappear, each repair can also have a broader effect on the network as a whole. For instance, if the problem is that a switch forwarded a packet to the wrong host, one possible “repair” is to disable the rule that generates flow entries for that switch. However, this would also prevent *all other* packets from being forwarded, which is probably too restrictive.

To mitigate this, we adopt the maxim of “primum non nocere” [20] and assess the global impact of a repair candidate before suggesting it. Specifically, we backtest the repair candidates in simulation, using historical information from the network. We can approximate past control-plane states from the diagnostic information we already record for the provenance; to generate a plausible workload, we can use a Netflow trace or a sample of packets. We then collect some key statistics, such as the number of packets delivered to each host. Since the problems we are aiming to repair are typically subtle (total network failures are comparatively easy to diagnose!), they should affect only a 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.

In essence, the metrics play the role of the test suite that is commonly used in the wider literature on automated program fixing. While the simple metric from above should serve as a good starting point, operators could easily add metrics of their own, e.g., to encode

<code>r7(v1) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 3, Hdr == 80, Prt := 2.</code>
<code>r7(v2) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi > 2, Hdr == 80, Prt := 2.</code>
<code>r7(v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi != 2, Hdr == 80, Prt := 2.</code>
(a)
<code>r6(v1,v2,v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 53, Prt := 2.</code>
<code>r7(v1,v2,v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 3, Hdr == 80, Prt := 2.</code>
<code>r7(v2,v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi > 3, Hdr == 80, Prt := 2.</code>
<code>r7(v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi < 2, Hdr == 80, Prt := 2.</code>
(b)

Figure 8: (a) Three repair candidates, all of which can generate forwarding flow entries for switch S2 by fixing `r7` in the original program in Figure 2; other parts of the program are unchanged. (b) Backtesting program that evaluates all three repair candidates simultaneously while running shared computations only once.

performance goals (load balancing, link utilization) or security restrictions (traffic from X should never reach Y). However, recall that, in contrast to much of the earlier work on program fixing, we do *not* rely on this “test suite” to *find* candidate repairs (we use the meta provenance for that); the metrics simply serve as a sanity check to weed out repairs with serious side effects. The fact that a given repair passed the backtesting stage is not a guarantee that no side effects will occur.

As an additional benefit, the metrics can be used to rank the repairs, and to give preference to the candidates that have the smallest impact on the overall network.

4.4 Backtesting multiple repair candidates

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 can leverage another concept from the database literature to speed up this process considerably. Recall that each backtest simulates the behavior of the network with the repaired program. 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 [19, 35].

Multi-query optimization exploits the fact that almost all computation is shared by almost all repair candidates, and thus has to be performed only once. We accomplish this by transforming the original program into a *backtesting program* as follows. First, we associate each tuple with a set of tags, we extend all relations to have a new field for storing the tags, and we update all the rules such that the tag of the head is the intersection of the tags in the body. Then, for each repair candidate, we create a new tag and add copies of all the rules the repair candidate modifies, but we restrict them to this particular tag. Finally, we add rules that evaluate the metrics from Section 4.3, separately for each tag.

The effect is that data flows through the program as usual, but, at each point where a repair candidate has modified something, the flow forks off a subflow that has the tag of that particular candidate. Thus, the later in the program the modification occurs, the fewer computations have to be duplicated for that candidate. Overall, the backtesting program correctly computes the metrics for each candidate, but runs considerably faster than computing each of the metrics round after round.

As an example, Figure 8(a) shows three repair candidates (`v1`, `v2`, and `v3`) for the buggy program in Figure 2. Each of them alters the rule `r7` in a different way: `v1` changes a constant, `v2` and `v3` change an operator. (Other rules are unchanged.)

In some cases, it is possible to determine, through static analysis, that rules with different tags produce overlapping output. For instance, in the above example, the three repairs all modify the same predicate, and some of the predicates are implied by others; thus, the output for switch 3 is the same for all three tags, and the output for switches above 3 is the same for tags `v2` and `v3`. By coalescing the corresponding rules, we can further reduce the computation cost. Finding *all* opportunities for coalescing would be difficult, but recall that this is merely an optimization: even if we find none at all, the program will still be correct, albeit somewhat slower.

5 Evaluation

In this section, we report results from our experimental evaluation, which aim to answer five high-level questions: 1) Can meta provenance generate reasonable repair candidates? 2) What is the runtime overhead of meta provenance? 3) How fast can we process diagnostic queries? 4) Does meta provenance scale well with the network size? And 5) how well does meta provenance work across different SDN frameworks?

5.1 Prototype implementation

We have built a prototype based on declarative and imperative SDN environments as well as Mininet [29]. It

generates and further backtests repair candidates, such that the operator can inspect the suggested repairs and decide whether and which to apply. Our prototype consists of around 30,000 lines of code, including the following three main components.

Controllers: We validate meta provenance using three types of SDN environments. The first is a declarative controller based on RapidNet [44]; it includes a proxy that interposes between the RapidNet engine and the Mininet network and that translates NDlog tuples into OpenFlow messages and vice versa. The other two are existing environments: the Trema framework [51] and the Pyretic language [37]. (Notice that neither of the latter two is declarative: Trema is based on Ruby, an imperative language, and Pyretic is an imperative domain-specific language that is embedded in Python.)

At runtime, the controller and the network each record relevant control-plane messages and packets to a log, which can be used to answer diagnostic queries later. The information we require from runtime is not substantially different from existing provenance systems [10, 33, 55, 63], which have shown that provenance can be captured at scale and for SDNs.

Tuple generators: For each of the above languages, we have built a meta tuple generator that automatically generates meta tuples from the controller program and from the log. The program-based meta tuples (e.g., constants, operators, edges) only need to be generated once for each program; the log-based meta tuples (e.g., messages, constraints, expressions) are generated by replaying the logged control-plane messages through automatically-instrumented controller programs.

Tree constructor: This component constructs meta provenance trees from the meta tuples upon a query. As we discussed in Section 3.4, this requires checking the consistency of repair candidates. Our constructor has an interface to the Z3 solver [11] for this purpose. However, since many of the constraint sets we generate are trivial, we have built our own “mini-solver” that can quickly solve the trivial instances on its own; the nontrivial ones are handed over to Z3. The mini-solver also serves as an optimizer for handling cross-table meta tuple joins. Using a naïve nested loop join that considers all combinations of different meta tuples would be inefficient; instead, we solve simple constraints (e.g., equivalence, ranges) first. This allows us to filter the meta tuples before joining them, and use more efficient join paradigms, such as hash joins. Our cost metric is based on a study of common bug fix patterns (Pan et al. [41]).

5.2 Experimental setup

To obtain a representative experimental environment, we set up the Stanford campus network from ATPG [58] in

Mininet [29], with 16 Operational Zone and backbone routers. Moreover, we augmented the topology with edge networks, each of which is connected to the main network by at least one core router; we also set up 1 to 15 end hosts per edge network. The core network is proactively configured using forwarding entries from the Stanford campus network; the edge networks run a mix of reactive and proactive applications. In our technical report [54], we include an experiment where the controller reactively installs core routing policies. Overall, our smallest topology across all scenarios consisted of 19 routers and 259 hosts, and our largest topology consisted of 169 routers and 549 hosts. In addition, we created realistic background traffic using two traffic traces obtained in a similar campus network setting [5]; 1 to 16 of the end hosts replayed the traces continuously during the course of our experiments. Moreover, we generated a mix of ICMP ping traffic and HTTP web traffic on the remaining hosts. Overall, 4.6–309.4 million packets were sent through the network. We ran our experiments on a Dell OptiPlex 9020 workstation, which has a 8-core 3.40 GHz Intel i7-4770 CPU with 16 GB of RAM and a 128 GB OCZ Vector SSD. The OS was Ubuntu 13.10, and the kernel version was 3.8.0.

5.3 Usability: Diagnosing SDNs

A natural first question to ask is whether meta provenance can repair real problems. To avoid distorting our results by picking our own toy problems to debug, we have chosen four diagnostic scenarios from four different networking papers that have appeared at CoNEXT [13, 58], NSDI [7], and HotSDN [4], plus one common class of bugs from an OSDI paper [31]. We focused on scenarios where the root cause of the problem was a bug in the controller program. We recreated each scenario in the lab, based on its published description. The five scenarios were:

- **Q1: Copy-and-paste error [31].** A server received no requests because the operator made a copy-and-paste error when modifying the controller program. The scenario is analogous to the one in Figure 1, but with larger topology and more realistic traffic.
- **Q2: Forwarding error [58].** A server could not receive queries from certain clients because the operator made an error when specifying the action of the forwarding rule.
- **Q3: Uncoordinated policy update [13].** A firewall controller app configured white-list rules for web servers. A load-balancing controller app updated the policy on an ingress point, without coordinating with the firewall app; this caused some traffic to shift, and then to be blocked by the firewall.

	Query description	Result
Q1	H20 is not receiving HTTP requests from H2	9/2
Q2	H17 is not receiving DNS queries from H1	12/3
Q3	H20 is not receiving HTTP requests from H1	11/3
Q4	First HTTP packet from H2 to H20 is not received	13/3
Q5	H2's MAC address is not learned by the controller	9/3

Table 1: The diagnostic queries, the number of repair candidates generated by meta provenance, and the number of remaining candidates after backtesting.

- **Q4: Forgotten packets [7].** A controller app correctly installed flow entries in response to new flows; however, it forgot to instruct the switches to forward the first incoming packet in each flow.
- **Q5: Incorrect MAC learning [4].** A MAC learning app should have matched packets based on their source IP, incoming port, and destination IP; however, the program only matched on the latter two fields. As a result, some switches never learned about the existence of certain hosts.

To get a sense of how useful meta provenance would be for repairing the problems, we ran diagnostic queries in our five scenarios as shown in Table 1, and examined the generated candidate repairs. In each of the scenarios, we bounded the cost and asked the repair generator to produce all repair candidates. Table 2 shows the repair candidates returned for Q1; the others are included in our technical report [54].

Our backtesting confirmed that each of the proposed candidates was effective, in the sense that it caused the backup web server to receive at least some HTTP traffic. This phase also weeded out the candidates that caused problems for the rest of the network. To quantify the side effects, we replayed historical packets in the original network and in each repaired network. We then computed the traffic distribution at end hosts for each of these networks. We used the Two-Sample Kolmogorov-Smirnov test with significance level 0.05 to compare the distributions before and after each repair. A repair candidate was rejected if it significantly distorted the original traffic distribution; the statistics and the decisions are shown in Table 2. For instance, repair candidate G deleted `Swi==2` and `Dpt==53` in rule `r6`. This causes the controller to generate a flow entry that forwards HTTP requests at S3; however, the modified `r6` *also* causes HTTP requests to be forwarded to the DNS server.

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 second 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 ID in the faulty predicate from `Swi==2` to `Swi==3`.

Table 1 summarizes the quality of repairs our prototype generated for all scenarios for the RapidNet con-

	Repair candidate (Accepted?)	KS-test
A	Manually installing a flow entry (✓)	0.00007
B	Changing <code>Swi==2</code> in <code>r7</code> to <code>Swi==3</code> (✓)	0.00007
C	Changing <code>Swi==2</code> in <code>r7</code> to <code>Swi!=2</code> (X)	0.00865
D	Changing <code>Swi==2</code> in <code>r7</code> to <code>Swi>=2</code> (X)	0.00826
E	Changing <code>Swi==2</code> in <code>r7</code> to <code>Swi>2</code> (X)	0.00826
F	Deleting <code>Swi==2</code> in <code>r7</code> (X)	0.00867
G	Deleting <code>Swi==2</code> and <code>Dpt==53</code> in <code>r6</code> (X)	0.05287
H	Deleting <code>Swi==2</code> and <code>Dpt==80</code> in <code>r7</code> (X)	0.00999
I	Changing <code>Swi==2</code> and <code>Act=output-1</code> in <code>r5</code> to <code>Swi==3</code> and <code>Act=output-2</code> (X)	0.05286

Table 2: Candidate repairs generated by meta provenance for Q1, which are then filtered by a KS-test.

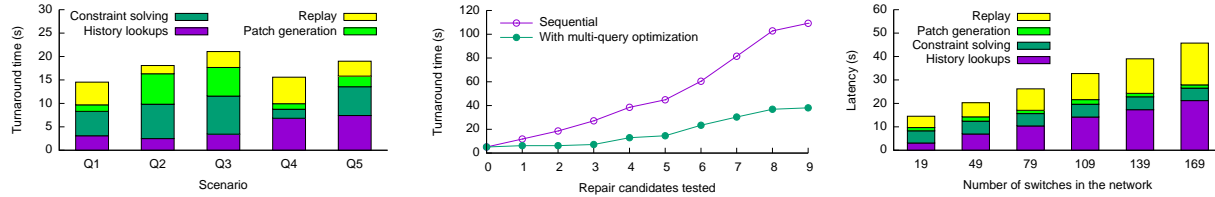
troller. Each scenario resulted in two or three repair suggestions. In the first stage, meta provenance produced between 9 and 13 repair candidates for each query, for a total of 54 repair candidates. Note that these numbers do not count expensive repair candidates that were discarded by the ranking heuristic (Section 3.5). The backtesting stage then confirmed that 48 of these candidates were effective, i.e., they fixed the problem at hand (e.g., the repair caused the server to receive at least a few packets). However, 34 of the effective candidates caused non-trivial side effects, and thus were discarded.

We note that the final set of candidates included a few non-intuitive repairs – for instance, one candidate fixed the problem in Q1 by manually installing a new flow entry. However, these repairs were nevertheless effective and had few side effects, so they should suffice as an initial fix. If desired, a human operator could always refactor the program later on.

5.4 Runtime overhead

Latency and throughput: To measure the latency and throughput overhead incurred by maintaining meta provenance, we used a standard approach of stress-testing OpenFlow controllers [14] which involves streaming incoming packets through the Trema controller using `Cbench`. Latency is defined as the time taken to process each packet within the controller. We observe that provenance maintenance resulted in a latency increase of 4.2% to 54ms, and a throughput reduction of 9.8% to 45,423 packets per second.

Disk storage: To evaluate the storage overhead, we streamed the two traffic traces obtained from [5] through our SDN scenario in Q1. For each packet in the trace, we recorded a 120-byte log entry that contains the packet header and the timestamp. The logging rates for the two traces are 20.2 MB/s and 11.4 MB/s per switch, respectively, which are only a fraction of the sequential write rate of commodity SSDs. Note that this data need not be kept forever: most diagnostic queries are about problems that currently exist or have appeared recently. Thus, it should be sufficient to store the most recent entries, perhaps an hour's worth.



(a) Time to generate the repairs for each of the scenarios in Section 5.3. (b) Time needed to *jointly* backtest the first k repair candidates from Q1. (c) Scalability of repair generation phase with network size for Q1.

Figure 9: Repair generation speed for all queries; backtesting speed and scalability result for Q1.

5.5 Time to generate repairs

Diagnostic queries does not always demand a real-time response; however, operators would presumably prefer a quick turnaround. Figure 9a shows the turnaround time for constructing the meta provenance data structure and for generating repair candidates, including a breakdown by category. In general, scenarios with more complex control-plane state (Q1, Q4, and Q5) required more time to query the time index and to look up historical data; the latter can involve loop-joining multiple meta tables, particularly for the more complicated meta rules with over ten predicates. Other scenarios (Q2 and Q3) forked larger meta-provenance forests and thus spent more time on generating repairs and on solving constraints. However, we observe that, even when run on a single machine, the entire process took less than 25 seconds in all scenarios, which does not seem unreasonable. This time could be further reduced by parallelization, since different machines could work on different parts of the meta-provenance forest in parallel.

5.6 Backtesting speed

Next, we evaluate the backtesting speed using the repair candidates listed in Table 2. For each candidate, we sampled packet traces at the network ingress from the log, and replayed them for backtesting. The top line in Figure 9b shows the time needed to backtest all the candidates sequentially; testing all nine of them took about two minutes, which already seems reasonably fast. However, the less time backtesting takes, the more repair candidates we can afford to consider. The lower line in Figure 9b shows the time needed to *jointly* backtest the first k candidates using the multi-query optimization technique from Section 4.4, which merges the candidates into a single “backtesting program”. With this, testing all nine candidates took about 40 seconds. This large speedup is expected because the repairs are small and fairly similar (since they are all intended to fix the same problem); hence, there is a substantial amount of overlap between the individual backtests, which the multi-query technique can then eliminate.

5.7 Scalability

To evaluate the scalability of meta provenance with regard to the network size, we tested the turnaround time of query Q1 on larger networks which contained up to 169 routers and 549 hosts. We obtained these networks by adding more routers and hosts to the basic Stanford campus network. Moreover, we increased the number of hosts that replay traffic traces [5] to up to 16. We generated synthetic traffic on the remaining hosts, and used higher traffic rates in larger networks to emulate more hosts. As we can see from Figure 9c, the turnaround time increased linearly with the network size, but it was within 50 seconds for all cases. As the breakdown shows, the increase mainly comes from the latency increase of the historical lookups and of the replay. This is because the additional nodes and traffic caused the size of the controller state to increase. This in turn resulted in a longer time to search through the controller state, and to replay the messages. Repair generation and constraint solving time only see minor increases. This is expected because the meta provenance forest is generated from only relevant parts of the log, the size of which is relatively stable when the affected flows are given.

5.8 Applicability to other languages

To see how well meta provenance works for languages other than NDlog, we developed meta models for Trema [51] and Pyretic [37]. This required only a moderate effort (16 person-hours). Our Trema model contains 42 meta rules and 32 meta tuples; it covers basic control flow (e.g., functional calls, conditional jumps) and data flow semantics (e.g., constants, expressions, variables, and objects) of Ruby. The Pyretic model contains 53 meta rules and 41 meta tuples; it describes a set of imperative features of Python, similar to that of Ruby. It also encodes the Pyretic NetCore syntax (from Figure 4 in [37]). Developing such a model is a one-time investment – once rules for a new language are available, they can be applied to any program in that language.

To verify that these models generate effective fixes, we recreated the scenarios in Section 5.3 for Trema and Pyretic. We could not reproduce Q4 in Pyretic because

	Q1	Q2	Q3	Q4	Q5
Trema (Ruby)	7/2	10/2	11/2	10/2	14/3
Pyretic (DSL + Python)	4/2	11/2	9/2	-	14/3

Table 3: Results for Trema and Pyretic. For each scenario from Section 5.3, we show how many repair candidates are generated, and how many passed backtesting.

the Pyretic abstraction and its runtime already prevents such problems from happening. Table 3 shows our results. Overall, the number of repairs that were generated and passed backtesting are relatively stable across the different languages. For Q1, we found fewer repair candidates for Pyretic than for RapidNet and Trema; this is because an implementation of the same logic in different languages can provide different “degrees of freedom” for possible repairs. (For instance, an equality check $Swi==2$ in RapidNet would be $match(switch = 2)$ in Pyretic; a fix that changes the operator to $>$ is possible in the former but disallowed in the latter because of the syntax of $match$.) In all cases, meta provenance produced at least one repair that passed the backtesting phase.

6 Related Work

Provenance: Provenance [6] has been applied to a wide range of systems [3, 12, 18, 38, 57]. It has been used for network diagnostics before – e.g., in [10, 55, 62, 63] – but these solutions only explain why some data was or was not computed from some given input data; they do not include the program in the provenance and thus, unlike our approach, cannot generate program fixes. We have previously sketched our approach in [53]; the present paper adds a full algorithm and an experimental evaluation.

Program slicing: Given a specification of the output, program slicing [1, 42, 52] can capture relevant parts of the program by generating a reduced program, which is obtained by eliminating statements from the original program. However, slices do not encode causality and thus cannot be directly used for generating repairs.

Network debugging: There is a rich literature on finding bugs and/or certifying their absence. Some systems, such as [15, 17, 24, 25, 26, 59], use static analysis for this purpose; others, including [46, 47, 56, 58], use dynamic testing. Also, some domain-specific languages can enable verification of specific classes of SDN programs [2, 28, 39]. In contrast, the focus of our work is not verification or finding bugs, but generating *fixes*.

Automated program repair: Tools for repairing programs have been developed in several areas. The software engineering community has used genetic programming [30], symbolic execution [40], and program synthesis [8] to fix programs; they usually rely on a test suite or a formal specification to find fixes and sometimes propose only specific kinds of fixes. In the systems community, ClearView [43] mines invariants in programs, corre-

lates violations with failures, and generates fixes at runtime; ConfDiagnoser [60] compares correct and undesired executions to find suspicious predicates in the program; and Sidiroglou et al. [48] runs attack vectors on instrumented applications and then generates fixes automatically. In databases, ConQueR [50] can refine a SQL query to make certain tuples appear in, or disappear from, the output; however, it is restricted to SPJA queries and cannot handle general controller programs. These systems primarily rely on heuristics, whereas our proposed approach uses provenance to track causality and can thus pinpoint specific root causes.

In the networking domain specifically, the closest solutions are NetGen [45] and Hojjat et al. [22], which synthesize changes to an existing network to satisfy a desired property or to remove incorrect configurations, which are specified as regular expressions or Horn clauses. While these tools can generate optimal changes, e.g., the smallest number of next-hop routing changes, they are designed for repairing the data plane, i.e., a snapshot of the network configuration at a particular time; our approach repairs control programs and considers dynamic network configuration changes triggered by network traffic.

Synthesis: One way to avoid buggy network configurations entirely is to synthesize them from a specification of the operator’s intent as, e.g., in Genesis [49]. However, it is unclear whether this approach works well for complex networks or policies, so having a way to find and fix bugs in manually written programs is still useful.

7 Conclusion

Network diagnostics is almost a routine for today’s operators. However, most debuggers can only find bugs, but not suggest a fix. In this paper, we have taken a step towards better tool support for network repair, using a novel data structure that we call meta provenance. Like classic provenance, meta provenance tracks causality; but it goes beyond data causality and treats the program as just another kind of data. Thus, it can be used to reason about program changes that prevent undesirable events or create desirable events. While meta provenance falls short of our (slightly idealistic) goal of an automatic “Fix it!” button for SDNs, we believe that it does represent a step in the right direction. As our case studies show, meta provenance can generate high-quality repairs for realistic network problems in one minute, with no help from the human operator.

Acknowledgments: We thank our shepherd Nate Foster and the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF grants CNS-1054229, CNS-1065130, CNS-1453392, CNS-1513679, and CNS-1513734, as well as DARPA/I2O contract HR0011-15-C-0098.

References

- [1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. PLDI*, 1990.
- [2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeanin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.
- [3] A. Bates, D. Tian, K. R. Butler, and T. Moyer. Trustworthy whole-system provenance for the Linux kernel. In *Proc. USENIX Security*, 2015.
- [4] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker. An assertion language for debugging SDN applications. In *Proc. HotSDN*, 2014.
- [5] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. IMC*, 2010.
- [6] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In *Proc. ICDT*, 2001.
- [7] M. Canini, D. Venzano, P. Perešini, D. Kostić, and J. Rexford. A NICE way to test openflow applications. In *Proc. NSDI*, 2012.
- [8] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proc. ICSE*, 2011.
- [9] A. Chapman and H. Jagadish. Why not? In *Proc. SIGMOD*, 2009.
- [10] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proc. SIGCOMM*, 2016.
- [11] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, 2008.
- [12] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proc. USENIX Security*, 2011.
- [13] R. Durairajan, J. Sommers, and P. Barford. Controller-agnostic SDN debugging. In *Proc. CoNEXT*, 2014.
- [14] D. Erickson. The Beacon OpenFlow controller. In *Proc. HotSDN*, 2013.
- [15] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proc. NSDI*, 2005.
- [16] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proc. SIGCOMM*, 2004.
- [17] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015.
- [18] A. Gehani and D. Tariq. SPADE: Support for provenance auditing in distributed environments. In *Proc. Middleware*, 2012.
- [19] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. In *Proc. VLDB*, 2012.
- [20] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Proc. HotOS*, 2013.
- [21] 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*, 2014.
- [22] H. Hojjat, P. Reummer, J. McClurgh, P. Cerny, and N. Foster. Optimizing Horn solvers for network repair. In *Proc. FMCAD*, 2016.
- [23] 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*, 2008.
- [24] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. NSDI*, 2013.
- [25] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI*, 2012.
- [26] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.
- [27] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. ICSE*, 2013.
- [28] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *Proc. NSDI*, 2015.
- [29] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proc. HotNets*, 2010.
- [30] 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*, 2012.
- [31] 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*, 2004.
- [32] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proc. ASID*, 2006.

- [33] D. Logothetis, S. De, and K. Yocum. Scalable lineage capture for debugging DISC analysis. Technical Report CSE2012-0990, UCSD, 2012.
- [34] 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, 2009.
- [35] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. SIGMOD*, 2002.
- [36] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Proc. SIGCOMM*, 2011.
- [37] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NSDI*, 2013.
- [38] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX ATC*, 2006.
- [39] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proc. NSDI*, 2014.
- [40] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proc. ICSE*, 2013.
- [41] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.
- [42] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *Proc. ICFP*, 2012.
- [43] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proc. SOSP*, 2009.
- [44] RapidNet project web page. <http://netdb.cis.upenn.edu/rapidnet/>.
- [45] S. Saha, S. Prabhu, and P. Madhusudan. NetGen: Synthesizing data-plane configurations for network policies. In *Proc. SOSR*, 2015.
- [46] C. Scott, A. Panda, V. Brajkovic, G. Nacula, A. Krishnamurthy, and S. Shenker. Minimizing faulty executions of distributed systems. In *Proc. NSDI*, 2016.
- [47] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *Proc. SIGCOMM*, 2014.
- [48] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *Proc. IEEE Security and Privacy*, 2005.
- [49] K. Subramanian, L. D’Antoni, and A. Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *Proc. POPL*, 2017.
- [50] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *Proc. SIGMOD*, 2010.
- [51] Trema: Full-Stack OpenFlow Framework in Ruby and C, 2019. <https://trema.github.io/trema/>.
- [52] M. Weiser. Program slicing. In *Proc. ICSE*, 1981.
- [53] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated network repair with meta provenance. In *Proc. HotNets*, 2015.
- [54] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated bug removal for software-defined networks. Technical Report MS-CIS-17-02, University of Pennsylvania, 2017.
- [55] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *Proc. SIGCOMM*, 2014.
- [56] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *Proc. USENIX ATC*, 2011.
- [57] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI*, 2012.
- [58] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT*, 2012.
- [59] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proc. NSDI*, 2014.
- [60] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *Proc. ICSE*, 2013.
- [61] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proc. SOSP*, 2011.
- [62] 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*, 2013.
- [63] 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*, 2010.

