Thinking inside the Box: Differential Fault Localization for SDN Control Plane

Xing Li*†, Yinbo Yu*‡, Kai Bu†, Yan Chen†§, Jianfeng Yang‡, Ruijie Quan†
†Institute of Cyberspace Research, Zhejiang University, Hangzhou, China
‡School of Electronic Information, Wuhan University, Wuhan, China
§Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, USA
Email: †{xing_li, kaibu}@zju.edu.cn, quanruij@gmail.com, ‡{yyb, yjf}@whu.edu.cn, §ychen@northwestern.edu

***Abstract*—The control plane of Software-Defined Networking (SDN) is the key component that oversees and manages networks. However, involving design or logic flaws in its policy enforcement and network control is inevitable, which can cause it to behave incorrectly and induce network anomalies. Unfortunately, existing approaches mainly focus on policy verification or fault troubleshooting with little fault localization capability for repairing these flaws in production environments. In this paper, we present FALCON, the first FAult Localization tool for SDN CONtrol plane. We design a novel causal inference mechanism based on *differential checking*, which symmetrically compares two system behaviors with similar processes and identifies the *causality* in related *code execution paths* with concrete contexts to explain *why* a fault happened in the SDN network. Our main contributions include 1) a lightweight *rule-based dynamic tracing* mechanism for recording system behaviors of the SDN control plane, 2) a *context-aware modeling* mechanism for modeling these behaviors, and 3) a *differential checking* mechanism for localizing controller faults according to formulated symptoms. Our evaluation shows that FALCON is capable of localizing faults in SDN control plane with low overhead on performance.**

I. INTRODUCTION

Separated from the data plane, the SDN control plane (CP) is logically centralized for networking. It leverages southbound protocols (e.g., OpenFlow) to govern traffic in the data plane, and exposes various northbound interfaces (NBI) for external applications to access networks. In current SDN solutions [1]–[3], the CP performs as a network operating system with various core and application modules[1], where network management policies are transformed as modules' code logics and mutual dependencies. The modular nature of SDN CP is the significant feature that guarantees the flexibility of network service deployment.

Unfortunately, SDN CP is error-prone as a software system meeting with complicated network dynamics [5]–[11]. The controller is typically reactive and event-driven that it detects input events (e.g., OpenFlow messages and NBI requests), processes them and takes actions following specific code logics. Thus, the root causes behind faults in SDN (e.g., forwarding loop and incorrect NBI responses) are typically flaws in these logics [7], [9]. However, it is difficult to find out them, since the logics behind may be *non-deterministic*[2] (i.e., context-dependent), *cross-module* and mixed with SDN network's *asynchrony* and *concurrency* [7], [8].

To localize root causes of faults in SDN CP, unfortunately, existing solutions have some limitations: 1) Some research efforts [5], [12], [13] use formal methods to verify the correctness of network policies or abstract program models. However, they rely on manual or static analysis to model policies or programs, which is time-consuming, error-prone and cannot handle dynamic changes of network and software in production environments; 2) Blackbox testing is another approach to identifying the input event sets which can cause the controller to fail [7], [8]. However, given the set, operators still need to manually localize the root cause inside the CP. Hence, how to diagnose faults in SDN CP is still an open issue. SDN networks involve data, control and application planes, which makes previous network or software diagnosis mechanisms inapplicable. We need to track behaviors of all these planes and touch the inner side of the SDN CP to point out which part and why the part goes wrong.

In this paper, we design FALCON, the first fault localization system, which can identify the detailed root causes of faults in SDN CP to help operators quickly repair them. FALCON utilizes a *rule-based dynamic tracing* mechanism to precisely trace the system behaviors of SDN at runtime, including interactions between adjacent SDN planes and program executions inside the CP. It further models these behaviors through a deterministic *context-aware* model mechanism and mines dependencies among models as the collaborative behaviors in SDN networks. The normal behavior models are regarded as diagnosis *references*. When faced with a failure, FALCON identifies the faulty models and corresponding references and then performs a *differential checking* mechanism to point out their differences. Finally, by finding the causality with static analysis, FALCON answers not only *how* the fault occurred with a minimal set of input events, but also *why* it can occur with a minimal set of state differences in relevant code execution paths. We build a prototype for OpenDaylight (ODL) [2] controllers to evaluate FALCON with several types of faults. The result attests its capability to reveal root causes. Moreover, it introduces a low overhead for controller's network management, even optimizes partial events' processing thanks to bytecode instrumentation, e.g., achieves up to 36.3% throughput improvement in processing RESTful requests.

The rest of the article provides the background and related work (§II), describes the overview of FALCON (§III) and the

*These authors contributed equally to this work.
[1]It is also called *application agent* [1], *plugin bundle* [4] or *control program* [5]. We use them interchangeably.
[2]With the same input, the controller may behave differently in each execution.

978-3-903176-15-7

details of its design and implementation (§IV, §V, §VI, and §VII), presents evaluation results (§VIII) and concludes (§IX).

II. Background and Related Work

A. Faults in SDN control plane

To present a deep understanding of SDN faults, we survey controller-related faults (also called bugs) found in literature [5]–[10] and report the first analysis of faults in a real SDN controller bug repository, ODL Bugzilla [14], in which we analyze all 298 confirmed bugs of ODL kernel projects [4] until October 16, 2017. For clarity, we classify these faults into three categories according to their root causes as follows:

Logic/design flaw (66%): To manage networks, various network policies are implemented in SDN control software with specific *code logics*. However, due to insufficient domain knowledge or misplaced assumptions, these logics may not always be designed correctly and even conflict with each other [5], [7], [15]. For example, input events in some specific order may hit some corner cases (which are not considered in current SDN CP design), and be processed incorrectly or discarded directly, even trigger harmful race conditions [10]. We name this type of faults *logic/design flaw*.

Coding mistake (12%): Careless programming in code logic implementation can cause a variety of software or network errors, e.g., data race, null pointer, and incorrect rule distribution. Although many coding mistakes can be found and handled promptly in coding or testing stage, it is not possible to exhaust all of them, e.g., incorrect usage of service identifier [16]. In addition, there may exist a lot of unreasonable memory allocations in the controller that can cause it to crash [6].

Performance anomaly (22%): SDN controllers often suffer from centralized bottleneck problems in practical applications [6]. Apart from this, their inherent asynchrony and concurrency also exacerbate these issues [6], resulting in various failures, e.g., partial failure in batch operation, message timeout or omission, data race, and even system crash.

According to our measurement, *Logic/design flaw* is the most popular category (66%) in all analyzed faults, and it usually has a higher need for diagnosis [14]. Furthermore, we observe that almost all logic/design flaw bugs and some bugs in the other two categories can raise abnormal code execution traces in control software [5], [7], which are deviated from execution traces when the faults are not triggered.

B. Related Works

Given that the SDN controller is a software entity, many approaches based on general software diagnostic techniques have been proposed, including *controller troubleshooting* (CT) [7], [8], [10] and *program analysis* (PA) [5], [12], [13]. Since the CT-based approaches leverage blackbox testing to find defects in the controller, they may fail to provide detailed root causes of occurred faults; PA-based approaches rely on formal models of controller software or network policies, which are often error-prone and inaccurate. Different from them, FALCON dynamically traces the SDN system behaviors at runtime and further parses them with static code analysis, thus it can provide detailed root cause analysis of faults in dynamic SDN networks.

III. Overview of FALCON

In the SDN field, the fault diagnosis problem is more complex than the one for common software, since SDN CP needs to simultaneously process dynamic network events from the data plane and collaborative northbound requests from the application plane. We need to identify not only relevant internal executions in SDN controllers, but also interactions among these planes, and associate these behaviors with the occurred failure as an understandable diagnosis result. To address such problem, we leverage two major properties of SDN CP faults to design our fault diagnosis mechanism:

(1) Incorrect internal executions. As described in §II-A, most faults in SDN CP are caused by logic or design flaws, which violate correct program logics and cause deviations from correct program executions. Specifically, the controller follows context-dependent code logics to process input events, due to defective design and implementation of these logics, changes in contexts may trigger an unexpected run that cannot be handled properly [7], [9], [10].

(2) Disordered input events. The interactions between SDN control and other planes often follow some fixed orders, which are defined in southbound protocols or code logics in controllers and applications for collaborative services. The disorder of input events can induce a different set of internal invocations inside SDN CP which may trigger failures. For example, to build an OpenFlow connection, a series of messages[3] are generated in order between the switch and controller, the disorder of these messages will trigger a failure [7].

Based on these properties, we design FALCON, a *differential fault localization* system, to localize the root cause of a failure in the SDN network which suggests the presence of a fault in the CP. To diagnose faults in the SDN environment, firstly, we design a rule-based dynamic tracing mechanism to record running contexts. We then propose a context-aware modeling mechanism to cluster trace data and model their causal relationships with concrete contexts as a context-aware behavior model, with which we aim to provide deterministic models for system behaviors in SDN CP. Given a failure, we formulate symptoms occurred in different planes as the diagnosis input and design a differential checking mechanism to localize the root causes. By comparing faulty and correct system behavior models, we aim to identify the minimal but sufficient system behaviors with concrete contexts and succinct code execution paths as the diagnosis report.

Fig. 1 depicts the architecture of FALCON, which contains two parts: *a production environment* and *a simulation environment*. In the production environment (Fig. 1(a)), we deploy a *trace agent* inside a controller and an *online monitor* outside of the controller. The trace agent traces activities in the controller at runtime. The online monitor collects trace

[3]Only if two `OFPT_HELLO` messages for protocol version negotiation are successfully processed, can the standard OpenFlow messages be exchanged, e.g., `OFPT_FEATURES_REQUEST`, `OFPT_PACKET_IN` [17].

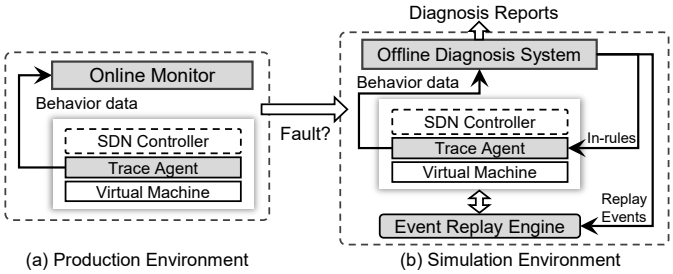

Fig. 1: An overview architecture of FALCON.

data from the trace agent, models them as system behavior models and stores them as *references* when there is no fault. When a failure occurs, we transmit the recent behavior models to the simulation environment (Fig. 1(b)) for fault diagnosis, in which a new controller is instanced with the same configuration and internal states of the production one through controller *restore* mechanisms [18], [19]. We leverage an *event replay engine* to simulate the data/application planes and reproduce practical failures by replaying collected input events. The *offline diagnosis system* performs the differential fault localization to identify the causality of a fault and output it as the *diagnosis report*. The usage of the two environments can guarantee both the authenticity of the diagnosis data and the accuracy of the diagnosis results.

IV. Dynamic System Behavior Tracing

We now present a dynamic system behavior tracing mechanism based on bytecode instrumentation for recording SDN system behaviors. Performing instrumentation on bytecode requires neither modification of controller's source code nor restart of the controller. We design a rule-based instrumentation mechanism to ease the configuration of dynamic tracing and control it to a relatively coarse granularity, *module-level*, to reduce the overhead.

A. Dynamic Tracing

Dynamic tracing can provide sensitive execution information of the SDN CP at runtime. However, it is often costly. As a software entity, given some key dynamic execution points and corresponding contexts, we can recover the entire execution path through static analysis. Thus, to find a feasible granularity of dynamic tracing, we leverage the following observations of mainstream SDN controllers [2], [3]: 1) Their modular nature indicates that their most of event processing tasks are handled under the collaboration among multiple modules; 2) Invocations among modules depend on pre-defined module interfaces, e.g., RPC and Notification; 3) Faults in controllers may originate inside a module and propagate to other modules through invocations or database operations. Hence, a module-level dynamic tracing that traces invocations through these module interfaces is enough for providing dynamic contexts, which can highly reduce the amount of inserted codes, thereby greatly decreasing the overhead than a method-level tracing.

B. Rule-based Instrumentation

To trace controllers, it is, however, challenging for operators to determine where and what code can be instrumented,

```
  public class PacketDispatcher {
    public void dispatchPacket(payload, ingress, srcMac, destMac) {
0:    NodeConnectorRef src = inventoryReader.getControllerSwitchConnectors(srcMac);
1:    NodeConnectorRef dest = inventoryReader.getNodeConnector(destMac);
2:    if (src != null)
3:      if (dest != null) sendPacketOut(payload, src, dest);
4:      else floodPacket(nodeId, payload, ingress, src);
    } }
```

Match				Action
212	PacketDispatcher	dispatchPacket	3+sendPacketOut+B	Thread, event, payload, src, dest, etc.

Fig. 2: An in-rule example for `Packet_Out`.

especially for these unfamiliar with bytecodes. To address this problem, we design a rule-based instrumentation mechanism to help operators ease this process, in which the expectant tracing feedback is specified in instrumentation rules (abbreviated as *in-rules*). An in-rule is a `<match, action>` tuple (see Fig. 2). The `match` field is used to match against bytecodes and specify where to insert codes, which consists of three name (`module`, `class` and `method`) and one location (`call site`) sub-fields. The three name sub-fields follow the code hierarchy to focus the in-rule on the `method`'s code snippet. The `call site` further localizes the instrumentation with a line number and a location (`B`(before) or `A`(after)) of a bytecode instruction in the snippet. The `action` field defines execution contexts that need to be profiled, e.g., thread, timestamp, invocation type, and variable values.

Covering expected traces by manually specifying in-rules is unfeasible. FALCON only requires operators to specify the in-rules for capturing input/output messages (e.g., RESTful requests and OpenFlow messages) and list the concerned module interfaces. Then, FALCON transforms the corresponding bytecodes into control flow graphs (CFGs) to search invocations of these interfaces and automatically generate in-rules for tracing them. Finally, with these inputs and generated in-rules, FALCON translates them into bytecodes to instrument the SDN controller. At runtime, these execution contexts defined in in-rules will be profiled and output as trace messages.

V. System Behavior Modeling

Given trace messages, we construct system behavior models of the SDN system at runtime: first, we process heavily interleaved trace messages and identify relevant internal invocation nodes and their causal relationships to construct a *context-aware* model; we further perform backtrace on these models with static analysis to mine the dependencies among them, which can make our models be accurate to capture collaborative properties in the SDN system.

A. Context-aware System Behavior Modeling

To process input events (i.e., controller tasks), the controller maintains multiple event handlers, each of which uses multiple threads to execute different *operations*. Each operation is executed in a thread with several synchronous invocations and may also involve other operations through asynchronous invocations (see Fig. 3). A task may produce multiple heterogeneous trace graphs due to *non-determinism*, each having a specific context for different conditional branches (e.g., *if*$\cdots$*else* in Fig. 2). We model such behaviors as follows:

Trace graph: Once getting a trace message, FALCON transforms it into a graph node according to its event type and clus-

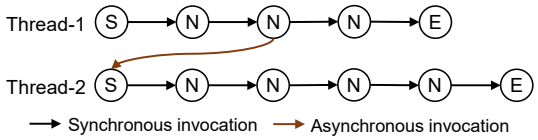

Fig. 3: A trace graph for a controller task.

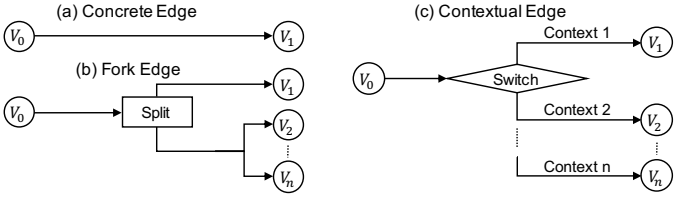

Fig. 4: Three types of graph edges.

ters it into a growing *chain graph* of an operation according to its thread ID. A chain graph is a set of synchronous nodes linked with their *happen-before* (HB) relationships. Since there is no identifier propagated through asynchronous invocations, we design a *multi-identifier* correlation mechanism for combining these chain graphs for the same task. Specifically, we construct a tuple containing multiple identifiers (including the caller's thread ID, location in its chain graph, timestamp, variable-value hashcode, and a parent-child graph path) to define existing asynchronous callers and match new coming asynchronous callee nodes. Finally, all chain graphs of a task are combined into a trace graph.

Context-aware model: CAM contains three kinds of edges (see Fig. 4): 1) A *concrete* edge has a pair of nodes representing their HB relationship; 2) A *fork* edge has multiple succeeding nodes (one is a concrete successor and others are asynchronous callees); 3) A *contextual* edge's succeeding node varies in specific contexts (i.e., condition values), which models data-dependent code logic. We say that in the code logic of a concrete edge, if there is a conditional branch varying the edge's succeeding invocations according to contexts, the edge is then transformed into a contextual edge with an additional context field to record the branch. Taking Fig. 5 for example, we combine heterogeneous trace graphs of a task into a CAM. Given two graphs (a) and (b) with their different contexts, we identify the conditional branch that leads to the two different edges (e_1) from the CFG of corresponding bytecodes. Then, we combine the two edges into a contextual edge (ce_1) with the branch that decides the succeeding nodes according to their contexts. Fig. 5(c) depicts the final CAM, in which from V_b has two possible succeeding nodes: V_c and V_d.

B. Augmentation with Model Dependency

Since SDN controllers are event-driven, their contexts are introduced by external events from data and application planes, e.g., OpenFlow messages and NBI requests. Thus, contexts in conditional branches come from their input event or previous tasks. Taking Fig. 2 for example, the action (send `PacketOut` or flood packet) of processing a flow rule request depends on the existing of the destination host in the controller's database. Mining such dependencies among task models can further address non-determinism and provide references in another dimension for diagnosis.

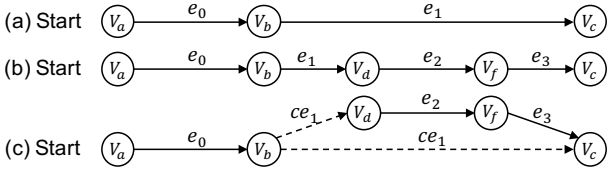

Fig. 5: An example of a trace graph combination.

Some faults are context-dependent. Thus, we start from these conditions' contexts in contextual transitions to identify model dependencies. A context can be introduced by a single input event or a set of input events in a specific order. Hence, given a context c, we iteratively backtrack current and previous models to search operations that insert/update its value and identify the corresponding input event or input event sequence. If c is introduced by a previous input event I^s with input value set s, we say that the current task model contextually depends on I^s. With such dependencies, we further augment CAMs.

VI. Differential Fault Localization

In this section, we discuss how FALCON diagnoses failures according to their symptoms with mined models. The fault localization consists of 3 phases: 1) parsing the failure symptom to locate the faulty models and their normal references; 2) symmetrically comparing them to find the differences; 3) performing static analysis from their differences to identify the related conditional branches and contexts.

From the controller's perspective, the failure symptoms may be *explicit* that we directly find anomalies from the controller, e.g., error log messages or code exceptions; or *implicit* that failures occur in other planes with no error reported in the CP, e.g., network problems or unexpected NBI responses. To perform diagnosis with these symptoms, we formulate them with the following syntax:

```
'time' : ('timestamp' | null)
'type' : ('REST' | 'log' | 'flow' | 'rule' )
'request': ('method' & 'url' & 'payload'
            & 'response content'
            & 'response status')
'log': ('status' & 'content')
'flow': ('messageType' & 'switchID'
         & 'OFVersion' & 'content')
'rule': ('switchID' & 'ruleID'
          & 'match' & 'action')
```

Faulty model locating: Given a symptom, FALCON then searches the faulty models and their references. An explicit symptom typically has a recorded timestamp, so FALCON can identify faulty models based on it and other characteristics in the symptom. For implicit symptoms without precise timestamps, FALCON starts from the latest model to search related faulty models which are different from their references.

Differential checking: In this phase, FALCON systematically compares the faulty and the reference models from their root nodes to locate their differences. Taking Fig. 6 for example, there are two heterogeneous models (*Run 1* and *2*) triggered by the same NBI request I with different contexts (S_1 and S_2), and *Run 2* leads to a failure. It is evident that the controller programs after V_b cannot run properly under the contexts S_2 of *Run 2*. Thus, to reason about the failure, we need to report

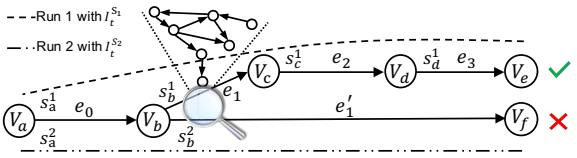

Fig. 6: An example of differential checking.

not only the differences and faulty execution path but also the key contexts causing this deviation.

Static analysis: With the different node V_b, we conduct static analysis on its bytecode to find out the related branch condition and the contexts in it. A failure is usually highly correlated with the current system state (i.e., contexts) introduced by a sequence of events; the previous events affect subsequent ones by modifying the contexts. So we also search input events that have modified these contexts. Although several contexts are involved, not all of them trigger the failure. Thus, FALCON leverages *delta debugging* with the event replay engine to eliminate unrelated contexts. In each replay, we change partial contexts and replay the changed input event sequence to check if the failure can still be reproduced. Finally, a minimal input event sequence and the triggering contexts are output with corresponding execution paths.

VII. IMPLEMENTATION

FALCON is implemented in Java with more than 10,000 lines of code, including *trace agent*, *online monitor*, *offline fault diagnosis*, excluding *event replay engine*.

Trace Agent: It is implemented based on several mature tools. First, we translate *in-rules* into codes that can be executed by a Java bytecode manipulation tool, ASM [20], which allows us to dynamically instrument SDN controllers and provides control/data flow analysis on bytecode. The agent is dynamically attached to a controller at its run-time. We then use an inter-thread messaging library (LMAX Disruptor [21]) to deliver trace data from multiple running threads to the agent thread which performs data transmission. Trace data and other program data (e.g., code file location) are sent to the outside *Online Monitor* through an off-heap inter-process messaging library (Chronicle Queue [22]).

Event Replay Engine: To reproduce failures, we implement an event replay engine which simulates both the data and application planes and sends channel messages to the controller. This engine is built on STS simulator [7] and we extend to support the generation of various northbound requests and southbound network events.

VIII. EVALUATION

To evaluate FALCON, we conduct some case studies to assess its fault localization capability (§VIII-A), and design several tests to measure its performance impact on SDN controller (§VIII-B). All the evaluations are performed on a Linux server running 64bit Ubuntu 14.04 with an Intel Xeon E2660 v2 2.2Ghz CPU (16 cores) and 64GB RAM.

In order to instrument the controller, we write in-rules for capturing *RSETful request/response* and *OpenFlow (OF) messages*, and tell FALCON to generate in-rules for the invocation interfaces we concern about, including ODL core interfaces (i.e., *Restconf operations*, *RPC*, *Notification listen* and *Data change listen*) and some other invocation interfaces (e.g., *Notification publish*). With all these in-rules, the controller is instrumented and ready to deliver trace messages outside.

A. Case Studies

To evaluate the effectiveness of our localization methodology, we selected 8 real-world faults from ODL Bugzilla [14], reproduced them and use FALCON to diagnose them. The overall diagnosis results are summarized in Table I.

The last column of Table I describes whether FALCON can diagnose the root cause of the fault. We can see that for faults from different projects with different symptoms, Falcon always plays a positive role in revealing the root causes. For bugs from 5033 to 8157, FALCON can successfully point out the faulty code logics and key contexts because these faults have sufficient reference models. As for fault without corresponding reference model, like Bug-3345, our differential localization mechanism cannot directly indicate the root cause, but FALCON can provide corresponding CAMs to free the operator from heavy log analysis task and help him understand the fault more easily with the internal view. We don't conduct case studies for faults of *performance anomaly*, but our model contains the time intervals between adjacent nodes, which can be used as the basis for diagnosing such faults.

Take Bug-5816 [23] in Table I as an example. The L2switch project's *host-expiry* feature lets ODL remove hosts that have not been observed for a long time from the network topology view. This fault says that in L2switch's reactive mode, hosts expired by this feature cannot be discovered again even if *ping* works and new flows get installed on switches.

After receiving the symptom, FALCON searches for models of OF messages related to the target host, and then identifies a deformed host-discovery model and corresponding host-purge model. Next, FALCON points out the problematic node in Addresstracker module by conducting differential checking on the deformed model and the reference model of host discovery. Following static analysis shows that the culprit is the improper *TimestampUpdateInterval* which makes Addresstracker not update the address in time, and the Hosttracker cannot learn the host consequently. Finally, FALCON confirms the root cause with replay engine and reports the diagnosis result.

B. Performance Measurement

Deploying FALCON into a controller may decrease the controller's performance in processing input events. We evaluated this performance impact by measuring the controller's throughput for processing OF messages and NBI requests under different workloads without and with FALCON (F-ODL), respectively. Since FALCON's instrumentation is built on ASM which can reduce bytecode size to optimize code execution efficiency [20], we also directly used ASM to instrument ODL (A-ODL) by reusing FALCON's in-rules with a simple *value add* instruction `action`. We performed each test 30 times and show the average results in Fig. 7.

TABLE I: Fault localization cases.

Bug ID	Description	Symptom	Project(version)	Root cause	Category	Diagnose
5033	AAA falsely authorizes user to restricted endpoint	unexpected response	aaa (B)	race condition	logic flaw	Yes
5816	Expired hosts never comeback after timing out	unexpected response	l2switch (Be)	constant misconfiguration	logic flaw	Yes
8157	Recreating a user fails after deleting it	error in log message	aaa (C)	defective user deletion	logic flaw	Yes
3345	Ping will fail in ring topology when a link down	unreachability	l2switch (Li)	incomplete topology update	design flaw	Indirectly
6053	NPE on port creation	NPE in log message	neutron (B)	incomplete JSON parsing	design flaw	Yes
7933	NPE when posting using XML	NPE in log message	netconf (C)	incomplete YANG support	design flaw	Yes
8939	Adding topology-netconf node via restconf fails	error in log message	netconf (N)	interface migration	coding mistake	Indirectly
8988	NPE when adding routes to app-peer	NPE in log message	netconf (N)	method misuse	coding mistake	Indirectly

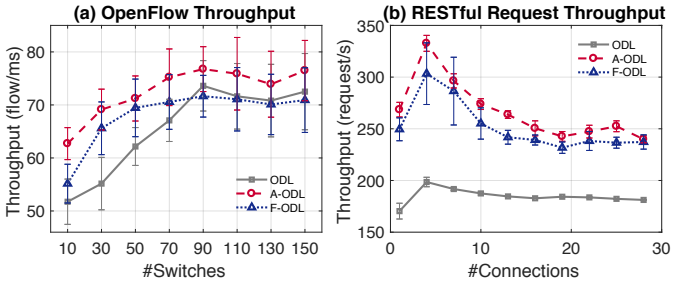

Fig. 7: Performance of ODL, ASM-ODL and FALCON-ODL in processing input events.

OpenFlow messages: We measured the controller performance in processing OF messages by running CBench [24], a benchmarking tool for testing OF controllers in the throughput mode. We started ODL with *L2switch*, *OpenFlowplugin* and *OpenFlowJava* plugins. FALCON generated 98 *in-rules* to instrument their 19 functional modules. We tested the throughput by running CBench with the number of simulated switches ranging from 10 to 150, each of which was assigned with 200 unique MACs (i.e., simulated hosts). As shown in Fig. 7 (a), A-ODL presented the best throughput than ODL (11.65%) and F-ODL (6.93%) on average due to bytecode optimization. For the same reason, when the number of switches is less than about 85, F-ODL also achieved a better throughput than ODL. Then, as the number of switches growing, ODL started to achieve a better but not noticeable throughput than F-ODL. In F-ODL, more threads need to be allocated for the delivery and processing of trace messages, which leads to slight throughput degradation. We believe this degradation is acceptable to most networks for two reasons: 1) most networks often distribute switches to multiple controllers for network reliability and scalability [25] and 2) as measured in [26], the time to generate a new rule after the controller receives a request could be more than 10ms, which is far greater than the introduced delays.

RESTful requests: ODL adopts RESTful APIs as its NBI. We tested the performance impact on processing RESTful requests with ODL Neutron plugin which provides 30 kinds of RESTful APIs (e.g., networking and QoS) with 185 kinds of requests (GET, POST, PUT, DELETE). These RESTful requests were generated and sent to ODL Neutron by the event replay engine. FALCON generated 20 *in-rules* to trace Neutron plugin (containing 4 modules). We built multiple concurrent connections (ranging from 1 to 28) between the engine and ODL to send requests (each connection sent 1850 requests), and counted the number of responses that can be received per second. Fig. 7(b) depicts the experimental results. Different from OF messages, F-ODL always has better throughput than ODL (36.3% on average) in processing RESTful requests regardless of the number of connections. The main reasons for different results between processing OF messages and RESTful requests come from two aspects: 1) RESTful requests have far lower arrival rate than OF messages and therefore ODL has lower CPU workload; 2) FALCON needs fewer in-rules to cover invocations in Neutron plugin than OF related plugins, which introduces less computing overhead.

C. Discussion

The evaluation suggests that FALCON is capable of localizing faults in SDN CP with low performance impact. Here we further discuss its several limitations and characteristics:

Intrusive Profiling: FALCON is intrusive that may lead to performance and security issues. For example, incorrect in-rules may introduce new bugs to controllers. Thus, in-rules' correctness verification shall be addressed in our future work.

Model Completeness: FALCON relies on in-rules to trace system behaviors. However, closed-source third-party middlewares or incomplete in-rules may lead to disrupting of modeling. This can be partially alleviated by applying multi-modal similarity check on middlewares' input and output [27].

Reference Sufficiency: The sufficiency of reference models is the key factor affecting FALCON's diagnosis effect. Since FALCON is deployed on controllers running in production environment, it can usually get enough normal models to enrich its reference library unless the correct model does not exist at all, which is not a common situation.

Method Generality: In this paper, FALCON is only evaluated in Java-based controllers. Nevertheless, it is easy to be deployed on other language-based controllers by adopting different underlying instrumentation tools (e.g., equip [28])) and modify the in-rule translation.

IX. CONCLUSION

SDN is an important technique for future networks. In this paper, we have presented FALCON, a system for localizing root causes of faults occurred in the SDN control plane. We design a rule-based tracing mechanism to exploit the internal system behaviors, model them with a context-aware model and realize a differential fault localization mechanism on system behavior models to localize the root causes of faults. We have also built a prototype of FALCON for ODL controllers, and our evaluation shows that it is practical for real controller runs.

ACKNOWLEDGMENT

This work is supported by National Key R&D Program of China (2017YFB0801703) and the Key Research and Development Program of Zhejiang Province (2018C01088).

REFERENCES

[1] ONF, "SDN architecture," Open Networking Foundation (ONF), Tech. Rep., 2014, https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf.

[2] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *IEEE 15th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2014, pp. 1–6.

[3] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "ONOS: towards an open, distributed SDN OS," in *Proceedings of the 3rd ACM Workshop on Hot topics in Software Defined Networking (HotSDN)*, 2014, pp. 1–6.

[4] OpenDaylight, "Project list," accessed on 2018-1-25. [Online]. Available: https://wiki.opendaylight.org/view/Project_list

[5] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[6] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A robust, secure, and high-performance network operating system," in *Proceedings of the ACM SIGSAC conference on computer and communications security (CCS)*, 2014, pp. 78–89.

[7] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock *et al.*, "Troubleshooting blackbox SDN control software with minimal causal sequences," in *Proceedings of the ACM conference on SIGCOMM*, 2014, pp. 395–406.

[8] K. Mahajan, R. Poddar, M. Dhawan, and V. Mann, "JURY: Validating Controller Actions in Software-Defined Networks," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 109–120.

[9] B. Chandrasekaran, B. Tschaen, and T. Benson, "Isolating and Tolerating SDN Application Failures with LegoSDN," in *Proceedings of the ACM Symposium on SDN Research (SOSR)*, 2016, p. 7.

[10] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the Brain: Races in the SDN Control Plane," in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 451–468.

[11] Y. Yu, X. Li, X. Leng, L. Song, K. Bu, Y. Chen, J. Yang, L. Zhang, K. Cheng, and X. Xiao, "Fault management in software-defined networking: A survey," *IEEE Communications Surveys & Tutorials*, 2018.

[12] T. Nelson *et al.*, "Static Differential Program Analysis for Software-Defined Networks," in *International Symposium on Formal Methods*, 2015, pp. 395–413.

[13] 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 *Proceedings of the ACM conference on SIGCOMM*, 2016, pp. 115–128.

[14] OpenDaylight, "OpenDaylight Bugzilla," accessed on 2018-1-25. [Online]. Available: https://bugs.opendaylight.org/

[15] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "Vericon: Towards Verifying Controller Programs in Software-Defined Networks," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 282–293.

[16] OpenDaylight, "Bug 8533 - Not possible to invoke RPC on mount points with new Restconf," 2017, accessed on 2018-1-25. [Online]. Available: https://bugs.opendaylight.org/show_bug.cgi?id=8533

[17] ONF, "OpenFlow Switch Specification Version 1.5.1," https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, 2015.

[18] OpenDaylight, "Persistence and Backup," accessed on 2018-2-25. [Online]. Available: http://docs.opendaylight.org/en/latest/getting-started-guide/persistence_and_backup.html

[19] ONOS, "Backup/Restore Tutorial," accessed on 2018-2-07. [Online]. Available: https://wiki.onosproject.org/pages/viewpage.action?pageId=18908160

[20] OW2 consortium, "ASM: A Java bytecode engineering library," accessed on 2018-1-25. [Online]. Available: http://asm.ow2.io/

[21] LMAX, "LMAX Disruptor: High Performance Inter-Thread Messaging Library," accessed on 2018-1-25. [Online]. Available: http://chronicle.software/products/chronicle-queue/

[22] OpenHFT, "Chronicle Queue: Stores a complete history of all your messages to disk," accessed on 2018-1-25. [Online]. Available: https://lmax-exchange.github.io/disruptor/

[23] OpenDaylight, "Bug 5816 - l2switch - Expired hosts never comeback after timing out," accessed on 2018-1-25. [Online]. Available: https://bugs.opendaylight.org/show_bug.cgi?id=5816

[24] R. Sherwood and Y. Kok-Kiong, "Cbench: an OpenFlow controller benchmarker," accessed on 2018-2-25. [Online]. Available: https://github.com/mininet/oflops/tree/master/cbench

[25] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proceedings of the ACM 1st workshop on Hot topics in software defined networks (HotSDN)*, 2012, pp. 7–12.

[26] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, and C. Hu, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 179–188.

[27] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "Anomaly Detection Using Program Control Flow Graph Mining From Execution Logs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 215–224.

[28] G. Romain, "equip: Python Bytecode Instrumentation," accessed on 2018-1-25. [Online]. Available: https://github.com/neuroo/equip