FORTIFY: Software Defined Data Plane Resilience

Umar Farooq*§, Mubashir Anwar†, Haris Noor‡, Rashid Tahir††, Santhosh Prabhu¶§, Ali Kheradmand‖§, Matthew Caesar†, Fareed Zaffar**
*Amazon Web Services, ‖Google, **LUMS, †UIUC, ††Univ. of Prince Mugrin, ‡Univ. of Wisonsin Madison, ¶VMware

***Abstract*—Given the scale and mission-critical nature of production networks today, it is essential to solidify their resilience to link failures. Building this resilience in each application separately is not scalable. In order to minimize downtime, at least some degree of resilience should be built directly into the data plane. Fast Failover groups in OpenFlow offer a mechanism to achieve this, but programming them introduces additional complexity to the existing arduous task of developing an SDN controller application. In this paper, we discuss how this complexity can be decoupled from the controller implementation. We introduce FORTIFY, a transparent resiliency layer that incorporates data plane fault tolerance into any existing controller application without any modification to it. FORTIFY operates as a shim layer between the controller and the data plane, and dynamically transforms the data plane rules computed by the controller to use Fast Failover groups. FORTIFY can be used off-the-shelf, or customized programmatically to choose specific types of backup paths. Experimental results collected on a production testbed demonstrate that FORTIFY is a practical, high-performance solution to data plane fault tolerance in SDNs.**

I. Introduction

Software Defined Networking (SDN) has been shown to be capable of bringing valuable benefits such as increased programmability, fine-grained control, and convenient management of large-scale networks. These benefits make SDNs attractive for critical domains such as finance [1], military [2] and healthcare [3]. Given the strategic nature of such networks, it is imperative that a high degree of availability and resilience be built into them. For example, even a minor disruption in a military network can result in a national security incident by providing a malicious attacker an opportunity to infiltrate, exploit, and disable vital services [4], [5].

To this end, there has been an extensive amount of past work to devise *protection* and *restoration* schemes for computer networks, both SDN and non-SDN [6]–[10]. Approaches for protection focus on pre-determined failure recovery, often leveraging the notion of protecting a "primary" path with a "backup" path allocated for use in the case of failure. In contrast, restoration involves dynamically trying to restore connectivity in case of a failure. While promising, both techniques have their corresponding set of challenges. Protection, for instance, provides speedy recovery at the cost of allocating resources in advance, which in practice often limits the scope of the approach to a limited set of failure scenarios (as resources are limited). On the other hand, restoration is more flexible in terms of handling diverse failure conditions, however the approaches in this category are generally slower compared to the aforementioned protection mechanisms [11], resulting in higher downtimes. Given that even milliseconds of downtime can cause huge losses to organizations, restoration-based schemes may often simply be impractical for larger commercial organizations [12].

§Work was fully performed while at UIUC.

In their nascency, SDNs relied solely on restoration-based schemes for fault tolerance. OpenFlow 1.0, for instance, requires the software controller to detect failures, and perform necessary modifications to the data plane in order to restore connectivity [13]. Over the years however, there has been an increased focus on protection through so-called *data plane fault tolerance*. Data plane fault tolerance refers to the ability of the network data plane to handle failures without involvement of the control plane, allowing for near-instantaneous recovery. In OpenFlow 1.3, data plane fault tolerance is enabled through a feature called *Fast Failover*, which constantly monitors switch interfaces and reroutes traffic through backup paths as required in the event of failures.

While data plane fault tolerance techniques make it *possible* for SDNs to provide high availability, they are by no means straightforward. Given their layered design, SDNs require the administrators to write their own control plane logic, which is filtered down to the layers below. To have admins additionally program for data plane fault tolerance makes the task of network management significantly more challenging and laborious. In contrast, protection schemes in non-SDN networks, such as MPLS link protection [14], takes this burden away from the administrator altogether. We believe this simplicity is one of the major reasons why, in spite of the many advantages of SDNs, most production networks continue to operate legacy solutions.

This problem has gained widespread traction in the networking community. Simplification of SDN programming, including for fault tolerance, has been attempted by numerous works in the past [9], [15]–[20]. There exist a multitude of programming frameworks and libraries for SDN programmers to use when writing their controllers. Proposed solutions typically provide a programming library or framework to automate certain tasks, ranging from computing paths to installing backup rules. However, these works still require the administrators to either modify their applications to incorporate these libraries or rewrite the application in a new domain-specific language [15].

In this paper, we propose a solution that builds on the philosophy [21] that functional and performance goals in SDN should be orthogonal and an SDN application developer should be able to focus purely on the functional aspects. Specifically, fault tolerance, being a performance goal, should be handled automatically and be agnostic to the functional goals of the application. To meet this objective, we introduce FORTIFY, an automated failure tolerance layer which provides data plane fault tolerance by transparently integrating with any SDN controller and providing the resilience needed for production networks. The proposed system quickly reacts to network failures (even multi-link failures) and reroutes traffic dynamically onto pre-computed backup paths to minimize downtime and eliminate delays incurred from control plane processing. We demonstrate

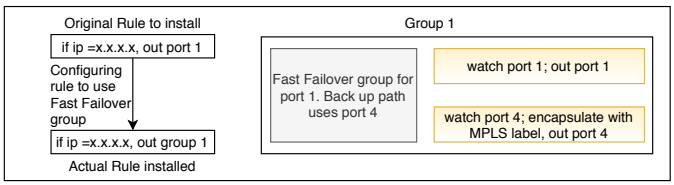


Fig. 1. Original rule for making a routing decision is modified to use a fast failover group entry in a group table instead of directly forwarding the packet to the specified out port.

that FORTIFY can be deployed in conjunction with various OpenFlow controller applications and platforms, even with controllers written for OpenFlow 1.0, *which has no support for data plane fault tolerance*. Our experiments show that FORTIFY achieves near-instantaneous data plane fault tolerance, and incurs very little overhead in regular network environments.

In summary, this work presents the design and implementation of an SDN fault tolerance framework that:

- Provides automatic multi-link failure resilience proactively in the data plane with tolerable overheads.
- Is controller-agnostic and seamlessly integrates with various types of proactive and reactive controllers without affecting their functionality.
- Allows for customizability of backup paths to take into consideration various administrator-defined constraints (such as including certain nodes and links and avoiding others).
- Translates between different OpenFlow versions on the fly to ensure that the fault tolerance mechanisms provide a consistent and compatible view across deployments.

II. Overview of OpenFlow Fast Failover

The OpenFlow protocol [22] is used for communication between an OpenFlow controller and OpenFlow-enabled network switches. The controller disseminates *rules* through OpenFlow messages to switches for routing traffic. These rules are installed in switches' flow tables, and forward traffic by matching against the rules. Newer versions of OpenFlow allow a set of ports to be represented as a single entity called a *group* for packet forwarding [23]. Groups are stored in a *group table*, whose entries are made up of a group ID, group type and a set of *action buckets*, which contain a set of actions related to each individual port in the group. Data plane fault tolerance is achieved through a feature called *Fast Failover* groups. These groups are installed as group table entries, and can be chained to flow table entries using group ID. Each *action bucket* in a fast failover group entry monitors the liveliness of associated port. Whenever a port (or associated link) goes down, action buckets are looked up in the group table entry. The first action bucket whose corresponding port is live is used for packet forwarding without additional input from controller.

III. Design

Our approach is designed to be deployed as a shim layer between the controller and the data plane as can be seen in Figure 2. A layered approach makes our design easily deployable to existing SDNs and allows the failure tolerance layer to be transparent to both the control and the data plane. FORTIFY is comprised of several modules, which we briefly describe below:

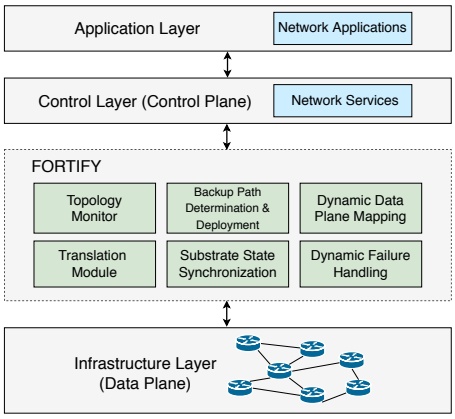


Fig. 2. FORTIFY sits between control and the data plane as a shim layer and intercepts all communication between them.

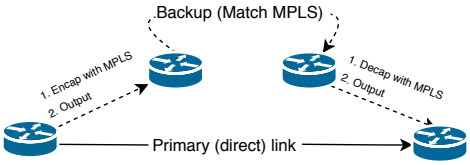


Fig. 3. Tunneling for backup paths through packet tagging.

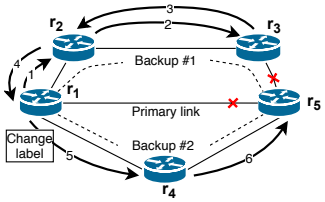


Fig. 4. Design for multi-link failure tolerance. We backtrack along tunnels to source node for link failures in backup paths.

A. Topology Monitor

The first task for FORTIFY is to acquire network state, which it does through the Topology Monitor module. Topology monitor intercepts all control messages between controller and the data plane to construct and store a graph representation of the underlying topology. This is possible from the vantage point of FORTIFY since it sits in the critical path between the control and the data plane. It does not need to actively poll the network by itself, since it can utilize the existing communication between the controller and the network to track changes in the topology. If any change in the network topology is detected, the topology monitor updates the graph abstraction accordingly.

B. Backup Path Determination

Once FORTIFY has the current state of the network topology from the topology monitor, it uses it to compute backup paths for all links in the network. The user provides the resiliency level r, the required number of backup paths for each link, and FORTIFY uses path computation algorithms to determine r alternative paths $p_0,\ldots,p_{r-1}$ for each of those links. This path computation algorithm is configurable according to the required business logic.

C. Backup Path Deployment

Once backup paths have been computed for each link, their forwarding rules get installed into the data plane as fast failover groups. These rules define tunnels along the backup paths, starting at the source of a link, and ending at the destination. The tag for the tunnel is defined uniquely by a triplet: the source of the link, the port number of the link, and the backup path number. The source and the port number uniquely identify the failed link l_p, while the backup path number identifies the backup path p_k for the failed link. A fast failover group, as illustrated in Figure 1, is then created at the source device, which forwards traffic along the primary link while it is up, and when not, redirects the traffic into the backup tunnels. The overall layout for single link failure is illustrated in Figure 3. In the event of primary link failure between two nodes, fast failover groups configured by FORTIFY are utilized to ensure reachability. These groups are used in conjunction with MPLS routing to route packets along a backup path. If there are failures along the backup path (multi-link failures), the packet is routed back to the source node where appropriate forwarding rules on the fast failover group are triggered to use the next available backup path using MPLS tunnels as shown in Figure 4.

D. Dynamic Data Plane Mapping

Once the backup path tunnels and the corresponding fast failover groups have been configured into the data plane, FORTIFY goes into data plane mapping mode. In this mode, it intercepts messages between the controller and the data plane, and modifies them in order to keep the existence of backup paths from the controller. This is important, since it precludes controllers from deleting backup paths and allows FORTIFY to be transparent from the vantage point of the control plane. For this purpose, FORTIFY modifies two types of openflow messages: `OFPT_FLOW_MOD` and `OFPT_MULTIPART_REPLY`. `OFPT_FLOW_MOD` is the message sent to install a new rule into the data plane and is modified to install the rule in the corresponding fast failover group instead. `OFPT_MULTIPART_REPLY` is a message from the data plane to the controller to acknowledge or report installed rules and is modified to extract individual forwarding rule(s) intended by controller from the fast failover group configured by FORTIFY. This allows FORTIFY data plane logic to be transparent to the controller.

E. Translation Module for different versions of OpenFlow

FORTIFY also has a translation module which provides compatibility between different versions of OpenFlow running in the control and the data plane. This translation module automatically infers the version of OpenFlow running on the controller and translates all messages exchanged to and fro accordingly. It should be noted that fast failover was introduced in OpenFlow 1.3 and earlier versions had no notion of it. The translation module allows the configuration of fast failover in the data plane even if the controller is running on earlier versions of OpenFlow that did not support it. This not only reinforces our claim that the controller can be left unmodified for achieving failure resilience, but also demonstrates that our design is robust against the challenges of practical deployment.

F. Dynamic Failure Handling

Like all networks, SDNs are prone to both transient and permanent link failures. Transient failures can cause unnecessary *convergence*, leading to high control traffic in the data plane. To prevent this, FORTIFY provides the option to delay sending updates about link failures to the controller, handling transient link failures seamlessly in the data plane without any additional traffic overhead in the control plane or data plane.

However, if there is a permanent failure on a backup path, the primary link would be left without any failover options and the network would be prone to frequent outages whenever such a primary link goes down, rendering any resiliency mechanisms ineffective. To mitigate such occurrences in our design, we employ the use of our topology monitor, which consistently monitors the state of the data plane after the initial configuration of the fast failover groups. Whenever a link is down for more than a specified threshold, we recompute the backup paths (provided they exist) and incrementally update the fast failover groups for the affected links accordingly. This design choice ensures that our system is robust to dynamic failures and adapts to changing network topologies owing to link failures beyond a timeout threshold. By localizing such failure handling with incremental updates, we also ensure that any valid previous computations for backup paths are fully leveraged and the controller does not have to redo any work immediately.

Our technique for providing transparent and automatic failure resilience through FORTIFY is summarized in Algorithm 1.

Algorithm 1

```
1:  Run topology monitor to get the state of network G=(V,E)
2:  for l_p in E do
3:      Compute backup path set S_p using computeFFRules
4:  end for
5:  while controller is active do
6:      Accept socket connection for router r_k
7:      Create thread toCtrl for r_k via callback handleComm
8:      Create thread toRouter for r_k via callback handleComm
9:  end while
10: procedure handleComm
11:     msg ← parse message received from listening socket
12:     if msg.type == OFPT_FEATURES_REPLY then
13:         Configure FF group in r_k
14:         Configure MPLS tunnels in r_k
15:     else if msg.type == OFPT_FLOW_MOD then
16:         Modify msg.rule to use FF group
17:     else if msg.type == OFPT_MULTIPART_REPLY then
18:         Modify msg.data to hide FF group from controller
19:     end if
20:     Send msg to destination
21: end procedure
```

IV. Implementation

We have implemented our prototype in C++. Our implementation supports controllers running both OpenFlow version 1.0 and 1.3. FORTIFY appears as controller for network devices and connects as data plane devices to the actual controller. Whenever a TCP connection is initiated by a router to connect to the controller on startup, it connects to FORTIFY, which in turn creates a new socket stream to provide a connection with the actual controller and maintains state for both these socket streams. Therefore, in effect we get complete visibility into all messages being exchanged between the data plane and the control plane.

We now describe the different APIs in our current prototype and the implementation choices that make this shim layer transparent and how it integrates with the control and data planes seamlessly.

A. API for Backup Path Computation

To support customizability of backup paths, we support an extensible interface, `computeFFRules`, where a path-finding algorithm is used to find backup paths. By default, `computeFFRules` uses Boykov-Kolmogorov (BK) algorithm [24] to compute edge-disjoint backup paths with resiliency level = 2 (i.e. two backup paths for every link). BK algorithm uses a modified version of maxflow to compute edge-disjoint/node-disjoint paths between any given pair of nodes in the network. However, as described in section III-B, the user can use this interface to specify backup paths for each link failure. Consequently, any policy for specifying backup paths can be implemented in the network through this interface.

B. API for Installing Fast Failover

This API is invoked after communication is established between the controller and any data plane switch across our shim layer. It installs the backup paths computed using `computeFFRules` into the data plane. We use MPLS as an example technology for path tunneling. For every new forwarding rule installed on a switch by the controller, a fast failover group is also installed so that backup paths can be used in case of a link failure. We use MPLS tags to identify backup routes and new rules are installed along the backup paths to support reachability under failures.

C. API for OpenFlow Version Translation

In the current implementation, FORTIFY provides compatibility between controllers operating on OpenFlow 1.0 and data plane devices running on OpenFlow 1.3. We chose to implement the translation module for OpenFlow 1.0 and 1.3 as a proof of concept, but our design does not depend on these versions and can be extended to cater to different versions. The translation module for different versions of OpenFlow as discussed in section III-E is also exposed as an API. It handles all the logic for translating messages from OpenFlow 1.0 to OpenFlow 1.3 and vice versa. The main method that we export as part of this API is `processMessage` which we describe as follows.

All OpenFlow packet translations need appropriate version type to be set in the packet header in order to be interpreted correctly at the destination. However, in addition to this, some OpenFlow message types need complete reorganization in the packet body as well. In our current implementation, we support translation for the OpenFlow message types as summarized in Table I[1].

V. Evaluation

We evaluate our implementation on emulated networks using Mininet over a Ubuntu 16.04 VM with 8 Gigabytes of RAM and a single 2.70 GHz Intel(R) Core(TM) i7-7500U CPU. We run our experiments with two different OpenFlow based SDN controllers: ONOS and Ryu. In the following sections we describe the performance and overhead of FORTIFY. We use edge-disjoint paths, described in section IV-A, as backup paths for each link.

[1]This list is specific to the cases we encountered during our experiments and is certainly not exhaustive

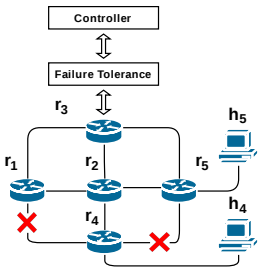


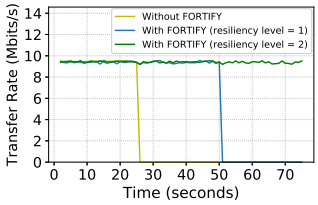
Fig. 5. Experimental setup to test the correctness and performance of FORTIFY with different controllers.

A. Ryu

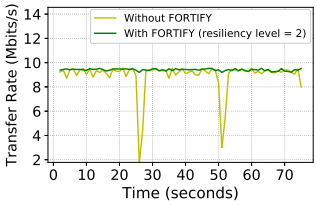
We use Ryu REST router application [25] to demonstrate the performance of FORTIFY on a non-resilient controller. Ryu on its own is not resilient to link failures and is a suitable choice to show and assess the effectiveness of our design.

For our experiments with Ryu, we use the topology in Figure 5. Here, normal communication between h_5 and h_4 occurs through the path $h_5-r_5-r_4-h_4$. The first and second backup paths for link r_5-r_4, as calculated by the edge disjoint algorithm, are $r_5-r_2-r_1-r_4$ and $r_5-r_3-r_2-r_4$ respectively. We measure the throughput between h_5 and h_4 in three different scenarios i) Without FORTIFY ii) With FORTIFY configured for single-link failure and iii) With FORTIFY configured for two link failures. The results can be seen in Figure 6(a).

We observe that without our failover mechanism in place, throughput goes to zero immediately when the link failure occurs. This is because the switch cannot reroute traffic on its own unless rules for failover are installed explicitly. Without a failover mechanism, the switch simply drops all packets if the primary link fails. However, with FORTIFY, we find that the hosts can still communicate with minimal loss in throughput despite the failure of the primary link between them. Moreover, for a link configured to have two backup paths, hosts can communicate even after a link failure in the first backup path. Even when no failure happens, throughput is comparable to that of the original, non-resilient network.



(a) Ryu



(b) ONOS

Fig. 6. Throughput with FORTIFY is comparable to that of the original network and it prevents drops after link failures. First failure at t = *25*, second at t = *50s*

Header modification for version type	Header and body modification	Translation not present but required	Not supported in OpenFlow v1.0
OFPT_HELLO	OFPT_FEATURES_REPLY	OFPT_PORT_MOD	OFPT_GROUP_MOD
OFPT_ERROR	OFPT_PACKET_IN	OFPT_TABLE_MOD	OFPT_SET_ASYNC
OFPT_ECHO_REQUEST	OFPT_PACKET_OUT	OFPT_BARRIER_REQUEST	
OFPT_ECHO_REPLY	OFPT_FLOW_MOD	OFPT_BARRIER_REPLY	
OFPT_FEATURES_REQUEST	OFPT_MULTIPART_REQUEST	OFPT_METER_MOD	
OFPT_PORT_STATUS	OFPT_MULTIPART_REPLY		

TABLE I

TRANSLATION SPECIFICATION FOR OPENFLOW 1.0 AND OPENFLOW 1.3 PACKETS FOR OUR EXPERIMENTS. THE LAST COLUMN REPRESENTS PACKET TYPES THAT WERE INTRODUCED IN LATER OPENFLOW VERSIONS AND HENCE ARE NOT SUPPORTED IN OPENFLOW VERSION 1.0.

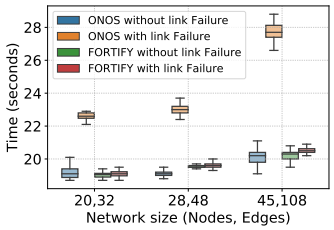


Fig. 7. Transfer completion time of a 20 MB file on Fat Tree topologies of different sizes across 40 measurements.

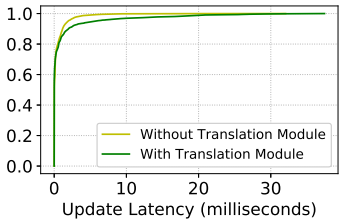


Fig. 8. CDF for OpenFlow message processing times in FORTIFY for $n = 5$ experimental runs on Ryu REST Router.

B. ONOS

In order to measure the generalizability of our technique, we test our implementation on ONOS [26], which is a production-grade control platform with deployments in the real world. We use ONOS's reactive forwarding application to install rules in switches. The ONOS reactive forwarding application is failure resilient. but it is not data plane-based failure resilient. During the time that it takes for the controller application to react to the failure and install new forwarding paths in the affected switch, packets get dropped and flow completion times are affected.

To demonstrate the time incurred by reactive controllers in recovering from link failures, we run throughput measurements (similar to the one in Section V-A) on the topology shown in Figure 5 with and without FORTIFY. For experiments with FORTIFY, we use resiliency level = 2 (i.e. two installed backup paths for link $r_5 - r_4$). We measure the throughput between the two hosts h_5 and h_4. The results can be seen in Figure 6(b). As expected, there was a drop in throughput upon link failure in case of ONOS's reactive forwarding strategy but no such drop was seen with FORTIFY, which had proactively installed the backup paths, thereby providing minimal disruption in the network.

For micro-benchmarking, we use fat tree topologies of different sizes. Figure 7 shows the flow completion time across 40 measurements of sending a 20 MB file between two hosts with and without link failure in between. For runs with link failure, we fail two links that are being used by the flow with a 5 second gap. We observe that failure recovery mechanism of FORTIFY **consistently outperforms** the reactive failure resiliency of ONOS and completes the flow in significantly lesser time. This is because unlike ONOS, FORTIFY proactively installs backup paths and thus does not incur the cost for re-computation of paths and installation of new flow rules upon link failure.

Moreover, we observe that transfer completion time with link failures in ONOS increases with the size of network as can be seen in Figure 7. This increase is due to a higher path re-computation time due to a larger size of the underlying topology. FORTIFY does not incur this cost since it has already configured backup paths in the data plane proactively and allows the switches to **failover immediately** in such an event. We also note that FORTIFY does not significantly increase the flow completion time when there are no link failures. This is because the only overhead of FORTIFY in the data plane is the addition of flow rules and on the fly modification of only two OpenFlow messages, which does not have any major impact on flow completion times.

The preliminary experiments above demonstrate that FORTIFY provides automatic failure resilience without incurring major overheads. Furthermore, with controllers that already have some notion of fault tolerance in the control plane, FORTIFY performs better than their built-in mechanisms. We discuss FORTIFY's overheads in further detail in the next section.

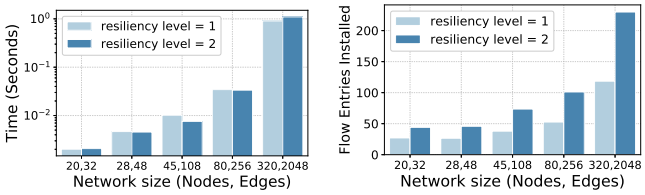
C. Evaluating Backup Path Computation

1) Control Overheads: We now investigate the overhead of our failure tolerance layer. Figure 8 shows the CDFs for message processing latency incurred by FORTIFY from Dynamic Data Plane Mapping as described in section III-D. Here we observe the latency cost incurred by FORTIFY each time a message is transmitted between the control plane and the data plane. From the processing time of all messages, we can see that most of them are processed very quickly since around 95% of the traffic requires less than 2 ms to be completely processed. This can be attributed to the fact that most messages do not require any modifications and are simply allowed to pass through. We also see that operating the controller and network switches in different versions of OpenFlow does not incur significant additional overhead in terms of network performance. Although our failure tolerance layer now conducts one more step in translating the packets between OpenFlow versions, in addition to failover configuration, we can still see that most of them are processed quickly. Around 90% of the packets require less than 2 ms to be completely processed with the translation module in effect.

2) Computational Overhead: We now evaluate the computational overhead of calculating backup paths. This overhead would vary depending on user's implementation of the interface

to calculate backup paths. Here we measure the edge-disjoint backup paths computation time for single-link and two-link failures in data center networks (fat tree topologies) of different sizes as illustrated in Figure 9(a). We observe that the time taken to compute backup paths is in the order of seconds depending on the size of the network and is within an acceptable range in terms of performance overhead. In addition to this, the cost for computation is encountered only during the initial deployment of the network with FORTIFY or whenever an incremental update is needed in the event of a permanent network failure as described in section III-F. We argue that the sparse invocation of the interface to compute backup paths (`computeFFRules`) provides a reasonably low overhead on the overall network performance and resources being consumed in FORTIFY.

3) Data Plane Overhead: Figure 9(b) shows that even for large Fat Tree topologies with high resiliency requirements (i.e. resiliency level of 1 and 2), FORTIFY adds flow rules in the order of hundreds. Modern switches support tens of thousands of flow rules per switch, which is why this cost is justifiably low and does not hamper FORTIFY's deployability.



(a) Time taken to compute edge-disjoint backup paths.

(b) Average number of extra flow entries installed on each switch.

Fig. 9. Overhead measurement experiments performed on data center networks of different sizes using FORTIFY.

VI. Related Work

There has been extensive work done on providing failure resiliency in networks. Older works focused on rerouting mechanisms that involve local decisions by routers for failure resilience. For standard link state routing protocols such as ISIS or OSPF, whenever a failure or repair is detected, the network is flooded with link state packets, reflecting the update in network topology. Following this, each router updates its routes using shortest path algorithm such as the Dijkstra's algorithm. However, there are inherent issues regarding network stability if the routers do all the above for transient changes in the network topology such as a link going up and down several times in a second [6].

Newer work focuses on failure resiliency in the context of SDNs by using failover to automatically switch to redundant or standby paths when there is a link or node failure [27]. For our purposes, we consider the rerouting of packets through an alternative path in case of link or node failure as failover. Since these alternative paths are pre-established before an actual failure, the switches can immediately failover to the backups in the event of failure and therefore this mechanism is referred to Fast Failover [9]. Solutions in this domain include linking external libraries to controllers to provide fault tolerance [9], [19], using designated network programming languages to specify both the functional and failure tolerance requirements [15], [28], and OpenFlow extensions [17], [20], [29] to compute and install feasible sets of paths. These solutions restrict the user to specified programming language or require separate integration into the controller. Our goal is specifically to curb extra work on controller integration and build a layer which is transparent to a controller, in that it requires little to no modification from the vantage point of the controller or a network administrator. More recently, designs for failure resiliency through programmable data planes and P4 [30] have emerged [31]. However, these designs require changes to the hardware of the switches or implementation of the P4 language and cannot easily be used with current networks. We demonstrate that FORTIFY can be used without any modifications to existing controllers.

VII. Discussion and Limitations

We now discuss various design considerations of FORTIFY. First, FORTIFY's current position in the SDN stack means that it needs to be situated in the critical path between the data plane and the control plane. This can be achieved by instantiating FORTIFY on the same machine as the controller or deploying it as a distributed fabric or cloud service to achieve better scalability and resilience of its own (i.e. the FORTIFY layer itself has redundant instances running concurrently). However, it is important to note that irrespective of the deployment mode, FORTIFY does not interfere with existing controller implementations, nor does it require any prior knowledge about the controller's semantics for complete and seamless integration. The proposed design is indeed controller-agnostic and can thus be used to provide failure resilience in existing controllers.

Second, our system is programmable for different failure reaction strategies. For example, a plethora of work has been done in the area of correlated failures and how to mitigate their effect in large IP networks [18], [32], [33]. Since the backup path computation component is customizable, it can be used to compute different possible backup paths for a link according to the user's needs in various capacities, such as to address QoS requirements. The failover paths installed in fast failover groups can be those which are in accordance with such failure reaction strategies. Users can also incorporate their own path selection strategy for choosing backup paths. All this body of work can be easily incorporated into the design of FORTIFY by extending the programmable interface to implement such techniques. However, there is an inherent tension between transparently achieving fault tolerance and having customizable resiliency. Our belief is that having a good set of pre-defined configurations is a good balance between the two. We consider it somewhat similar to how compilers specify optimization levels - it is not really a full blown API, but still allows some degree of control.

Third, our approach is agnostic to being a layer beneath the controller or a module in the controller. The key idea of FORTIFY is to handle fault-tolerance automatically - without the need for explicit setup by the application developer. The current choice of situating failure resiliency in a shim layer is made to allow existing controllers to work with FORTIFY without modifications. The fact that this was possible to do further shows that failure resiliency is orthogonal to the rest of the controller logic and can be addressed separately. Such a layered approach to abstract complexity into another layer has been used previously in SDNs [34] and our work takes a first step by evaluating the pros and cons of incorporating resilience as a first class primitive in the controller.

VIII. Conclusion

In this paper, we the discuss the architecture and implementation of FORTIFY, a fault tolerance shim layer in the SDN stack that provides fast, controller agnostic data plane fault tolerance in SDN environments. FORTIFY leverages Fast Failover group feature introduced in OpenFlow 1.3 to automatically and transparently handle multi-link failures without complicating the control plane. We provide a simple interface to the user for customizing backup paths and provide different path computation algorithms to help in the implementation of the interface. Our work also provides a translation unit to make controller applications running different versions of OpenFlow seamlessly compatible with the data plane, provided data plane components run OpenFlow versions that have Fast Failover group functionality available. We demonstrate that our technique is sufficient for fulfilling functional objectives of making a data plane based failure resilient network from non-resilient controller applications and can be easily extended to provide QoS guarantees as needed. Our experiments demonstrate that our design has low performance overhead and can be easily deployed in existing production environments with minimal disruption.

References

[1] "The new network: SDN for financial services," [Online; accessed 25-Feb-2020]. [Online]. Available: https://sdn.cioreview.com/cxoinsight/the-new-network-sdn-for-financial-services-nid-23289-cid-147.html

[2] V. Gkioulos, H. Gunleifsen, and G. K. Weldehawaryat, "A systematic literature review on military software defined networks," *Future Internet*, vol. 10, no. 9, p. 88, 2018.

[3] "SDN: Powering the next generation of healthcare networks," 2019, [Online; accessed 25-Feb-2020]. [Online]. Available: https://cbcommunity.comcast.com/community/browse-all/details/sdn-powering-the-next-generation-of-healthcare-networks

[4] The Washington Post, "Russian military was behind 'notpetya' cyberattack in ukraine, cia concludes," [Online; accessed 26-Feb-2020]. [Online]. Available: https://tinyurl.com/y9gph48o

[5] "Homeland security creates anti-hacking center to protect industries," [Online; accessed 26-Feb-2020]. [Online]. Available: https://www.cnet.com/news/homeland-security-creates-center-to-protect-industries-from-hacks/

[6] G. Iannaccone, C.-N. Chuah, S. Bhattacharyya, and C. Diot, "Feasibility of ip restoration in a tier 1 backbone," *IEEE Network*, vol. 18, no. 2, pp. 13–19, 2004.

[7] L. Sahasrabuddhe, S. Ramamurthy, and B. Mukherjee, "Fault management in ip-over-wdm networks: Wdm protection versus ip restoration," *IEEE journal on selected areas in communications*, vol. 20, no. 1, pp. 21–33, 2002.

[8] A. Fumagalli and L. Valcarenghi, "Ip restoration vs. wdm protection: Is there an optimal choice?" *IEEE network*, vol. 14, no. 6, pp. 34–41, 2000.

[9] Y.-D. Lin, H.-Y. Teng, C.-R. Hsu, C.-C. Liao, and Y.-C. Lai, "Fast failover and switchover for link failures and congestion in software defined networks," in *2016 IEEE International Conference on Communications (ICC)*. IEEE, 2016, pp. 1–6.

[10] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling fast failure recovery in openflow networks," in *2011 8th International Workshop on the Design of Reliable Communication Networks (DRCN)*. IEEE, 2011, pp. 164–171.

[11] D. Staessens, S. Sharma, D. Colle, M. Pickavet, and P. Demeester, "Software defined networking: Meeting carrier grade requirements," in *2011 18th IEEE Workshop on Local Metropolitan Area Networks (LANMAN)*, 2011, pp. 1–6.

[12] "Amazon found every 100ms of latency cost them 1% in sales," [Online; accessed 11-Dec-2020]. [Online]. Available: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/

[13] O. N. Foundation, "Openflow switch specification version 1.0.0," 2009. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf

[14] "Mpls traffic engineering fast reroute — link protection," [Online; accessed 11-Dec-2020]. [Online]. Available: https://www.cisco.com/en/US/docs/ios/12_0st/12_0st10/feature/guide/fastrout.html#wp1015327

[15] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "Fattire: Declarative fault tolerance for software-defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, pp. 109–114.

[16] C. Hannon, D. Jin, C. Chen, and J. Wang, "Ultimate forwarding resilience in openflow networks," in *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, 2017, pp. 59–64.

[17] A. Capone, C. Cascone, A. Q. Nguyen, and B. Sanso, "Detour planning for fast and reliable failure recovery in sdn with openstate," in *2015 11th international conference on the design of reliable communication networks (DRCN)*. IEEE, 2015, pp. 25–32.

[18] A. Capone, C. Cascone, A. Q. T. Nguyen, and B. Sansò, "Detour planning for fast and reliable failure recovery in sdn with openstate," in *2015 11th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2015, pp. 25–32.

[19] N. E. Petroulakis, G. Spanoudakis, and I. G. Askoxylakis, "Fault tolerance using an sdn pattern framework," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–6.

[20] C. Cascone, L. Pollini, D. Sanvito, A. Capone, and B. Sansó, "Spider: Fault resilient sdn pipeline with recovery delay guarantees," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, 2016, pp. 296–302.

[21] S. Prabhu, M. Dong, T. Meng, P. B. Godfrey, and M. Caesar, "Let me rephrase that: Transparent optimization in sdns," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 41–47. [Online]. Available: https://doi.org/10.1145/3050220.3050226

[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[23] O. N. Foundation, "Openflow switch specification version 1.3.0," 2012. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf

[24] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 9, pp. 1124–1137, 2004.

[25] S. Ryu, "Framework community: Ryu sdn framework," *Online. http://osrg. github. io/ryu*, 2015.

[26] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "Onos: Towards an open, distributed sdn os," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620744

[27] K. Foerster, A. Kamisinski, Y. Pignolet, S. Schmid, and G. Tredan, "Bonsai: Efficient fast failover routing using small arborescences," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 276–288.

[28] H. Li, Q. Li, Y. Jiang, T. Zhang, and L. Wang, "A declarative failure recovery system in software defined networks," in *2016 IEEE International Conference on Communications (ICC)*, 2016, pp. 1–6.

[29] R. M. Ramos, M. Martinello, and C. Esteve Rothenberg, "Slickflow: Resilient source routing in data center networks unlocked by openflow," in *38th Annual IEEE Conference on Local Computer Networks*, 2013, pp. 606–613.

[30] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[31] R. Sedar, M. Borokhovich, M. Chiesa, G. Antichi, and S. Schmid, "Supporting emerging applications with low-latency failover in p4," in *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*, ser. NEAT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 52–57. [Online]. Available: https://doi.org/10.1145/3229574.3229580

[32] H. Saito, Y. Miyao, and M. Yoshida, "Traffic engineering using multiple multipoint-to-point lsps," in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, vol. 2. IEEE, 2000, pp. 894–901.

[33] V. Sharma and F. Hellstrand, "Framework for multi-protocol label switching (mpls)-based recovery," 2003.

[34] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," 01 2009.