Destroying networks for fun (and profit)

Nick Shelly*,§ Brendan Tschaen,† Klaus-Tycho Förster,* Michael Chang,‡
Theophilus Benson,† Laurent Vanbever*
*ETH Zürich, ‡Princeton University, †Duke University, §Forward Networks

ABSTRACT

Network failures are inevitable. Interfaces go down, devices crash and resources become exhausted. It is the responsibility of the control software to provide reliable services on top of unreliable components and unpredictable events. Guaranteeing the correctness of the controller under all types of failures is therefore essential for network operations. Yet, this is also an almost impossible task due to the complexity of the control software, the underlying network, and the lack of precision in simulation tools.

Instead, we argue that testing network control software should follow in the footsteps of large scale distributed systems, such as those of Netflix or Google, which deliberately induce live failures in their production environments during working hours, and analyze how their control software reacts.

In this paper, we describe Armageddon, a framework for introducing sustainable and systematic chaos in networks. While we do cause failures, we do so without violating some operator-specified network invariants (*e.g.*, end-to-end connectivity). The injected failures also guarantee some notion of coverage. If the controller can sustain all of them, then it can be considered resilient with a high degree of confidence. We describe efficient algorithms to compute failure scenarios and implemented them in a prototype. Applied to real-world networks, our algorithms a coverage of 80% of the links within only three iterations of failures.

1. INTRODUCTION

"The best way to avoid failure is to fail constantly"

—John Ciancutti, Netflix

Upon migrating their platform to the cloud in late 2010, Netflix caused a sensation by announcing that their initial suite of systems included a daemon whose sole purpose was to randomly kill instances and services within their *production* environment [4]. The daemon is known internally as the "Chaos Monkey" (CM), a reference to the act of unleashing a wild, weaponized animal in their Data Centers (DC). Over the years, Netflix amplified the amount of "chaos" they introduced in their infrastructure, *e.g.* by killing entire availability zone (Chaos Gorilla) [7]. In 2012 alone, Netflix CM killed over 65,000 software instances. While most failures go unnoticed, introducing chaos helped Netflix engineers to uncover numerous bugs, as well as isolate and fix them [6, 5]. Other cloud providers reported running a CM of their own. Google has DiRT (Disaster Recovery Testing) [15], Amazon has GameDay [37], and Microsoft Azure has a "Search Chaos Monkey" [2]. Ironically, Amazon EC2 even recommends its customers to run a CM internally to shield themselves from cloud failures [1].

Similarly to cloud platforms, failures are guaranteed to appear in Software-Defined Networks (SDNs) such as those underlaying a DC or a Wide-Area Network (WAN) [17, 34, 19]. For example, large DCs can see up to 5 devices and up to 40 links failures *per-day*, on average [17]. Actually, we even expect this number to keep growing as more and more networks starts to rely on low-cost, commodity equipments.

In such unreliable environments, relying on correct, fault-tolerant control software is a must. Unfortunately, guaranteeing their correctness under any possible failure is almost impossible (§2). Many common "white-box" approaches to detecting and eliminating bugs, such as model checking, symbolic execution or unit testing, have proven difficult on large scale systems, including those involving SDN controllers. As an illustration, two popular SDN controllers, OpenDayLight [9] and ONOS [8], account for more than 220,000 lines of code (and growing!) scattered in more than 2000 files. Bugs are bound to exist in software engineering projects of such magnitude. While helpful, running the control software in virtual environments such as Mininet [27] only detects a subset of the possible errors as the behavior of real devices often differs substantially from idealized virtual ones. Many bugs (*e.g.*, race conditions) only manifest once deployed in production settings, *i.e.,* when running on real hardware [25, 24].

In this paper, we argue network operators should include "black box" testing by systematically inducing failures into the *production environment* to uncover bugs. Intuitively, purposely inducing failures in production may appear counter-productive and even prevent networks from meeting Service Level Agreements (SLA). However, failing network components does not necessarily mean creating disruptions. Networks are (usually) composed of highly redundant physical and software components. In theory, one should be able to fail redundant parts of the networks with little to no observable effects. In practice, if failing redundant resources *does* create disruption, operators and developers must be informed so that immediate remedial actions can be taken. Indeed, it is better to deal with disruptions during normal working hours, when engineers are available to correct the issues, rather than to allow the issue to manifest (inevitably) at unpredictable hours.

Randomly killing network resources is trivial, but introducing *useful and systematic* chaos is challenging. First, the induced failures must be realistic, meaning that the probability of them happening should be above a given threshold. At the same time, the induced failures should enable the system to recover, *i.e.* they should be reasonable. For instance, the combined failure of all links of a network, although theoretically possible, is: *i)* too unlikely; and *ii)* useless to test as the network ceases to exist. Likewise, failing all redundant resources, while preserving connectivity, could cause unacceptable congestion. In short, operational requirements, like uptime and security or the fact that sensitive components must remain isolated, must be taken into account when computing failures. Second, the induced failures should be such that they guarantee some notion of coverage, for instance, by exercising all failure combinations or by failing each resource at least once. Third, inducing failures should be performed quickly to minimize instabilities: less failures is better. To do so, we aim at minimizing the number of failures rounds or "iterations". For instance, we minimize the number of iterations required to fail every link at least once.

We take the first step toward inducing systematic chaos in networks and introduce *Armageddon*, a Chaos Monkey for SDNs[1]. Armageddon automatically computes failure scenarios that preserve specified network-wide invariants (*e.g.*, end-to-end connectivity) and guarantees some notion of coverage. Failure scenarios can target both the control- and the data-plane. Once computed, Armageddon induces the failures in the network while monitoring it. If a problem is discovered (*e.g.*, loss of reachability while the graph is still connected), Armageddon immediately brings back the failed resources and reports the precise failure scenario to the operator/developer. In a sense, Armageddon automatically learns unit tests.

[1]While we focus on SDN, Armageddon principles can be applied to any kind of network.

In summary, we make the following contributions:

Armageddon framework. We describe the design as well as a prototype implementation of Armageddon: a controller-agnostic Chaos Monkey for SDNs. Armageddon efficiently computes reasonable failure scenarios and induces them in production while monitoring correctness. Upon detecting violations, Armageddon automatically generates detailed reports (§3).

Algorithms. Computing *what* to fail and *when* to fail turned out to be a challenging algorithmic problem. We describe and prove efficient algorithms to compute failure scenarios while guaranteeing any-to-any connectivity and complete coverage (§4).

Evaluation. Using representative network topologies, we show that Armageddon achieves maximum failure coverage in few iterations (§5).

We start by describing the shortcomings of current verification techniques. We then present Armageddon and how we can incorporate it into production networks. In this paper, we focus on *preserving reachability* upon *failures*. Exploring other types of failures (*e.g.*, introducing delays, losses, software failures) and invariants (*e.g.*, preserving capacity or waypoint services) is part of our ongoing work.

2. MOTIVATION & BACKGROUND

SDN control planes are complex beasts with intricate functionalities to handle a spectrum of dynamic network events; *e.g.* link failures, software failures, or changes in network policies. In few years, many techniques have been designed to detect and diagnose SDN control plane bugs. However, these techniques are often limited in the extent to which they can exercise code paths representative of large scale productions networks. Next, we examine these techniques and discuss their limitations.

Modeling and simulating networks. Many bugs have been discovered in modern SDN control planes using a combination of model checking [14], symbolic execution [14], and fuzz testing [32]. Unfortunately, these techniques are limited to the properties that can be safely modeled, simulated, or emulated within their frameworks. For example, NICE [14] does not model the complex switch behaviors that can lead to concurrency bugs [28, 29] or that violate the OpenFlow [26] specification, significantly limiting the set of control plane bugs that can be explored. Similarly, STS [32] does not model interactions with realistic traffic matrices and is thus unable to quantify performance related bugs.

Checking invariants. Reactive approaches [20, 22, 35, 10] aim at detecting bugs in real time by examining the control plane output or by exercising the data plane. While these techniques can detect realistic bugs, they detect them when they have already occurred. Instead of waiting for bugs to eventually happen, Armageddon actively uncovers them when network operators are available to debug and resolve the issues.

Detecting and recovering from failures. To combat network failures and disasters, techniques have been developed ranging from proactively testing virtual substrates of the production network [13] to developing efficient recovery techniques for general router bugs [21] and specific network failures *e.g.* device/link failures [38, 30]. Armageddon also tests production networks but utilizes the whole network and includes algorithms to ensure that failures have a minimal impact on production traffic. This eliminates the need for virtualization and simplifies the data plane. Automated failure recovery techniques remain useful and crucial, but Armageddon ensures that failures, or rather disasters, occur when operators are available, thus allowing them to monitor the process of these failure detection tools.

Destroying SDNs in the industry. BigSwitch, one of the main players in the SDN industry, recently announced a "Chaos Monkey Style" stress testing product [3]. While BigSwitch's CM randomly fails resources, it does not aim to guarantee coverage or even preserve connectivity. This product, however, does confirm the industrial interest for a system like Armageddon.

3. ARCHITECTURE

Armageddon introduces failures in networks in three consecutive stages (Fig. 1) and using two inputs: an updated view of the network topology and a set of network-wide invariants to be maintained. The former is collected automatically from the network messages. The latter is given by the network operator according to the policies implemented by the controller. Examples of invariants include: preserving end-to-end connectivity, maintaining at least $x\%$ of the capacity, or preventing congestion from appearing.

Using the topology and the invariants, Armageddon computes a set of failure scenarios. Each failure scenario represents a set of network resources (*e.g.*, links) to fail simultaneously while preserving the invariants and maximizing coverage. Armageddon then schedules the failures for execution while monitoring the network. Whenever a violation of a network-wide invariant is discovered, Armageddon reports it back to the developer along with the precise failure scenarios.

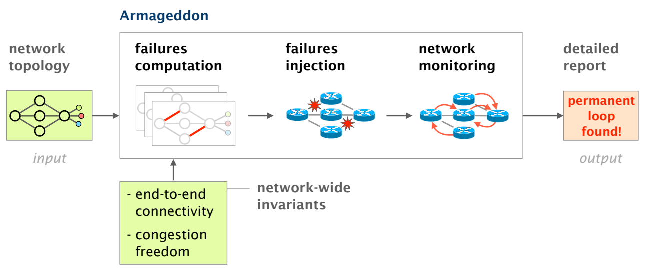

Figure 1: Armageddon workflow

Armageddon can introduce both control- and data-plane failures. At the data-plane level, possible failure scenarios include: failing forwarding resources (*e.g.*, ports, devices), increasing link delays and losses or pretending resource exhaustion. At the control-plane level, failure scenarios include: failing controller replicas, or sending random drop or erroneous commands.

We now describe how Armageddon executes and monitors the network. Then in §4, we describe how Armageddon computes failures, maintaining invariants and maximizing coverage.

Inducing failures. Armageddon must be able to induce diverse classes of active (injection) and passive (withholding messages) failures anywhere in the network as well as precisely tracking the network state. To do so, Armageddon interposes itself between the SDN controller and the physical network as a hypervisor. As such, the hypervisor can intercept all the messages (typically OpenFlow messages) exchanged between the controller and the switches as well as spawn new ones.

Using a hypervisor-based approach provides Armageddon two distinct advantages. First, it enables Armageddon to be controller-agnostic. Second, it gives the controller the "illusion" of failure without actually failing anything in the physical network. For example, the hypervisor can take a switch's port down and spawn an `OFPT_PORT_DOWN` message for the controller. At this point, the controller operates as if the port has gone down, and the hypervisor drops any messages to/from this port on the switch. Moreover, The hypervisor ensures that this port is not used by modifying `FLOOD` or `BROADCAST` actions to skip these downed ports.

We have implemented a working prototype of the hypervisor on top of OpenVirteX [12]. Our prototype tracks the network state, schedules network failures and monitors the network for actual network failures. It also provides an interface to the network administrator that serves two primary functions. Firstly, the administrator can specify extra failures scenarios and schedule parameters. Secondly, after the failure has been injected, the interface is used to convey information from the Invariant Checker (see below), informing the administrator of any undesirable changes in the network state.

Monitoring for correctness. Once the faults have been injected into the network, Armageddon needs to verify that the network behaves properly. To do so, Armageddon introduces an Invariant Checker that communicates with the hypervisor module. Whenever a change in the network state is detected, the Invariant Checker verifies that all network invariants hold. We have implemented the Invariant Checker using NetPlumber [20].

Generating reports and restoring correctness. As soon as a network-wide invariant is violated, Armageddon reports back: *i)* the exact failure scenario that created the issue; *ii)* a concise description of the issue (*e.g.*, forwarding loop for port 80 between s_1 and s_2) as well as; *iii)* the complete trace of network events. This information enables the network operators and/or application developers to go back through the failure scenario and determine what was the cause of the issue and, hopefully, fix the problem. Internally, Armageddon automatically accumulates the failed scenarios as unit tests to be checked again later on. Doing so, Armageddon automatically "learns" effective unit tests.

After generating the report, Armageddon brings backs the failed resources in an attempt to restore violated invariants. However, it might be that, even after having brought back the network resources, the invariant is still violated. This is the case if the controller crashed for instance. While being obviously bad for the network in the short-term, we think that Armageddon crashing the system is actually a *desirable feature* in the long run. Indeed, the exact same failure could have just well happened outside of office hours when few (or no) engineers are around to deal with the problem.

Dealing with concurrent physical failures. Armageddon immediately brings back all failed resources whenever an *actual* network outage is detected. Indeed, as Armageddon reduces the amount of redundancy, actual failures can lead to large disruptions. Similarly, Armageddon never induces failures if some physical resources have failed. Armageddon simply remains passive in order to not stress an already weakened network.

4. COMPUTING SMART FAILURES

In this section, we look at how Armageddon computes failure scenarios that maintain network-wide invariants. Such invariants can be any property that the control-plane ought to maintain upon failures, e.g., congestion freedom. In this short paper, we focus on end-to-end reachability. Reachability is indeed the most fundamental property that any control software must maintain. If a link fails, the control software should restore reachability provided the physical graph is still connected. Armageddon computes failure scenarios that optimize two other objectives besides reachability: *coverage* and *speed*. In short, it aims at failing each link at least once, in as few iterations as possible.

Coverage. While distributed protocols are guaranteed to maintain reachability as long as the network is connected, SDN controllers do not. As such, we want to check if the controller can handle the failure of every single link.

A naïve algorithm would be to perform the following test for every link e. First, check if the network N is still connected without e. Second, if yes, fail the physical link e and see if routing is still possible between all nodes in the SDN; if not, inform the network operator that a single link of failure is in the network.

Coverage + Speed. While the runtime of this naïve algorithm is polynomial, its number of iterations is not acceptable in practice with large networks containing thousands of edges. In the best case, even huge networks should only need a small amount of iterations in average for all testable edges.

As such, Armageddon aims at solving the CONNECTIVITY TESTING problem:

PROBLEM 1 (CONNECTIVITY TESTING). *Find the minimum number of iterations k, where each k_i is a set of failed edges, that are needed to fail every edge at least once while still maintaining network connectivity?*

Finding a solution for $k = |E|$ is easy (cf. the naïve algorithm), but minimizing k turns out to be an algorithmically challenging problem. Theorem 1 proves that optimally solving CONNECTIVITY TESTING requires at least 2 iterations, in the best case; and as many iterations as nodes in the network, in the worst-case.

THEOREM 1. *Let $N = (V, E)$ be a connected network where at least one edge can be failed while maintaining connectivity.* CONNECTIVITY TESTING *needs at least* 2 *and at most $|V| = n$ iterations. These bounds are sharp.*

PROOF. We start with the lower bound: If $e = (u, v)$ can be removed, then e is part of at least one 2-connected component in N. Then, $N' = (V, E \setminus E')$ is still connected, i.e., there is a path P from v to u in N' that joined with e yields a cycle, i.e., one iteration never suffices. For networks where 2 iterations suffice, consider a clique with at least 4 nodes: First, remove a spanning tree T, and second, remove all edges except for T.

We now prove the upper bound. CONNECTIVITY TESTING for a ring of n nodes needs exactly n iterations. To show that no graph needs more than n iterations, observe that after failing a spanning tree, at most $n - 1$ edges can be left to fail. As any of these $n - 1$ edges will be part of a cycle (else they could not be failed), at least one edge can be failed per iteration, resulting in a sharp upper bound. $\square$

```
1: $compute\_link\_failures\_scenarios(N = (V, E))$
2: if $N$ has spanning trees $T_1 = (V, E_1), T_2 = (V, E_2)$, $E_1 \cap E_2 = \emptyset$ then
3:     $fail$ $E \setminus E_1$ and $fail$ $E \setminus E_2$
4: else
5:     $\forall e \in E$ set link weights of $c(e) = 1$
6:     repeat
7:         compute minimum weight spanning tree (MST) $T = (V, E')$
8:         $fail$ all links $E'' = E \setminus E'$ not in the MST $T$
9:         set sum of new edge failures $\lambda = \sum_{\forall e \in E''} c(e)$
10:        $\forall e \in E''$ set $c(e) = 0$
11:    until $\lambda = 0$ or $\forall e \in E : c(e) = 0$
12: end if
```

Algorithm 1: Greedy Killer algorithm. It fails all edges in 2 iterations if N contains at least 2 disjoint spanning trees. Else, at most $|V|$ iterations are needed.

Algorithms. Observe that networks for which all edges can be failed in 2 iterations are characterized by every iteration containing a spanning tree. With this in mind, we propose an algorithm, Greedy Killer (see Algorithm 1), for solving Connectivity Testing which first checks for two edge-disjoint spanning trees and, in the negative case, proceeds to fail all edges that can be failed in multiple iterations.

Lemma 1. Greedy Killer *will fail all edges that can be failed and mark all the others with weight 1.*

Proof. Greedy Killer will never fail an edge that cannot be failed, as it always leaves a spanning tree.

Assume that there is an edge $e = (u, v)$ that can be failed (and thus part of a cycle C), but Greedy Killer will not fail e. Now, consider the network N after Greedy Killer has finished. If there are edges (e.g., e) in C with weight 1, then there is a spanning tree with weight W that will fail at least one of these edges e' that includes all edges from C except e'. Any spanning tree with weight W or less will fail at least one edge, as the weight W is less than the sum of all edge weights in the network. Thus, Greedy Killer would not have been finished, leading to a contradiction. □

Identifying all bridges (*i.e.*, edges whose removal disconnect the network) can also be performed by other algorithms, e.g., [36], but Greedy Killer will mark them as well, allowing us to inform the network operator about all single points of failure in the topology.

Lemma 2. *In each iteration,* Greedy Killer *fails the maximum amount of edges that have not failed yet.*

Proof. Consider any fixed iteration. Let E' be the set of edges e' which either cannot be failed or have been failed in a previous iteration. Let $N' = (V, E')$ and let $C_1, \dots, C_k$ be the connected components of N' that are maximal. Any spanning tree (which maintains connectivity) for N will thus need to contain $k-1$ edges from of individual weight 1 to connect $C_1, \dots, C_k$ (which are individually connected by edges in E'), i.e., no MST can have a weight of less than $k-1$. As the weight of any MST will be exactly $k-1$, Lemma 2 holds. □

We can now prove that Greedy Killer performs according to specification:

Theorem 2. Greedy Killer *is correct.*

Proof. Combining the observation on edge-disjoint spanning trees and Lemma 1 leaves only one open question: if Greedy Killer always finishes after at most n iterations. Note that after one iteration, there are at most $(n-1)$ edges still left to fail, as the first iteration failed all edges not contained in a spanning tree. As the algorithm will fail all failable edges eventually (cf. Lemma 1), it cannot take more than n iterations: Then, there would be an iteration where no edge is failed, yielding a contradiction to Lemma 2. □

Running time. Checking if a network has two edge-disjunct spanning trees can be done in $O(n^2)$ time [31]. Kruskal's MST algorithm takes $O(m \log n)$ time [16]. As such, Greedy Killer runtime is $O(nm \log n)$.

5. PRELIMINARY EVALUATION

In this section, we evaluate the efficiency of our algorithm in minimizing the number of iterations required to completely test a network and how much it differs from the optimal solution.

Datasets. We evaluate Greedy Killer using 261 topologies extracted from the Internet Topology Zoo [23] and 7 RocketFuel topologies [33]. Topologies range from large ISP networks to medium enterprise campuses.

Armageddon assumes that the network resources are redundant. In most networks, this is a given. In some though, single point of failures exist. We remove them before applying Greedy Killer to ensure meaningful results. We performed four pre-processing steps: *i)* we converted the network graphs into undirected graphs; *ii)* removed nodes from 244 networks with only one edge and thus cannot be failed (median of 27% of nodes); *iii)* discarded 14 networks with fewer than 3 edges, and finally *iv)* partitioned 135 of the graphs by removing the edges whose disruption disconnected the graph (median of 10% of edges) and, doing so, created 146 new topologies. After pre-processing, we ended up with 265 networks with a 25th-percentile, median and maximum size of 23, 36 and 972.

Greedy Killer is efficient in practice. In most networks, Greedy Killer fails each link in 5 iterations or less. 78% of the networks can be failed entirely in six iterations, 91% in 8 iterations (see Figure 2). Networks

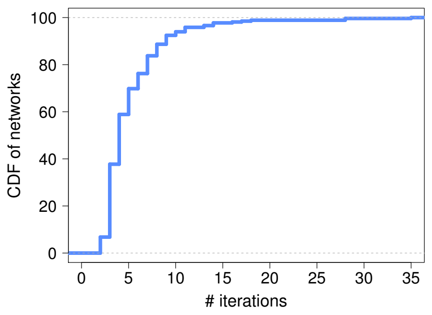

Figure 2: In 60% of the networks, max. 5 iterations are required to fail each edge at least once. Higher values indicate networks with less redundancy (*e.g.*, ring-shaped).

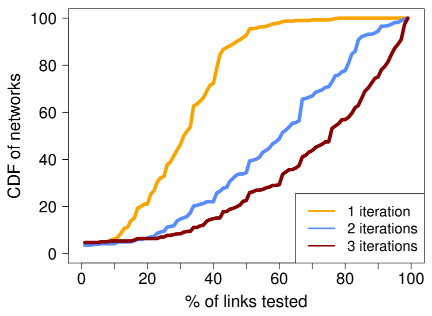

Figure 3: In most networks, only 2 (resp. 3) iterations are required to fail 60% (resp. 80%) of the edges.

that require over 5 iterations are less redundant. Indeed, we observe that these network showed some kind of ring shape, and thus few edges could be tested with each iteration. As an illustration, VTL WaveNet 2008 network required 28 iterations as there were 88 nodes and 92 edges. Thus, a minimum of 87 edges were required at all times to maintain connectivity.

Since most networks are highly redundant, only a few iterations are required to test all links. In Figure 3, we observe that for most networks, over 80% of the links can be tested in 3 iterations. As described in Section 4, maintaining connectivity is the most basic invariant that must be satisfied. Other invariants will increase the number of iterations required. A quantitative evaluation of this increase is part of our ongoing work.

Greedy Killer is optimal most of the time. We now compare the number of iterations computed by GREEDY KILLER with the optimal solution, *i.e.* the smallest number of iterations to fail all possible links. In a network $N' = (V', E')$ with $|V'| = n'$ and $|E'| = m'$, at least $n' - 1$ edges always need to stay in N' to maintain reachability. Therefore, at most $m' - (n' - 1)$ edges can be failed in each iteration, yielding a lower bound of $\lceil m'/(m' - n' + 1) \rceil$ iterations.

We find that GREEDY KILLER produces the optimal number of iterations for at least 138 of the 265 networks (52%). An open question we are working on is to find an optimal algorithm to succeed in the minimum amount of iterations possible.

6. DISCUSSIONS AND FUTURE WORK

Currently, GREEDY KILLER aims to fail as many resources as possible while maintaining network connectivity. This allows Armageddon to explore important properties related to reachability. As part of ongoing work, we plan to extend GREEDY KILLER to include the following properties:

Stronger coverage properties. Focusing on connectivity reduces our coverage as we might miss bugs, for instance, the ones that appear upon a precise sequence of failures. We plan to improve Armageddon's coverage in our ongoing work by applying stateless model checking to introduce more non-determinism like order. More over, we also plan to leverage domain knowledge about the network topology and current (or expected) traffic patterns to reduce the number of failure scenarios explored to a small but representative number.

Preserving more advanced properties (*e.g.*, network capacity, congestion-freedom). Although connectivity is preserved the failures may not preserve more advanced properties, for instance, failing entire spanning trees can decrease network capacity so much that congestion starts to appear. Similarly, security policies can also legitimately prevent reachability in connected graph.

Fortunately, it is easy to combine GREEDY KILLER with other network-wide invariants. For example, to test the congestion-free properties of a SWAN-like SDN controller [18], we can first monitor the current traffic to estimate the demands. Then, using Multi-Commodity Flow formulations [11], we can maximize the set of not yet failed edges in each iteration while maintaining sufficient network capacity.

7. CONCLUSIONS

Network failures should be embraced, not avoided. By systematically destroying redundant parts of the network while monitoring it, Armageddon makes sure that the control software does its job correctly. At its core, Armageddon is based on an efficient algorithm, GREEDY KILLER, which quickly computes failure scenarios that maintain connectivity. Our results on representative topologies indicate that Armageddon is quickly able to fail each link at least once. Overall, we expect our approach to foster a new breed of "destroying" tools for networks.

8. REFERENCES

[1] Amazon AWS Official Blog. EC2 Maintenance Update. `https://aws.amazon.com/blogs/aws/ec2-maintenance-update-2/`.

[2] Azure's Search Chaos Monkey is wreaking havoc to find potential points of failure. `http://bit.ly/1HPLtQ9`.

[3] Big Switch Networks. Chaos Monkey and Big Cloud Fabric. `http://bit.ly/1RDxYaO`.

[4] NetFlix. 5 Lessons We've Learned Using AWS. `http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html`.

[5] NetFlix. Can Spark Streaming survive Chaos Monkey? `http://techblog.netflix.com/2015/03/can-spark-streaming-survive-chaos-monkey.html`.

[6] NetFlix. Chaos Monkey Released Into The Wild. `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`.

[7] NetFlix. Simian Army. GitHub repository. `https://github.com/Netflix/SimianArmy`.

[8] ONOS Controller Platform. `http://onosproject.org/`.

[9] OpenDaylight Controller Platform. `http://www.opendaylight.org/`.

[10] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN Traceroute: Tracing SDN Forwarding Without Changing Network Behavior. In *ACM SOSR*, Santa Clara, CA, USA, Jun 2015.

[11] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows - theory, algorithms and applications.* Prentice Hall, 1993.

[12] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, W. Snow, and G. Parulkar. OpenVirteX: A network hypervisor. *Open Networking Summit*, 2014.

[13] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 111–122, New York, NY, USA, 2008. ACM.

[14] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford, et al. A NICE Way to Test OpenFlow Applications. In *NSDI*, volume 12, pages 127–140, 2012.

[15] T. Claburn. Google Vs. Zombies – And Worse. InformationWeek - Network Computing, 2013. `http://ubm.io/1ftfjxA`.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

[17] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 350–361, New York, NY, USA, 2011. ACM.

[18] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In D. M. Chiu, J. Wang, P. Barford, and S. Seshan, editors, *ACM SIGCOMM*, pages 15–26. ACM, 2013.

[19] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software DeïňĄned WAN. In *ACM SIGCOMM*, 2013.

[20] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*, pages 99–111, 2013.

[21] E. Keller, M. Yu, M. Caesar, and J. Rexford. Virtually eliminating router bugs. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 13–24, New York, NY, USA, 2009. ACM.

[22] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: verifying network-wide invariants in real time. SIGCOMM '12, pages 467–472, 2012.

[23] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[24] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT Way for Openflow Switch Interoperability Testing. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 265–276, New York, NY, USA, 2012. ACM.

[25] M. Kuzniar, P. Peresini, and D. Kostić. Providing Reliable FIB Update Acknowledgments in SDN. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 415–422, New York, NY, USA, 2014. ACM.

[26] M. Kuźniar, P. Perešíni, M. Canini, D. Venzano, and D. Kostić. A SOFT Way for OpenFlow Switch Interoperability Testing. In *Proceedings of ACM CoNEXT'12*, Dec 2012.

[27] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[28] Miserez, J. Bielik, P. El-Hassany, A. Vanbever, L. Vechev, and Martin. SDNRacer: Detecting Concurrency Violations in Software-Defined Networks. In *ACM SOSR*, Santa Clara, CA, USA, Jun 2015.

[29] P. Perešíni, M. Kuźniar, N. Vasić, M. Canini, and D. Kostić. OF.CPP: Consistent Packet Processing for OpenFlow. In *Proceedings of HotSDN'13*, Aug 2013.

[30] 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*, HotSDN '13, pages 109–114, New York, NY, USA, 2013. ACM.

[31] J. Roskind and R. E. Tarjan. A note on finding minimum-cost edge-disjoint spanning trees. *Mathematics of Operations Research*, 10(4):701–708, 1985.

[32] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox sdn control software with minimal causal sequences. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 395–406, New York, NY, USA, 2014. ACM.

[33] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, Feb. 2004.

[34] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A Network-state Management Service. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 563–574, New York, NY, USA, 2014. ACM.

[35] P. Tammana, R. Agarwal, and M. Lee. CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In *ACM SOSR*, Santa Clara, CA, USA, Jun 2015.

[36] R. Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, 2(6):160 – 161, 1974.

[37] A. Tseitlin. The antifragile organization. *Commun. ACM*, 56(8):40–44, Aug. 2013.

[38] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. In *ACM SIGCOMM 2012*, SIGCOMM '12, pages 419–430, New York, NY, USA, 2012. ACM.