FALCON: Differential Fault Localization for SDN Control Plane*

Yinbo Yu[a], Xing Li[b], Kai Bu[b], Yan Chen[b,c], Jianfeng Yang[a,*]

[a]School of Electronic Information, Wuhan University, Wuhan, 430072, China
[b]Institute of Cyberspace Research, Zhejiang University, Hangzhou 310058, China
[c]Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208, USA

Abstract

The control plane of Software-Defined Networking (SDN) is the key component for overseeing and managing networks. As a software entity, the control plane is inevitable to involve design or logic flaws in its policy enforcement and network control, which can cause it to behave incorrectly and induce network anomalies. Existing approaches mainly focus on policy verification or fault troubleshooting, which have little fault localization capabilities for locating these flaws in production environments. In this paper, we present FALCON, the first FAult Localization tool for the 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 hybrid 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 diagnosing controller faults according to formulated symptoms. Our evaluation shows that FALCON is capable of diagnosing faults in the SDN control plane with low overhead on performance.

Keywords: Software-Defined Networking, control plane, fault localization, network reliability

1. Introduction

Separated from the data plane, the control plane in SDN is logically centralized for networking. It leverages southbound protocols (e.g., OpenFlow (OF)) to govern traffic in the data plane and exposes various northbound interfaces (NBI) for external applications or other high-level affairs (e.g., NFV orchestrator) to control networks. In current SDN solutions [2–4], the control plane performs as a network operating system with various core and application modules[1], where network management policies are implemented as modules' code logics and mutual dependencies. Moreover, for scale network management and high availability, the SDN control plane is typically physically distributed, in which several controllers coordinate with each other via eastbound/westbound protocols. The modular and physically distributed nature of the SDN control plane are the significant features that guarantee the flexibility and capability of network service provision.

Unfortunately, as a software system meeting with complicated network dynamics, the SDN control plane is error-prone [7–9]. The control plane is typically reactive and event-driven that it detects input events (e.g., OF messages and NBI requests), processes them and takes actions following specific code logics. Thus, the root causes behind faults in SDN, like network anomalies (e.g., unreachability or forwarding loop) and incorrect NBI responses, are usually flaws in control logics implemented in the SDN control plane [7, 10]. However, figuring out these defective logics is non-trivial because they may be *non-deterministic* (i.e., context-dependent), *cross-module* and mixed with *asynchronous* and *concurrent* network activities [7, 8].

To locate the root causes of faults in the SDN control plane, unfortunately, existing solutions have some limitations: 1) Some research efforts [6, 11] use formal methods to verify the correctness of network policies or abstract program models. 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) Black-box testing is another approach to identifying the input event set triggering the controller to fail [7, 8]. Given the set, operators still need to manually locate the root cause in their codes. Hence, how to diagnose faults in the SDN control plane is still an open issue. The three-layered network architecture and distributed control plane make

* A preliminary version of the article has been published in Proc. of 2019 IFIP/IEEE International Symposium on Integrated Network Management (IM) (Washington DC, USA, April 8-12, 2019) as a 7-page mini conference paper [1]. Enhancements over the conference version are highlighted in Section 1.

* Corresponding author
Email addresses: yyb@whu.edu.cn (Yinbo Yu), xing_li@zju.edu.cn (Xing Li), kaibu@zju.edu.cn (Kai Bu), ychen@northwestern.edu (Yan Chen), yjf@whu.edu.cn (Jianfeng Yang)

[1] It is also called *application agent* [2], *plugin bundle* [5] or *control program* [6]. We use them interchangeably.

previous network or software diagnosis mechanisms inapplicable. Thus, we need to touch the inner side of the SDN system to point out which part of the control plane 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 the SDN control plane. FALCON is a gray-box solution. It performs a *rule-based hybrid tracing* mechanism on controllers' bytecode to precisely track the *system behaviors* of SDN at runtime, including northbound/-southbound interactions between two adjacent planes, program executions inside the controller, as well as westbound/eastbound interactions among distributed controllers. With these behavior data, FALCON further models them through a deterministic *context-aware* model mechanism and mines dependencies among input events as the collaborative behaviors. These models under normal conditions are regarded as diagnosis *references*. When faced with a failure, FALCON first identifies the faulty behavior models and corresponding references, and then performs *differential checking* on them. The differential checking is a mechanism which symmetrically compares these models and points out the causality of their differences to answer the diagnosis question about not only *how* the fault occurred by providing a minimal set of input events, but also *why* it occurred by identifying a minimal set of state differences in relevant code execution paths. We aim to determine the causality with minimal but sufficient information to describe a root cause.

We have built a prototype for OpenDaylight (ODL) platform [3]. It conducts online system behavior monitoring for the control plane in a production environment and performs offline diagnosis with an event replay mechanism in a simulation environment to avoid affecting other normal services. Specifically, when a failure occurs, it sends the recent behavior models and references recorded in the production environment to the simulation environment, where we use an extended STS simulator [7] as a replay engine to simulate data/application planes and replay input events for fault diagnosis. We evaluate FALCON with several types of faults in SDN controllers. The result attests its capability to reveal root causes with involving low performance impact, even optimizing the efficiency of controllers handling of input events under the relatively low workload. Such a performance impact is thanks to bytecode optimization enabled by our used instrumentation tool.

We highlight the major contributions to diagnosing faults in the SDN control plane as follows:

- A comprehensive study of faults in the control plane (§2.1);
- A rule-based hybrid tracing mechanism built on bytecode instrumentation, which tracks system behaviors in the control plane (§4);
- An online system behavior modeling mechanism which deterministically models SDN system behaviors (§5);
- A fault localization mechanism based on differential checking and static analysis which locates precise root causes of occurred failures (§6);
- A complete implementation of FALCON for ODL (§7) and an evaluation to attest its practicality (§8).

This paper is an extended version of the work presented [1], which has illustrated the effectiveness and practicability of FALCON for the single controller mode. Besides giving details on the background of SDN fault diagnosis (§2), this paper enhances FALCON for the physically distributed SDN control plane with many techniques, such as adding the *static recovery* mechanism (§4.4) for obtaining fine-grained system behaviors, extending the system behavior modeling with detailed trace graph construction, fast model comparison, and cross-controller trace behavior association (§5.1), presenting the detailed fault localization algorithm (§6), more comprehensively implementing FALCON (§7), evaluating its diagnosis capability (§8.2) and analyzing its performance (§8.3).

2. Background and Motivation

To motivate our solution, we conduct a fault measurement in the SDN control plane and review related works focusing on diagnosing faults in SDN networks.

2.1. Faults in SDN control plane

Designing a feasible fault localization technique for SDN requires a deep understanding of SDN faults. Thus, we survey controller-related faults (also called bugs) found in literature [6–8, 10, 12, 13] and report the first analysis of bugs in a real SDN controller bug repository, ODL Bugzilla [14], in which we analyze all 298 confirmed bugs of ODL kernel projects [5] until October 16, 2017. For clarity, we classify these faults into three categories in accordance with their root causes as follows:

Logic/design flaw: To manage networks, various network policies are implemented in SDN control software with specific *code logics*. However, these logics may not always be designed correctly and even conflict with each other due to insufficient domain knowledge or misplaced assumptions [6, 7, 15]. For example, in [16], a bug is found that it can make ODL generate `FLOW_MOD` packets with improper fields set hierarchy and finally cause control/-data state inconsistency. The culprit is the incorrect code logic of `FLOW_MOD` generation. In addition, input events in some specific order may hit some unconsidered corner cases of the current design and be processed incorrectly or discarded directly, even trigger harmful data races which can crash the controller [13]. We name this type of faults *logic/design flaw*.

Coding mistake: Careless programming in the implementation of code logic 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

Table 1: Fault types in ODL Bugzilla and the comparison of FALCON and related work (●=diagnosed, ◐=partially diagnosed, ○=not diagnosed).

Fault Category	Proportion	CT	PA	PV	Falcon
Logic/design Flaw	66%	●	●	●	●
Coding Mistake	12%	◐	○	○	●
Performance Anomaly	22%	○	○	○	◐
Providing Root Causes		×	✓	×	✓
Adapting to dynamic changes		✓	×	✓	✓
Introducing low overhead		✓	✓	✓	✓

is not possible to exhaust all of them, e.g., incorrect usage of service identifier [17]. In [12], the author also showed that there are a lot of unreasonable memory allocations in controllers which can cause controllers to crash.

Performance anomaly: SDN controllers often suffer from centralized bottleneck problems in practical applications [12]. One reason is that they are usually installed on common servers with limited processing and I/O capabilities. In addition, their inherent asynchrony and concurrency also exacerbate performance issues [12] and result in various failures, e.g., partial failure in batch operation, message timeout or omission, data race, and even system crash. We mainly focus on these performance anomalies that can lead to different internal executions at different runs. For example, the delay in data reading due to I/O delay may introduce data race among concurrent threads.

We summarize our measurement in Table 1. *Logic/design flaw* is the most popular categories (66%) in all analyzed bugs and usually has a higher need for diagnosis [14]. Furthermore, we observe that most of logic/design flaw bugs and partial ones in the other two categories can raise abnormal code execution traces in the control software, which are deviated from the desired execution traces where these bugs will not happen. This observation inspired our *fault localization* mechanism based on differential checking for execution paths.

2.2. Related Work

Given that the SDN controller is a software entity, diagnosing these preceding faults certainly requires some general diagnostic techniques proposed for common software, such as static analysis and dynamic detection. Many studies based on software diagnosis have been proposed for SDN fault diagnosis. We classify these related works into three categories as follows:

Controller Troubleshooting (CT) focuses on identifying an input event sequence which can trigger a fault in controllers. Scott *et al.* [18] proposed a troubleshooting tool (W^3) for SDN networks. It can compare the high-level policies (specified in the control plane) with low-level configurations (installed in the data plane) to find policy-violations and then identify the minimal causal set of events by reproducing the violation with the arbitrary sequential ordering of interaction events in a simulation environment. Based on W^3, the authors further designed STS [7], which involves fuzzing testing to test controllers and identifies the exact input events triggering occurred failures via event replay and decremented event elimination. Jury [8] attempts to detect which controller in a cluster is erroneous by comparing the action differences for the same input event. CONGUARD [13] is proposed to find data races in SDN controllers by changing the order of input events to trigger competitive read-write operations.

Program Analysis (PA) is often used to analyze the correctness of SDN applications with desired properties [6, 11, 19]. NICE [6] uses model checking and symbolic execution to model a control program and figure out its invariant violations. Nelson *etc.* [11] model two versions of an SDN control program written in a declarative language to find their differential properties and counterexamples. DiffProv [19] models the status of each flow rule generated by applications as a provenance tree and performs a differential provenance to identify differences between faulty and correct trees for fault diagnosis. It is similar to our solution but focuses on the correctness of flow rules.

Policy Verification (PV) [20–22] examines the network models built from OF rules with a set of network invariants, e.g., no forwarding loops or no black holes. It incrementally builds the network model as the network evolves by monitoring change commands (OF rules) generated from the controller. With the model, it verifies if every change on the network can violate defined invariants or affect existing traffic. Hence, these works can check the correctness of network policies in the controller in real-time.

We further compare these works, as well as FALCON in Table 1, in terms of 4 solution goals: 1) covering more types of faults; 2) providing root causes; 3) adapting to dynamically changing networks/controller architectures; 4) introducing low overhead to the control plane. CT can basically cover two bug categories, but they cannot provide the detailed reason why the controller is faulty. Although PA can provide root causes, manual or static analysis they relied on makes PA suffer from state-space exploration problem and difficult to be applied in dynamic changing productive SDN controllers. PV approaches can only indicate if there is a fault in the current network and are complementary to FALCON that they can use their detection results to trigger FALCON for detailed fault localization. These three kinds of works still suffer from limitations for diagnosing controller faults at runtime in a production environment, which motivates us a two-environment-based diagnosis mechanism. Since FALCON tracks runtime system behaviors and diagnoses faults in an additional simulation environment, it can provide more diagnosis capabilities and low impact on performance.

3. Overview of Falcon

In this section, we give the overview of FALCON from three aspects: the basis of our diagnosis mechanism, major faced challenges and the architecture of FALCON.

3.1. Basis of FALCON

A common software fault diagnosis problem is that given a failure, how we can identify its causality from the software and form the causality as an understandable output. However, under the SDN context, this problem becomes complex since SDN controllers need to simultaneously process dynamic network events from the data plane and northbound requests from the application plane under collaborative logics, as well as cluster events among distributed controllers. Thus, to locate the root cause of an occurred failure, we need to identify these events and relevant internal executions in SDN controllers and associate them as an understandable diagnosis result. To address such problem, we leverage two major properties of SDN control plane faults to seek a feasible solution:

(1) Incorrect internal executions: as described in §2.1, most faults in the SDN control plane are caused by logic or design flaws, which violate correct program logics and cause deviations from desired program executions. Nondeterminism of the control plane makes these faults more complicated. Specifically, the controller follows context-dependent code logics to process input events and changes in contexts may trigger an unexpected run that cannot be handled properly due to defective design and implementation of these logics [7, 10, 13].

(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. As mentioned in §2.1, the disorder of input events can induce a different set of internal invocations inside the control plane which may trigger failures. For example, to build an OF connection, a series of messages[2] are generated in order between the switch and controller, the disorder of these messages will trigger data race [7].

With the first property, we realize that the root cause of the fault exists in execution path deviation. Thus, for locating the root cause, we need to identify where the incorrect internal executions happen. The second property notes that there are dependencies among external input events of the control plane and the fault may be caused by a series of interrelated external input events rather than a single event. Thus, the occurrence sequences of input events are another import data for reasoning root causes. Based on these properties, we design FALCON, a *differential fault localization* system, to locate the root causes of failures in the SDN network which suggests the presence of a fault in the control plane.

3.2. Design Challenges and Solutions

Because of the complexity of SDN environments, implementing FALCON will naturally raise the following challenges, which are addressed in our detailed design:

[2] Only if two `OFPT_HELLO` messages for protocol version negotiation are successfully processed, the standard OF messages can be exchanged, e.g., `OFPT_FEATURES_REQUEST`, `OFPT_PACKET_IN` [23].

***Recording fault evidence*:** Accurate fault diagnosis relies on sufficient evidence. Here, the evidence is the system behavior of the SDN control plane. To trace such system behavior, existing tracing mechanisms, however, suffer from several problems. Deploying a *channel proxy* outside of controllers is a common approach to obtaining interactions between two adjacent planes [21, 24, 25]. Unfortunately, this approach can only provide partial system behaviors and introduces extra communication delay. Software-based *static analysis* and *logging* mechanisms are also applied to diagnosing faults in SDN networks [6, 7, 11, 26]. They, however, may be inefficient for the control plane: static analysis may suffer from the state-space explosion problem and be time-consuming to extract software behaviors from control programs [6]; logging mechanisms (e.g., Log4j [27]) are deployed into source codes in an ad-hoc manner, which is inflexible and may result in incomplete behavior logging with massive noise data [28]. What's more, programmers have to manually analyze log files and diagnose faults depending on likely subjective assumptions and experiences, which is tedious. To address this challenge, we design a rule-based hybrid tracing mechanism in §4 to record running contexts via bytecode instrumentation and recover fine-grained execution paths via static analysis on controllers' bytecode.

***Modeling system behaviors*:** FALCON aims to model system behaviors from dynamic trace data. However, several factors challenge it. Firstly, due to the *concurrency* of networks, multiple event (e.g., OpenFlow messages) processing tasks in the controller are executed in parallel and thus, collected traces are interleaved. Even in a single task, various *asynchronous operations* complicate the execution traces and there is no unique identifier propagated through internal invocations. Secondly, the existence of cross-controller invocations (e.g., leader election) distributes trace data in different cluster nodes and then poses challenges to analyze these distributed data, especially when facing potential unsynchronized internal time. Therefore, it is difficult to associate these data with their tasks. Moreover, the *non-determinism* of control logics means with the same input, the controller may exhibit different behaviors under different system contexts [7, 8]. Thus, given an input event with concrete values, it might be hard to infer its deterministic execution path. To address this challenge, we design a context-aware modeling mechanism in §5 to cluster trace data and model their causal relationships with concrete contexts as a context-aware behavior model, with which we can provide deterministic models of system behaviors.

***Diagnosing an occurred fault*:** In the SDN environment, fault diagnosis is faced with not only various failure symptoms but also complicated behaviors in different planes. As we discussed in §2.2, how to diagnose these faults in the control plane quickly and accurately is still an open issue; the existing SDN fault diagnosis techniques (i.e., CT, PA, and PV) are insufficient to reveal the root causes of faults and adapt to a distributed SDN control

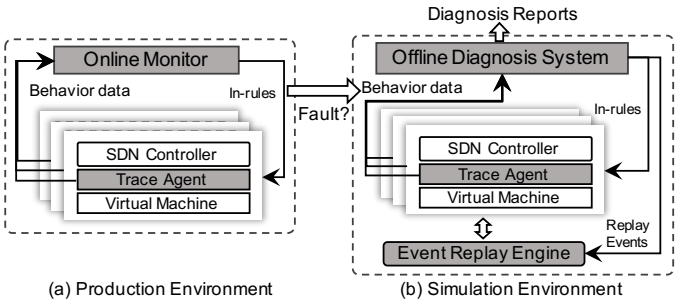

Figure 1: An overview architecture of FALCON.

plane's dynamic changes. In order to address this challenge, we formulate symptoms occurred in different planes as the diagnosis input and design a differential checking mechanism to locate the root causes (§6). 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.

3.3. Architecture of FALCON

Given these challenges of fault diagnosis in the SDN environment, we present the architecture of FALCON in Fig. 1, which contains two parts: *production environment* and *simulation environment*. The production environment records and models system behaviors in real production runs and the simulation environment provides the fault localization. Note that the two environments are reasonable in the real world. Support engineers usually collect system configurations and faulty data in productions provided by users and reproduce failures in this own simulation environment to understand and diagnose them without interfering with the production environment.

In the production environment (Fig. 1(a)), we deploy a *trace agent* inside every controller in the cluster and an *online monitor* outside of the distributed control plane. These trace agents track activities in controllers at runtime and send them out. The online monitor collects and aggregates the trace data, models them as system behavior models and stores them as *references* when there is no fault. Note that FALCON does not provide failure detection function and relies on operators reporting if a failure occurs. Operators can use existing network failure detectors (e.g., ping and traceroute) and log file console to find if there is a failure occurred in the network or controller. When a failure occurs, we transmit the recent behavior models to the simulation environment since there must exist a set of models reflecting the fault's causality.

To diagnose the failure, a new controller cluster is instanced in the simulation environment (Fig. 1(b)) with the same configuration and internal states of the production one through controller *restore* mechanisms [29, 30]. We leverage an *event replay engine* to simulate the data/application planes and reproduce practical failures by strictly 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.

4. Hybrid System Behavior Tracing

In this section, we present a hybrid system behavior tracing mechanism based on bytecode[3] instrumentation for dynamically recording SDN system behaviors and static analysis for recovering fine-grained execution paths. 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. Combining with static recovery inside the modules, FALCON achieves a balance of accurate invocation path construction and low performance cost.

4.1. Target data

In order to get enough data which can reflect system behaviors in the SDN control plane, we should be clear about what data we need to record. Since the control plane is reactive and event-driven, the interactions (i.e., input and output events, e.g., OF messages and NBI requests) between the control and other planes need to be tracked. These interactions trigger the internal processing inside the control plane. Thus, we also need to record internal method invocations to reveal the execution paths of the internal processing. In addition, system state changes can affect the controller's behaviors due to the non-determinism. Therefore, we need to record the reading and writing operations of values in the database, which represent the current system states.

From the perspective of the controller cluster, although most tasks are processed in a single controller node, the handling of some special tasks involves invocations across controller nodes (e.g., *routed RPC*). In addition, the *roles* of controller nodes in a cluster also have a significant impact on event processing. For example, only the *leader* controller of a database *shard*[4] can perform a *write* operation on it and the other write operations of the shard on *follower* controllers need to be routed to the leader. Therefore, to build a global system behavior model, the identities and roles of controller nodes, as well as important cluster events (e.g., leadership changes) need to be taken into account and carefully recorded.

4.2. Dynamic Tracing

Dynamic tracing can provide the above sensitive execution information of the SDN control plane at runtime, but

[3] Bytecode is a form of instructions compiled to run.
[4] A database shard is a horizontal partition of data in a database.

at the cost of computing performance. As a software entity, not all of the SDN controller's execution information needs to be dynamically recorded. Given some key 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 [3, 4]: 1) Their modular nature indicates that their most of event processing tasks are handled under the collaboration among multiple modules; 2) All modules operate on a logically centralized database; 3) Invocations among modules depend on pre-defined module interfaces, e.g., RPC and Notification; 4) Faults in controllers commonly originate in logic flaws inside a module and then may propagate to other modules through invocations or database operations. Thus, controlling the dynamic tracking of the controller's internal invocations at a module-level is sufficient for providing dynamic contexts, which can highly reduce the amount of inserted codes, thereby greatly lowering the overhead. Taking ODL as an example, we can intuitively feel the quantity gap between module-level and method-level: To manage networks, ODL often installs around 300 modules, each of which may offer several to hundreds of class files, and a class may contain several to dozens of methods.

4.3. Rule-based Instrumentation

To track the system behavior data at module-level, bytecode instrumentation is an efficient approach, which allows users to insert specific code into programs and track their desired code behaviors. However, it is typically challenging for operators to determine where and what code can be instrumented. What's more, they may not be familiar with bytecodes. To address these problems, we design a rule-based instrumentation mechanism to ease the instrumentation process. With this mechanism, operators only need to specify their expectant tracing feedback in instrumentation rules (abbreviated as *in-rules*), and then the instrumentation mechanism will automatically translate these in-rules into bytecodes and insert them into corresponding positions in controller bytecodes.

An in-rule is a <`match, action`>tuple (see Fig. 2). The `match` field is used to match against bytecodes for specifying 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 of modular object-oriented-programming-based software to focus the in-rule on the `method`'s code segment. The `call site` represents the location of instrumentation, which is defined by a location before (`B`) or after (`A`) a bytecode instruction with the line number in the code snippet. The `action` field defines execution contexts that need to be profiled, e.g., controller ID, thread, timestamp, invocation type, and variable values. For convenience, in Fig. 2, we use a source code segment rather than bytecode with an in-rule as an example, in which the in-rule is used to

```
0:public class PacketDispatcher {
1: public void dispatchPacket(payload, ingress, srcMac, destMac) {
2:   NodeConnectorRef src = inventoryReader.getControllerSwitchConnectors(srcMac);
3:   NodeConnectorRef dest = inventoryReader.getNodeConnector(destMac);
4:   if (src != null)
5:     if (dest != null)
6:           sendPacketOut(payload, src, dest);
7:     else floodPacket(nodeId, payload, ingress, src);
8:   } }
```

Match				Action
212	PacketDispatcher	dispatchPacket	6+sendPacketOut+B	Thread, event, payload, src, dest, etc.

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

track contexts of *sendPacketOut* in the module with ID 212 (ODL L2switch Arphandler module).

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 OF messages) and list the invocation interfaces (e.g., RPC, notification) that are used in invocations among modules. Then, FALCON transforms the corresponding controller bytecodes into control flow graphs (CFGs), searches the invocations of these interfaces on them, and automatically generates in-rules for tracing them. Finally, FALCON translates these input and generated in-rules into bytecodes and inserts them into the controllers according to in-rule `match` fields. At runtime, these execution contexts defined in in-rules will be profiled and output as trace messages.

4.4. Static Recovery

Although for many faults, the deviations of faulty execution paths can be reflected in module-level trace messages (about 70% according to our fault measurement, e.g., data race among modules), there are still some faults that need method-level invocation paths to figure out execution deviations as the root cause. Adding more in-rules to obtain fine-grained trace messages is not feasible due to overhead. Thus, FALCON leverages mature control flow analysis techniques on bytecode to recover fine-grained executions from collected module-level trace messages. Specifically, given two adjacent module-level trace messages originated in the same module, we extract their locations (i.e., the `match` field of their in-rules) and perform static analysis to identify the execution path inside of the module between their locations. Thanks to concrete contexts in the two messages, we can narrow down the state space of static analysis and obtain the deterministic execution path. Note that we do not consider the invocation routing among modules since it is application-agnostic and maintained by controllers.

5. System Behavior Modeling

Given trace messages, we now address the problem of system behavior modeling. Starting with a trace message representing the beginning of a task which handles an input event (e.g., an OF message or NBI request), we process heavily interleaved trace messages and identify relevant internal invocation nodes and their causal relationships to

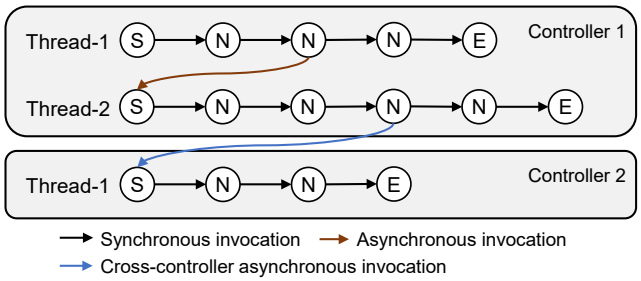

Figure 3: A trace graph for a control plane task.

```
NodeConnectorRef src=inventoryReader.getControllerSwitchConnectors(…);
NodeConnectorRef dest=inventoryReader.getNodeConnector(…);
if (srcConnectorRef != null) {
    if (destNodeConnector != null)
        sendPacketOut(payload,srcConnectorRef,destNodeConnector);
    else
        floodPacket(nodeId, payload, ingress, srcConnectorRef);
} else{……}
```

Conditional branch

Figure 4: Conditional branch example in controller software.

construct a *context-aware* model at run-time. We further perform backtrace on mined 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.

5.1. 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*. As shown in Fig. 3, each operation is executed within a thread with several synchronous invocations and may also involve other operations through asynchronous invocations. In a controller cluster, asynchronous invocations can be further divided into asynchronous invocations within a controller node and across controllers for handling some special tasks (e.g., routed RPC). Thus, an operation can be triggered by an input event or an asynchronous invocation. In an operation, these synchronous invocations can be linked with their *happens-before* relationships as an invocation chain graph. Among operations, their *asynchronous* invocation relationships can be used to link these chains into an invocation tree graph. With these two types of relationships, invocation messages for processing a task can be constructed as a trace graph. Moreover, given different contexts, a task may produce multiple heterogeneous trace graphs due to non-determinism. Fig. 4 is an example that various conditional branches (e.g., *if* $\cdots$ *else*) with different contexts can result in various execution paths.

To model such behavior of each task, we, however, are first faced with heavily interleaved trace messages. Thus, we cluster trace messages for different tasks and then associate the trace messages of each task with two types of causal relationships (*happens-before* and *asynchronous*) to construct them as a trace graph. Finally, we combine heterogeneous trace graphs of each task as a context-aware model (CAM). We describe the construction as follows:

Trace graph. Once getting a trace message, FALCON transforms it into a graph node by node template defined in in-rule's `action` field, i.e., a set of invariant keywords (e.g., "event=") and variables. Inside a single controller, the node shall belong to an operation of a task (i.e., a chain graph). In the controller cluster, each controller handles transactions relatively independently and only makes cross-controller invocations in limited circumstances (e.g., remoted RPC, data change notification). For a task, each controller has a relatively complete trace graph and these trace graphs can be associated with these cross-controller asynchronous relationships, which we called the *locality* of cross-controller invocation. Hence, the node shall be clustered into the corresponding controller for graph construction. FALCON clusters the node into a growing *chain graph* of an operation according to its both controller ID (I^c) and thread ID (I^t). Each chain graph starts with an initial node representing an operation, associates all synchronous invocations belong to the operation by their happens-before relations with the same I^c and I^t, and is completed by the terminal trace message of the operation.

We further combine chain graphs according to their asynchronous relationships inside a controller and across controllers. Since there is no identifier propagated through asynchronous invocations, we design a *multi-identifier* correlation mechanism, in which we construct a tuple containing multiple identifiers to define asynchronous callers and match asynchronous callees. Specifically, the tuple contains the caller's I^c, I^t, location in its chain graph, timestamp and hashcode-based abstraction of its variable values. If the caller's chain graph originates from another asynchronous invocation, we also add a parent-child path between the graph and its parent graph into the tuple. We then use this tuple to match asynchronous callee nodes in different controllers. Finally, all chain graphs of a task distributed in the controller cluster are combined into a global trace graph. Alg. 1 describes the detailed trace graph construction.

Context-aware model. A CAM contains three kinds of edges (see Fig. 5): 1) A *concrete* edge has a pair of preceding and succeeding invocation nodes representing their happens-before relationship; 2) A *fork* edge has multiple succeeding nodes (one is a concrete successor and others are asynchronous callees), which models asynchronous relationships; 3) A *contextual* edge has different succeeding nodes under specific contexts, which is context-aware for modeling data-dependent code logics. We say that in the code logic between the two nodes of a concrete edge, if there is a conditional branch (e.g., in Fig. 4) varying the edge's succeeding invocations under different condition values (i.e., contexts), the edge is then transformed into a contextual edge with an additional context field to record the condition.

We take Fig. 6 for example to discuss the construction of CAM. Firstly, to combine the trace graphs belonging to the same task, we need to distinguish the heterogeneity of each new coming trace graph. However, compar-

Algorithm 1: Online trace graph construction.

Input : A trace message m from the trace stream
Output: A tree graph
Global : Pool Set Ω of $<I^c, I^t, g>$; Tuple set B for asynchronous invocations; Uncompleted tree graph set G;

```
1  n ← TransformIntoLogNode (m);
2  g ← ∅;
3  if n is an initial node then
4      if <I_n^c, I_n^t> ∈ Ω then
5          g ← PutToTree (Ω,n,G,B);
6      Ω ← Ω∪InitializeChain (n);
7  else
8      g_n ←getGraph (Ω,I_n^c,I_n^t) ∪n;
9      if n is a terminal node then
10         g ← PutToTree (Ω,g_n,G,B);
11     else
12         Ω ← Ω ∪ g_n;
13         if n is an asynchronous caller node then
14             B ← B∪ GenerateTuple (g_n,n);
15 if g ≠ ∅ and IsComplete(g) then return g;
16 else return ∅;
```

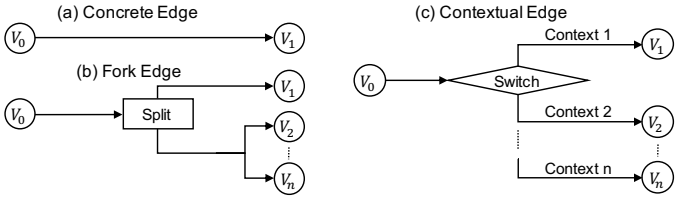

Figure 5: Three types of graph edges.

ing two trace graphs node by node from the root node is time-consuming. Thus, we simplify this procedure by abstracting each trace graph as a hash-based *skeleton tree*. In the skeleton tree, a node is an *abstraction* of a chain graph and edges follow original relationships among chain graphs; the *abstraction* of the chain graph is a hash value for the string concatenation of its nodes' ID in order; the node's ID is a hash value for the string concatenation of its controller ID, module ID, event type, and variable names. This skeleton tree is incrementally built up as the trace graph construction. Through this *skeleton tree comparison* mechanism, we quickly confirm whether a new coming trace graph is identical with an existing trace graph (or a CAM) and if not, where are their differences. From their differences, we identify the conditional branch that leads to the two different edges (e_1 in Fig. 6 (a) and (b)) from the CFG of corresponding bytecodes. We then combine the two edges into a contextual edge (ce_1) with the branch that decides the succeeding nodes according to their contexts. Fig. 6(c) depicts the final CAM, in which from V_b has two possible succeeding nodes: V_c and V_d.

5.2. Augmentation with Model Dependency

Since SDN controllers are event-driven, their contexts are mainly introduced by external input events. Thus,

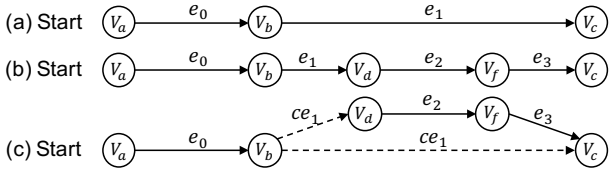

Figure 6: An example of a trace graph combination.

Algorithm 2: Fault localization

Input : f, S_f, S_r
Output: $\Gamma(f)$

```
1  C_f ← ParseSymptom (f);
2  M_f, M_r ← FaultyModelLocating (C_f, S_f, S_r);
3  N_d ← DifferentialChecking (M_f, M_r);
4  S_c ← StaticAnalysis (N_d);
5  foreach context ∈ S_c do
6      context' ← ModifyContext (context);
7      if EventReplay (context', S_c) ⇒ f then
8          S_in ← RelatedModelMining (context);
9          C_f ← C_f∪ (S_in, context);
10 Γ ← ResultAggregation (M_f, C_f);
11 return Γ;
```

contexts in these conditional branches come from their input events or previous tasks. Taking Fig. 4 for example, the action (send `PacketOut` or flood packet) of processing a flow rule request depends on if the destination host has been recorded in the controller's database. Following the second property in §3.1, there may exist *temporal dependencies* among several input events and their corresponding internal system behaviors. A temporal dependency is an association property of two or more input events generated for collaborative services. Mining such dependencies among task models can further address non-determinism and provide references in another dimension for diagnosis since many faults are context-dependent.

To mine temporal dependencies among task models, we leverage static analysis to figure out data dependencies of their control logics. We start from the contexts of branches in contextual edges to identify their 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, we iteratively backtrack current and previous models to search the operations who insert or update its value and identify their corresponding input event or the sequence of input events. If the context is introduced by a previous input event I_s, where s is the input value set, we say that the current task model contextually depends on I_s. With such dependencies, we further associate these two task models to augment mined CAMs. Such models are stored as *references* which result in no failures.

6. Differential Fault Localization

In this section, we discuss how FALCON uses mined models to diagnose faults according to their symptoms. The procedure of FALCON's differential fault localization

consists of 3 phases: 1) parsing the failure symptom to identify the faulty models and corresponding references; 2) symmetrically comparing them to find their differences; 3) performing static analysis from their differences to identify the related conditional branches and contexts. We summarize this procedure in Alg. 2. FALCON mines a set of system behavior *references* (denoted by S_r) from the production environment. When faced with a failure with the symptom f, FALCON takes the recent behavior model set S_f (which contains the models triggering faults) and f to locate the *causality* Γ as follows.

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 control plane, 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')
```

6.1. Faulty Model Identification

Given the symptom f of a failure C_f, FALCON searches the faulty models M_f and their references M_r from recorded trace data (Line 2). Since the controller needs to simultaneously handle a large number of collaborative input events from both southbound and northbound interfaces, it is challenging to identify valid faulty models from massive system behavior models. An explicit symptom typically has a timestamp recorded in the log file, so FALCON can search related models happened before this timestamp. For implicit symptoms without precise timestamps, FALCON starts from the latest model to search the related faulty models which are different from the corresponding references in S_r.

6.2. Differential Checking

In this phase, FALCON systematically compares the faulty and the reference models from their root nodes to identify their differences (Line 3). The different node is denoted by N_d. Taking Fig. 7 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* results in 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 not only the differences and faulty execution path but also the key contexts causing this deviation. Since the goal of differential checking is to find the differences between two models, which is the same as the model

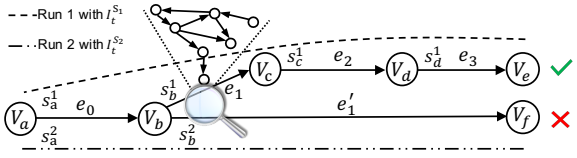

Figure 7: An example of differential checking.

comparison process for the CAM construction described in §5 (i.e., *skeleton tree comparison* mechanism), we can reuse this mechanism to figure out their differences and corresponding contexts leading to these differences.

6.3. Static Analysis

With the different node N_d (i.e., V_b in Fig.7), we conduct static analysis on its bytecode to find out the related conditional branch and the contexts S_c in it (Line 4). 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 (Line 7). In each replay, we change partial contexts and replay the changed input event sequence to check if the failure can still be reproduced. If the failure cannot be reproduced after modifying a context, we regard the context as a key context that triggering the failure. What's more, 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 perform *backtrace* to mine input events S_{in} that have modified these contexts (Line 8). Finally, we aggregate the results to generate the final diagnosis report, which consists of the faulty trace graphs, the faulty CAMs and the corresponding references, and a series of key contexts with related input events (Line 10).

7. Implementation

FALCON is implemented in Java with more than 15,000 lines of code, including trace agent, online monitor, offline diagnosis system, excluding event replay engine. In this paper, FALCON is only evaluated in Java-based controllers. Nevertheless, FALCON is generous since we only need to adopt different underlying instrumentation tools and modify the in-rule translation for other language-based controllers, e.g., Ryu with Python equip [31].

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 [32], which allows us to dynamically instrument SDN controllers and provides control/data flow analysis on bytecode. We then use an inter-thread messaging library (LMAX Disruptor [33]) to deliver trace data from multiple running work threads to the agent thread which performs data transmission. The agent is dynamically attached to the Java virtual machine (JVM) running an SDN controller through JVM attachment mechanism and is remotely controlled by FALCON's Online Monitor to install *in-rules* and track the controllers

Data transmission and collection: To efficiently deliver behavior data and other program data (e.g., bytecode file location) out to the outside *Online Monitor*, we serialize data via Protobuf[34], an efficient structured data serialization mechanism, and utilize *Kafka*[35] *producer* to compress these data and transmit them out. The online monitor subscribes messages from different controllers on the *Kafka server*, aggregates them and finally builds global system behavior models. We use different *partitions* of *Kafka topic* to receive trace messages from different controllers separately, which ensures that trace messages from the same controller are consumed sequentially.

Clock Synchronization: There exist many happens-before relationships among trace messages. Thus, we need these messages' timestamps to associate them. However, since different FALCON components are distributed in different hosts, including these hosts holding distributed controllers and trace agents and the host holding the online monitor, we need to keep their physical clock synchronization to avoid false trace message association. Thus, we use the *Network Time Protocol* (NTP) [36] to deploy the *online monitor* as a time server outside the controller cluster and let each host periodically synchronize time with the time server. With NTP, we can confirm the happens-before relationships of trace messages in accordance with their timestamps.

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 it to support the generation of various northbound requests and southbound network events. What's more, we disable the *automatic leader election mechanism* of the ODL cluster in the simulation environment and use REST requests to designate the leadership of shards leaders to simulate the leadership changing in the controller cluster in the production environment.

8. Evaluation

In this section, we conduct several case studies to assess the fault localization capability of FALCON (§8.2) and design several evaluations to measure FALCON's performance impact on OpenDaylight (ODL) controllers (§8.3). All the evaluations are performed on Ubuntu 14.04 64-bit Linux system running on an 2.2Ghz Intel Xeon E2660 v2 processor (16 cores) with 64GB RAM.

8.1. Instrumentation

As a preparation, our first step is to instrument the controller. We write in-rules for ODL core interfaces and critical cluster events, including *RSETful request/response*, *Restconf operations*, *OpenFlow message in/out*, *RPC*, *Notification listen*, *Data change listen* and *Member event*. Then, we tell FALCON other invocation interfaces we concern about (e.g., *Notification publish* and application-specific interfaces). FALCON generates corresponding in-rules after receiving these desires. With all these in-rules, the controller is instrumented and ready to deliver trace messages outside.

8.2. Case Studies

To evaluate the effectiveness of our localization methodology, we selected 12 real-world bugs from ODL Bugzilla [14], reproduced them and use FALCON to diagnose them. The last two (i.e., Bug-6937, 8885) are bugs in the controller cluster mode. The overall diagnosis results are summarized in Table 2. Note that different from the fault categories in §2.1, we further distinguish *design flaw* from *logic flaw* in this table for clarity, where the former bugs are caused by incomplete designs and the later are caused by conflicting code logics.

The last column of Table 2 describes whether FALCON can identify the root cause of the fault. We can see that for faults from different projects with different symptoms, Falcon plays a positive role in revealing root causes. For bugs from 4969 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 the operator, so that he can get away from the heavy log analysis task and find the problem more easily with the internal view. We do not conduct case studies for faults of *performance anomaly*, but our trace graphs contain the time intervals at runtime among different internal invocations, which can be used as the basis for pinpointing which components are responsible for such performance delays.

For clarity, we take a bug in the single controller mode and a bug in the controller cluster mode from Table 2 as examples to explain the actual fault localization process and demonstrate how FALCON pinpoints the root cause. We run the production environment with connected Mininet[5] as the data plane. We inject different operations into the environment to trigger normal/abnormal network runs.

Bug-5816 [37]: The ODL controller uses the ODL L2switch plugin to provide Layer-2 switch functionality for managing OpenFlow (OF) switches, e.g., processing `Packet_In` messages and generating `FLOW_MOD` messages. There is a *host-expiry* feature in the plugin which can make ODL remove hosts that have not been observed for a long time from the network topology view. However, due to the bug [37], in the reactive mode of L2switch, hosts expired by this feature cannot be discovered again even if *ping* works and new flows get installed on switches.

To reproduce this bug, we built a data plane with 10 fully-connected hosts by Mininet. When we find a host was expired, we ping other hosts on this host and successfully observed the bug. Then, we transmit the resent behavior models to the simulation environment with the symptom.

[5] A OpenFlow network emulator: `http://mininet.org/`

Table 2: Fault localization cases (ODL version: Li=Lithium, Be=Beryllium, C=Carbon, B=Boron, N=Nitrogen).

Bug ID	Description	Symptom	Project(version)	Root cause	Category	Diagnose
3345	Ping will fail in ring topology when a link down	unreachability	l2switch (Li)	incomplete topology update	design flaw	Indirectly
4969	NPE in JSONCodecFactory	NPE in log message	yangtools (Be)	incomplete YANG support	design flaw	Yes
7933	NPE when posting using XML	NPE in log message	netconf (C)	incomplete YANG support	design flaw	Yes
6053	NPE on port creation	NPE in log message	neutron (B)	incomplete JSON parsing	design flaw	Yes
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
7976	Error when closing peers and updating routes	error in log message	bgpcep (C)	race condition	logic flaw	Yes
8157	Recreating a user fails after deleting it	error in log message	aaa (C)	defective user deletion	logic 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
6937	Routed RPC in cluster breaks after isolation/heal	unexpected response	controller (B)	incomplete cluster healing	design flaw	Yes
8885	New node cannot join existing cluster at runtime	error in log message	controller (C)	incorrect shard initiation	design flaw	Yes

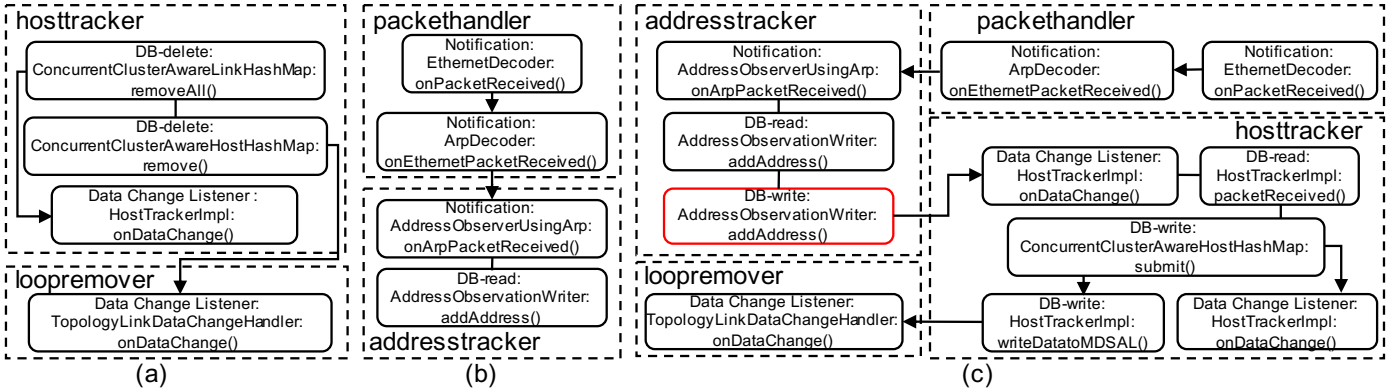

Figure 8: Simplified System Behavior Models in Bug-5816 [37].

After receiving the symptom, FALCON searches for models of OF messages related to the target host and then identifies the deformed host discovery model (Fig. 8(b)) and corresponding host purge model (Fig. 8(a)). Next, FALCON points out the problematic node in L2switch's *Addresstracker* module by conducting differential checking on the deformed model and the reference model of host discovery (Fig. 8(c)). Following static recovery in the Addresstracker module shows that the culprit is the improper *TimestampUpdateInterval*[6] value configured in the module which makes it not update the address in time and L2switch *Hosttracker* module cannot learn the host consequently. Finally, FALCON confirms the root cause with our replay engine and reports the diagnosis result.

Bug-6937 [38]: In the cluster mode, the routed RPC is a popular remote invocation protocol for collaborative processing among distributed controllers. In this protocol, when a routed RPC service is registered on a controller node, all invocations to the service from other controllers in the cluster will be automatically routed to the controller node. However, there is a bug [38] that says after a controller is isolated and then rejoin the cluster, the routed RPC invocation initiated from the controller will fail with a *No Implementation Available* error.

In a 3-node controller cluster, we reproduce this bug by launching routed RPC invocations on a controller node that is rejoined after isolation and other nodes, respectively. As Bug-6937 reports, the invocation launched on the rejoined controller node fails, while invocations initiated on other nodes succeed. Next, we send recently models and the symptom to the simulation environment. With this symptom, FALCON identifies the behavior model of the failed request and its successful reference model. The differential checking on them indicates the difference is in the *Sal-Remoterpc-Connector* module[7]. After performing corresponding static analysis, FALCON points out the key context, *clusterMembers*[8], which does not contain the address of the rejoined controller. It then mines the related models for the context (Line 8 in Alg. 2) and finds that the behavior model corresponding to the processing of the Gossip[9] *UnreachableMember* message involved modifications to the context. With the diagnosis report, we find that the processing of the *UnreachableMember* message removed the address of the isolated controller from the *clusterMembers*, but the behavior model corresponding to the processing of the *ReachableMember* message on behalf of the rejoining controller node did not add the address back to the context. At this point, the design fault of incomplete cluster healing is successfully diagnosed.

8.3. Performance Measurement

FALCON may introduce performance impact on controllers for managing networks. The performance of a controller can be measured in two procedures: network initialization (i.e., topology building) and network maintenance. In the former procedure, the controller needs to

[6] A configurable value describing the update interval for addressing changes in Addresstracker module.

[7] Connecting callers and callees of routed RPCs for invocations.

[8] An array of active nodes in a cluster.

[9] ODL uses Gossip protocol [39] to broadcast and maintain members reachability in a cluster.

discovery network links and generates flow rules to build links among switches. During network maintenance, the controller needs to handle OpenFlow (OF) messages sent from the edge switches for routing packets from their connected hosts. Thus, to evaluate the performance impact of FALCON, we follow the evaluation method designed in [40] and perform evaluations from two aspects: 1) Throughput and latency for network maintenance, and 2) Network topology building time.

8.3.1. Throughput and Latency

Deploying FALCON's trace agent into controllers may introduce delays in processing input events. We evaluated this delay by measuring the controller latency (i.e., how much time it takes to process an event) and throughput (i.e., how many events it can process per second) in processing OF messages and NBI requests 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, we also directly used ASM to instrument ODL (A-ODL) by reusing FALCON's in-rules with a simple value add instruction `action`. Different from the preliminary version [1], we performed our evaluations both for the single and cluster controller modes, in which each ODL controller is run in a virtual machine (allocated with 12 cores CPU and 20 G memory) installed in our server. In the cluster mode, we launched an ODL cluster with three controller nodes[10]. In addition, we ran our benchmark tools directly in our server. We conducted atop ODL controllers (0.6.1 version) and performed each test 30 times. The average results under the two modes are shown in Fig. 9 and 10, respectively.

OpenFlow messages: We measured the performance of controllers in processing OF messages by running CBench [41], a benchmarking tool for testing OF controllers, in latency and throughput mode, respectively. To process OF messages, ODL utilizes three major *L2switch*, *OpenFlowplugin*, and *OpenFlowJava* plugins, in which there totally exist 19 functional modules. FALCON generated 102 *in-rules* to instrument these modules in each controller. We tested the latency by running CBench configured with a simulated switch connecting to 10,000 unique MAC addresses (i.e., simulated hosts). In the throughput testing, the number of switches ranged from 10 to 150, while each switch connected to 200 unique hosts.

In the single mode, as shown in Fig. 9 (a) and (b), A-ODL gains the best performance with lower latency (32.44%) and average higher throughput (21.79%) than ODL. F-ODL gains better latency than ODL (18.16%). When connecting with a small number of switches (less than 70 in Fig.9(b)), F-ODL can also gain better throughput than ODL, e.g., 26.36% with 10 switches. Its throughput performance then starts to be lower than ODL as the number of switches increases to more than about 70. In F-ODL, more threads need to be allocated for the delivery and processing of trace messages, which leads to throughput degradation (8.56% on average). After the number of switches was greater than 70, F-ODL consumed more processing resource than the one the VM can provide. As measured in [42], 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.

In most network scenarios, networks are usually managed by a controller cluster. This mode can improve the network availability, but at the cost of computing resources since more processing resources need to be scheduled for state synchronization and load balance among controllers. As shown in Fig. 10, the performance of ODL cluster is decreased both for processing OF messages and NBI requests than in the single mode, e.g., a higher latency (25.5%) of processing OF messages in the cluster. In the cluster mode, A-ODL and F-ODL both can gain better performance than ODL, i.e., 41.38% and 23.78% in latency, and 37.83% and 15.81% in throughput, respectively. Different from the single mode, F-ODL can always have better throughput than ODL. The fundamental reason lies in that OF messages are balanced among three controllers in the cluster and the number of OF messages required to be processed in each controller is decreased.

RESTful requests: ODL utilizes RESTful protocols to format its NBIs. 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 24 *in-rules* to track Neutron plugin (containing 4 modules) in each controller. We measured the processing latency of RESTful requests by recording the time interval between when a RESTful request is sent out and its response is received in the engine. To evaluate ODL's throughput, we built multiple concurrent connections (ranging from 1 to 28) between the engine and ODL to send requests (each connection will send 1850 requests) and counted the number of responses that can be received per second in our engine.

Fig. 9(c) demonstrates the average result of latency testing in the single mode, where A-ODL decreases the latency in processing GET (44.97%), POST (44.87%), PUT (48.39%), and DELETE (44.41%) requests (45.35% reduction totally) and F-ODL is also faster than ODL (43.96%, 42.33%, 45.66% and 42.44% latency reduction (43.54% totally) in processing four these types of requests, respectively). Fig. 9(d) demonstrates the results of throughput evaluations, in which A-ODL and F-ODL both gain better throughput than ODL (47.05% and 28.26% on average, respectively). Different from OF messages, F-ODL always has better performance than ODL in processing RESTful requests in spite of the number of connections increases.

[10]Note that increasing the cluster size can improve the networking capability of the control plane and thus shall release the performance impact of FALCON due to distributed workload. Thus, we do not further change the cluster size to conduct more evaluation.

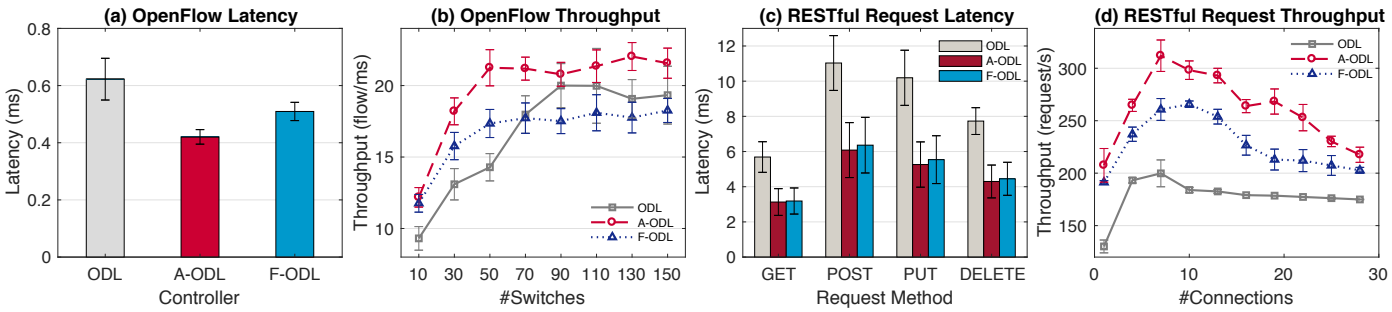

Figure 9: Latency and throughput evaluation of ODL, A-ODL, and F-ODL in the single controller mode. We evaluated latency and throughput impact introduced by FALCON on controllers handling OpenFlows messages and RESTful requests, respectively.

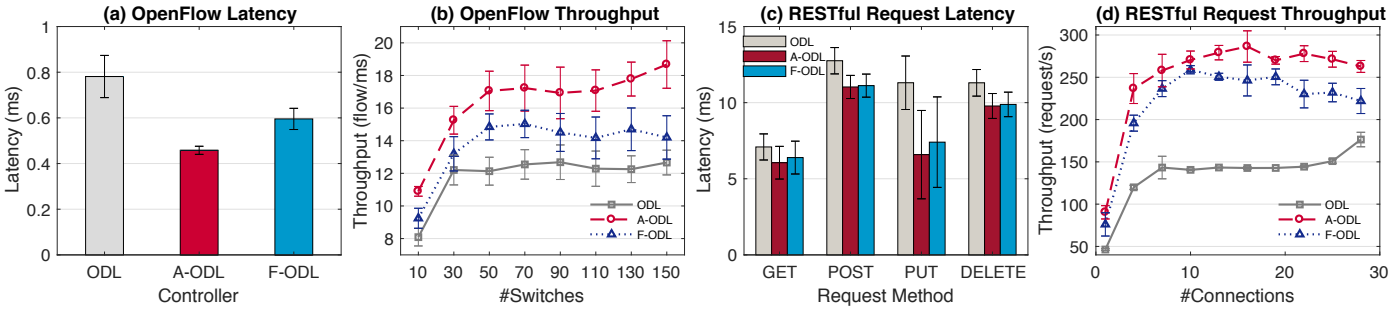

Figure 10: Latency and throughput evaluation of ODL, A-ODL, and F-ODL in the controller cluster mode.

In the cluster mode, as shown in Fig. 10 (c) and (d), A-ODL and F-ODL also obtain better processing performance. But comparing to ODL, the average latency of processing these four types of RESTful requests has only 14.61%, 13.51%, 41.76%, and 13.46% reduction (17.47% totally) in A-ODL, 9.86%, 12.83%, 34.5%, and 12.55% reduction (14.63% totally) in F-ODL, respectively. More difference is that in the single mode, when the number of connections is larger than around 10, the throughput in all ODLs starts to decrease, but does not in the cluster mode. In the cluster mode, the throughput of process requests in ODL increases gradually as the number of connections increases, but A-ODL and F-ODL can still gain better throughput than ODL (87.09% and 63.89% on average, respectively). When the number of connections is more than 8, A-ODL and F-ODL can have a stable throughput, but not exceed the peak throughput in the single mode. But with around 20 connections, A-ODL's throughput slightly decreases. The main reasons for such different performance between two modes and also different from processing OF messages 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.

8.3.2. Network Topology Building Time

The speed of building the underlying physical network topology is another important performance metric for SDN controllers. Therefore, we measured the impact of FALCON on the speed of the control plane handles two common types of network topologies (linear and fat-tree) with different size of switch nodes. The speed can be measured by accounting the topology building time that a controller processes Link Layer Discovery Protocol (LLDP) packets to be aware of network links. Following the evaluation methodology in [40], we build network topologies by Mininet [43] (a system for rapidly prototyping large networks) and extract building time from controller logs. ODL uses a module, called `topology-lldp-discovery`, in `OpenFlowplugin` to process LLDP packets. Thus, we calculate the time gap as between timestamps in starting and ending logs of LLDP processing generated from this module as the building time. Note that the building time also includes Mininet building time [40]. For managing networks, a controller cluster is more scalable than a sin-

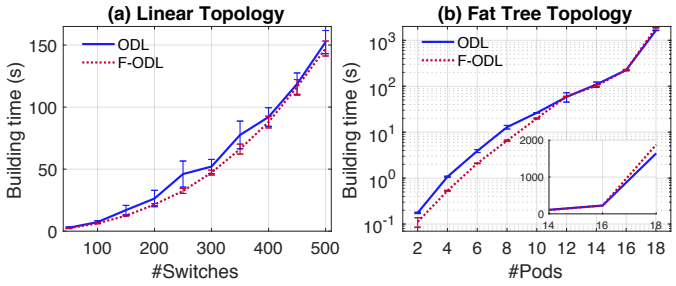

Figure 11: Topology building time evaluation of a 3-controller cluster on (a) linear and (b) fat-tree topology networks. In a linear network, all switches are two-paired except the edges which are connected to one switch. That is for a linear network containing k switches, it has $k-1$ links. In a fat-tree network, k pods mean the network consists of $k^3/4$ hosts and $5k^2/4$ switches with $k^3/2$ links.

gle controller. Thus, our evaluation was performed with a 3-controller cluster. Fig. 11 plots our results on the topic of topology building time.

The linear topology is the simplest one, on which F-ODL and ODL have the performance characteristic similar to the throughput and latency evaluation. That is as shown in Fig. 11 (a), compared to ODL, F-ODL can achieve a more stable and lower building time thanks to ASM optimization. Unlike linear topology, a fat-tree topology has more links, which requires controllers handling more LLDP packets. As we can observe in Fig. 11 (b), with a small size of network topology, F-ODL can outperform than ODL. As the number of pods[11] is greater than 10 (containing 125 switches and 500 links), F-ODL and ODL can achieve similar topology building time, but the one achieved by F-ODL is more stable. After the number of pods reaches 18 (containing 405 switches and 2916 links), the building time achieved by ODL is lower than the one achieved by F-ODL. In our evaluation environment, it shall be a network size where FALCON starts to introduce scalability degradation. The reason is similar to the performance decreasing in a single controller mode that the processing/memory resource required by F-ODL is more than ODL and starts to exhaust all available resource of the controller as the growth of network size. Providing diagnosis functionality for SDN control plane is inevitable to involve scalability issue to networking. We believe this performance impact is acceptable for normal networks since a large network is usually divided into several domains to distribute workload and each domain is managed by a controller cluster for assuring network availability. Thus, FALCON shall be able to run in a larger network with lower performance impact by allocating more computer resource to controllers and increasing the cluster size.

9. Discussion

Our experimental evaluation shows that while FALCON is capable of diagnosing faults in the SDN control plane, it does not introduce explicit performance overhead in most cases thanks to ASM's optimization on bytecode. Here, we further discuss three limitations of FALCON:

Intrusive Profiling: FALCON is a gray-box approach which is intrusive and may induce performance and security issues. Benefiting from ASM's optimization on bytecode, FALCON involves an acceptable overhead on SDN network management. However, incorrect in-rules may introduce errors to controllers. Thus, in-rules' correctness verification shall be addressed in our future work.

Model Completeness: FALCON relies on inserting in-rules to track system behaviors. However, third-party middlewares (closed source or written in other languages) or the incompleteness of in-rules may lead to disrupting of modeling. This problem can be partially alleviated by applying multi-modal similarity check on the input and output of middlewares [44].

Reference Sufficiency: The sufficiency of reference models is the key factor affecting FALCON's diagnosis effect. Existing solutions rely on predefined invariants as the *references* [6, 8, 11], or simply assume that there have been sufficient references [19]. However, the complicated networks and frequently evolved control software suggest that no matter how much effort is spent on software testing, it is hard to exhaust all system behaviors as references before deployed as productions. Since FALCON is deployed on controllers running in the production environment, it can usually get enough normal models to enrich its reference library unless the correct model does not exist at all. Note that this is not a common situation.

[11] In a fat-tree network, k pods mean the network consists of $k^3/4$ hosts and $5k^2/4$ switches with $k^3/2$ links.

10. Conclusion

SDN is an important technique for future networks. In this paper, we have presented FALCON, a system for locating root causes of faults occurred in the SDN control plane. As a gray-box solution, we design a rule-based hybrid tracing mechanism to exploit the internal system behaviors, model these behaviors with a context-aware model and realize a differential fault localization mechanism on two system behavior models to locate the root causes of faults. Currently, FALCON can support both for the single controller mode and distributed controller mode, We have built a prototype of FALCON for OpenDaylight platform. Our evaluation shows that FALCON is practical for real controller runs.

Acknowledgments

This work is supported by National Key R&D Program of China (2017YFB0801703) and the Key Research and Development Program of Zhejiang Province (2018C01088). We thank IM 2019 Chairs and Reviewers for their helpful feedback.

References

[1] X. Li, Y. Yu, K. Bu, Y. Chen, J. Yang, R. Quan, Thinking inside the box: Differential fault localization for sdn control plane, in: 2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), IEEE, 2019, pp. 353–359.

[2] O. N. F. (ONF), SDN architecture, Tech. rep., Open Networking Foundation (ONF), https://goo.gl/hTv77E (2014).

[3] J. Medved, R. Varga, A. Tkacik, K. Gray, Opendaylight: Towards a model-driven SDN controller architecture, in: IEEE WoWMoM, 2014, pp. 1–6.

[4] P. Berde, M. Gerola, J. Hart, et al., ONOS: towards an open, distributed SDN OS, in: ACM HotSDN, 2014, pp. 1–6.

[5] OpenDaylight, Project list, https://wiki.opendaylight.org/view/Project_list, (accessed on 2018-12-10).

[6] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford, A NICE way to test OpenFlow applications, in: USENIX NSDI, 2012, pp. 1–14.

[7] C. Scott, A. Wundsam, et al., Troubleshooting blackbox SDN control software with minimal causal sequences, in: ACM SIGCOMM, 2014, pp. 395–406.

[8] K. Mahajan, R. Poddar, M. Dhawan, V. Mann, JURY: Validating Controller Actions in Software-Defined Networks, in: IEEE DSN, 2016, pp. 109–120.

[9] Y. Yu, X. Li, X. Leng, L. Song, K. Bu, Y. Chen, J. Yang, L. Zhang, K. Cheng, X. Xiao, Fault management in software-defined networking: A survey, IEEE Commu. Surveys Tuts. 21 (1) (2019) 349–392.

[10] B. Chandrasekaran, B. Tschaen, T. Benson, Isolating and Tolerating SDN Application Failures with LegoSDN, in: ACM SOSR, 2016, p. 7.

[11] T. Nelson, et al., Static Differential Program Analysis for Software-Defined Networks, in: FM, 2015, pp. 395–413.

[12] S. Shin, Y. Song, T. Lee, et al., Rosemary: A robust, secure, and high-performance network operating system, in: ACM CCS, 2014, pp. 78–89.

[13] L. Xu, J. Huang, S. Hong, J. Zhang, G. Gu, Attacking the Brain: Races in the SDN Control Plane, in: USENIX Security, 2017, pp. 451–468.

[14] OpenDaylight, OpenDaylight Bugzilla, https://bugs.opendaylight.org/, (accessed on 2018-12-10).

[15] T. Ball, N. Bjørner, et al., Vericon: Towards Verifying Controller Programs in Software-Defined Networks, in: ACM PLDI, 2014, pp. 282–293.

[16] OpenDaylight, Specific flow validation in OpenFlow plugin, https://git.opendaylight.org/gerrit/#/c/3493/, (accessed on 2018-1-25).

[17] OpenDaylight, Bug 8533 - Not possible to invoke RPC on mount points with new Restconf, https://bugs.opendaylight.org/show_bug.cgi?id=8533, (accessed on 2018-12-10).

[18] R. C. Scott, A. Wundsam, K. Zarifis, S. Shenker, What, Where, and When: Software Fault Localization for SDN, Tech. Rep. UCB/EECS-2012-178, EECS Department, University of California, Berkeley, accessed on 2018-6-28 (Jul. 2012).
URL https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-178.pdf

[19] A. Chen, et al., The good, the bad, and the differences: Better network diagnostics with differential provenance, in: ACM SIGCOMM, 2016, pp. 115–128.

[20] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, S. Whyte, Real Time Network Policy Checking Using Header Space Analysis., in: USENIX NSDI, 2013, pp. 99–111.

[21] A. Khurshid, X. Zou, W. Zhou, M. Caesar, P. B. Godfrey, VeriFlow: Verifying network-wide invariants in real time, in: USENIX NSDI, 2013, pp. 15–27.

[22] A. Panda, O. Lahav, K. J. Argyraki, M. Sagiv, S. Shenker, Verifying Reachability in Networks with Mutable Datapaths., in: USENIX NSDI, 2017, pp. 699–718.

[23] ONF, OpenFlow Switch Specification Version 1.5.1, https://goo.gl/Yfh4zh, (accessed on 2018-12-10).

[24] A. Wundsam, D. Levin, et al., OFRewind: Enabling Record and Replay Troubleshooting for Networks., in: USENIX ATC, 2011, pp. 15–17.

[25] T. Nelson, D. Yu, Y. Li, R. Fonseca, S. Krishnamurthi, Simon: Scriptable interactive monitoring for SDNs, in: ACM SOSR, 2015, pp. 1–7.

[26] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, N. McKeown, I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks., in: USENIX NSDI, 2014, pp. 71–85.

[27] Apache, Apache Log4j 2, https://logging.apache.org/log4j/2.x/, (accessed on 2018-12-07).

[28] K. Nagaraj, C. Killian, et al., Structured comparative analysis of systems logs to diagnose performance problems, in: USENIX NSDI, 2012, pp. 26–26.

[29] ODL, Persistence and Backup, https://goo.gl/uydxJQ, (accessed on 2018-12-10).

[30] ONOS, Backup/Restore Tutorial, https://goo.gl/4bh1D1, (accessed on 2018-12-07).

[31] G. Romain, equip: Python Bytecode Instrumentation, https://github.com/neuroo/equip, (accessed on 2018-12-10).

[32] OW2 consortium, ASM: A Java bytecode engineering library, http://asm.ow2.io/, (accessed on 2018-12-10).

[33] LMAX, LMAX Disruptor: High Performance Inter-Thread Messaging Library, http://chronicle.software/products/chronicle-queue/, (accessed on 2018-12-10).

[34] Google, Protocol Buffers, https://developers.google.com/protocol-buffers/, (accessed on 2018-12-100).

[35] Apache, Apache Kafka, http://kafka.apache.org/, (accessed on 2018-12-10).

[36] Network Time Foundation, NTP: The Network Time Protocol, http://www.ntp.org/, (accessed on 2018-12-10).

[37] OpenDaylight, Bug 5816 - l2switch - Expired hosts never come-back after timing out, https://bugs.opendaylight.org/show_bug.cgi?id=5816, (accessed on 2018-12-10).

[38] OpenDaylight, Bug 6937 - Routed RPCs in cluster breaks after isolation/heal, https://bugs.opendaylight.org/show_bug.cgi?id=6937, (accessed on 2018-12-10).

[39] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry, Epidemic algorithms for replicated database maintenance, in: ACM PODC, 1987, pp. 8–32.

[40] L. Andrade, M. Borba, A. Ishimori, F. Farias, E. Cerqueira, A. Abelém, On the benchmarking mainstream open software-defined networking controllers, in: Proceedings of the 9th Latin America Networking Conference, ACM, 2016, pp. 9–12.

[41] R. Sherwood, Y. KOK-KIONG, Cbench: an OpenFlow controller benchmarker, https://github.com/mininet/oflops/tree/master/cbench, (accessed on 2018-12-10).

[42] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, C. Hu, RuleTris: Minimizing rule update latency for TCAM-based SDN switches, in: IEEE ICDCS, 2016, pp. 179–188.

[43] B. Lantz, B. Heller, N. McKeown, A network in a laptop: rapid prototyping for software-defined networks, in: Proc. ACM HotNets, 2010, p. 19.

[44] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, S. Bhattacharya, Anomaly Detection Using Program Control Flow Graph Mining From Execution Logs, in: ACM KDD, 2016, pp. 215–224.