Measuring Resiliency of System of Systems using Chaos Engineering Experiments

Thomas Bailey, Patrick Marchione, Pete Swartz, Raed Salih, Michael R. Clark, Robert Denz
Riverside Research, 2640 Hibiscus Way, Beavercreek, OH, USA 45431

ABSTRACT

Chaos Engineering (CE), which Netflix introduced in 2008, is used by researchers to assess and find weaknesses in system resiliency. Such weaknesses can arise, when subsystems are individually robust, but that robustness disappears when multiple subsystems are paired together in a System of Systems (SoS). CE researchers develops methods and metrics for finding such fragilities. In this paper, we expand previous examinations of CE experimentation for SoS and introduce Security Chaos Engineering (SCE) for SoS. These SCE experiments include terminating message service, flooding multi queues/message, and injecting corrupted Service. SCE assumes compromise by adding a malicious actor to the tests that can induce adversarial failures into a SoS. For our SoS testbed, we instantiated a virtual Unmanned Aerial Vehicle (VUAV).
We use the open-source Chaos Toolkit to run consistent CE and SCE experiments on the VUAV. Chaos Toolkit with SCE exposes the VUAV attack surfaces to evaluate performance and system security. This research allows us to establish an understanding of baseline system performance and gaps in procedures, techniques, and tools from the state of the art as applied to DoD-relevant systems like SoS. We use the load placed on the Central Processing Unit (CPU) and Random-Access Memory (RAM) by the VUAV as metrics for baseline performance. The results showed that these two metrics did not provide enough fidelity in where CE/SCE creates failures. Feeding these results into the CE methodology allows for additional metrics to better pinpoint failures with CE/SCE testing.

Keywords: Chaos Engineering, System of Systems, Security Chaos Engineering, Attack Tree

1. INTRODUCTION

1.1 Background

Modern computer systems are getting more complex as we are quickly adopting practices that increase flexibility of development and velocity of deployment. This adoption solves diverse challenges among different development levels; however, this also raises an alarm of how much confidence we can have in such complex systems that respond to different turbulent conditions that will inevitably occur. Common turbulent or "unusual" events include a user entering an unforeseen combination of inputs, or a data center being destroyed by natural events, both of which may seem like an attack based on day-to-day norms. To deal with these challenges a new set of applied research tooling based in assuming failure and compromise has emerged in the following disciplines:

- **Chaos Engineering (CE)** is an engineering practice that seeks to test/assess the resiliency of systems to natural failures.
- **Security Chaos Engineering (SCE)** adds a malicious actor to the tests that can adversely induce failures into a system. These principles can be applied at any layer of the system (from the hardware to the networks, to the applications).

Most CE methodologies have been developed and implemented on commercial clouds, like Netflix and Amazon. Netflix maintains and uses Chaos Monkey [14] for randomly terminating virtual machines among their production services in order ensure their production services resiliency against instance failures. Chaos Monkey automatically runs during Netflix's business hours and produces a summary of the results for Netflix's Engineers to analyze [7]. Netflix uses Simian Army [15] on Amazon Web Services (AWS), which adds fault Injections on top of Chaos Monkey by creating autonomous software agents, called *Monkeys*, to improve performance and resiliency of their cloud's applications and services. A common CE technique is fault injection, which injects faults into a designated system to understand how it

behaves during the presence of faults [7]. Currently, Amazon provides management service called Amazon Web Services Fault Injection Simulator (AWS FIS) to perform fault injection experiments for customers AWS workloads [16].

The Department of Defense (DoD) has begun adopting CE as well. The Kessel Run (a.k.a., the Air Force Life Cycle Management Center's Detachment 12), a DevSecOps unit applies CE experiments to Black Pearl and other software units within DoD [17]. Also, Lincoln Laboratory at MIT introduce Disruption Platform to Practice CE to improve the resiliency of in the DoD mission systems [18]

This increased CE adoption necessitates the need to understand the baseline system performance and gaps in procedures, techniques, and tools. This is especially true as further state of the art CE techniques are applied to DOD-relevant System of Systems (SoS). SoS are complex systems built for specific task from diverse systems or dedicated systems that pool their resources and capabilities together to offers more functionality and performance than simply the sum of the constituent systems.

Open-source CE can support and perform different CE experiments of the designated system. There is a long list of open-source CE tools. In this research, we used Chaos Toolkit to implement our CE experiments. Chaos Toolkit is a tool that provides a streamlined approach to writing and executing CE experiments on a system. It does this through the definition of experiments in a JSON format. Each experiment has a hypothesis where a steady state can be defined. This allows users to identify whether a CE experiment brought their system out of its steady state and to what degree. The Chaos Toolkit also provides support for methods in the form of probes and actions. Probes allow users to monitor their system during experimentation, while actions allow you to perturb a system during experimentation. Actions are the primary way to introduce chaos into your system. By using actions to shut down services or overload functions, you can monitor the effects of chaos upon the production system.

Chaos Toolkit also provides rollbacks at the end of each experiment. While not necessary, rollbacks give users the ability to revert methods taken during an experiment. This allows a user to gracefully return the system to a steady state. One of the greatest strengths of Chaos Toolkit comes in its ability to execute Python functions and shell scripts. Chaos Toolkit ships with support for executing python functions as methods, allowing a user to create chaos in their system through python scripting. The Chaos Toolkit also allows you to execute shell scripts as actions or probes. This provides flexibility in what a user can do with this tool. By executing python scripts inside shell scripts, a user can have Chaos Toolkit take almost any action on their system.

The rest of this paper is organized as follows. Section 2 discusses the background and related work. Section 3 introduces the motivation and paper contributions. Section 4 presents SoS testbed design and the setup. Section 5 discusses results of CE experiments. Section 6 presents our conclusions and recommendations.

2. RELATED WORK

Researchers have studied and used CE to find weaknesses in systems and assess their resiliency. As part of this effort, we present some the most recent advances in CE within this section. Beginning with Pierce et al. [7], who conducted initial research into the applicability of using CE on middleware network services. They evaluate the limits and resilience of message applications using fault injection and network manipulation. To perform fault injection, they throw different Java programming exceptions of targeted methods under different circumstances using Triple Agent and Perses that manipulate byte code passing through the JVM and find targeted methods to inject exception into the JVM. To perform network manipulation, they tested two different message applications written in C++ and Java respectively. The network manipulation experiment includes packet corruption, packet delay, packet loss, and packet reordering. Attention was given to the number of packets affected by a failure and if one affected packet is correlated to next preceding packet. The performance of application and system are observed using System Health Monitor (SHM). Finally, the out of band SHM exchanges messages with message applications in certain timeframes to check if they are operating properly.

Torkura *et al.* [1] propose Risk-Driven Fault Injection (RDFI) to detect security vulnerabilities in multi-cloud Infrastructure. RDFI associates cloud security among the CE principles to improve security benefits. RDFI is implemented

in CloudStrike, which is a cloud security system that implements CE principles. RDFI injects security faults in the target cloud infrastructure to detect failures that impact the confidentiality, availability, and integrity (CIA). RDFI includes four steps: Execute, Monitor, Analyze, and Plan. Execute injects security faults in the selected cloud infrastructure. Monitor maintains real-time visibility of the target cloud infrastructure. Analysis collects security information and refines it to identify vulnerabilities and the impact of security risks. Plan uses the security information acquired to select security mechanisms to protect the cloud infrastructure. Plan also uses security information to enrich fault models by planning more attacks on other assets in the cloud infrastructure. The authors claim that technique provides an effective model for emerging security remediation according to security information gained through security fault injection. Their technique along with CloudStrike have been deployed on AWS and Google Cloud Platform (GCP).

Zhang *et al*. [6] propose a Java chaos engineering system ChaosMachine to detect the resilience strengths and weaknesses of the system during try-catch block execution. ChaosMachine comprises the three components: monitoring sidecar, perturbation injector, and chaos controller. The monitoring sidecar collects information required for system resilience analysis. The perturbation injector throws exceptions into system during runtime. The chaos controller controls perturbation injectors and analyzes information collected by monitoring sidecars. ChaosMachine has deployed against open-source Java applications including content management and e-commerce. The authors claims that ChaosMachine able to identify and analyze try-catch blocks located in Java classes and produce actionable reports to improve resilience of the targeted system.

Additional research proposed by others is a framework to implementing CE for commercial and industrial uses [3, 4] while other advocating for using CE practice for security in response to the growing threat of cyberattacks [5,12]. Our research builds upon this foundation by defining a methodology and approach for running SCE experiments.

3. MOTIVATION AND PAPER CONTRIBUTIONS

In this paper, we demonstrate our methodologies for understanding and designing CE/SCE experiments using the open-source Chaos Toolkit [13]. This research provides understanding of CE/SCE parameter space, optimizing the CE/SCE methods within that parameter space. In completing this effort, we have advanced our ability to apply CE/SCE in DoD systems and industrial control systems that often do not have any standardized means for CE/SCE testing. There has been less application of CE/SCE to industrial control systems. Thus, we use Open PLC throughout our VUAV testbed. This allows us to understand impacts beyond the cloud and focus on how chaos impacts both software and hardware.

This paper attempts to develop a repeatable methodology for applying CE (including SCE) to complex systems such as SoS that expose new threats or attack surfaces. Understanding the SoS performance under chaos aids in understanding these attacks. Thus, revealing new methods and procedures (e.g., attack tree development) and tools/techniques for applying CE/SCE principles to DoD SoS.

4. SOS TESTBED DESIGN AND SETUP

4.1 SoS Testbed: A Global Position System (GPS) Sensor System

A key method for testing hardware interactions with OpenPLC is to include actual hardware in the system under CE/SCE testing. In our VUAB testbed we use Global Navigation Satellite System (GNSS) hardware to transmit signals realistic navigation signals. Accordingly, the DoD has developed the Navstar Global Position System (GPS) based on GNSS, which is a space-based navigation system created in the 1970's. GPS determines the position, the velocity, and the time in a common reference system, anywhere on or near the Earth.

Thus, in this research project, we used GSG-5 Series GNSS Simulator as shown in Figure 1 to provide a realistic hardware approach for testing.

Figure 1. GSG-5 Multi-Function GNSS simulator

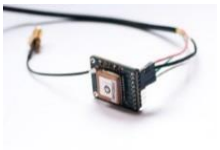


Figure 2. Adafruit Ultimate GPS Breakout board

GSG-5 Series simulators reproduces the environment for use by a GNSS receiver. As an essential part of a scenario-based simulations, such as drones flight paths or connected cars driving routes, the GSG-5 can be used for testing a variety of intelligent applications. GSG-5 offers ease of use with an out-of-the-box configuration including a comprehensive set of pre-defined scenarios. GSG 5 can simulate all constellations, all frequencies, and movements/trajectories anywhere on or above Earth [8].

We use the GSG-5 to provide a pre-set path of GPS co-ordinates to our VUAS through a GNSS receiver via the Adafruit Ultimate GPS Breakout board, as shown in Figure 2. It receives the GPS signals from the GSG-5 through a SMA-to-uFL cable that attaches directly to the breakout board. The GPS breakout board includes the MTK3339 chipset, which outputs the GPS signal information into the onboard serial RX port. This port, along with the TX, Power, and Ground, is connected to a general processing system, such as a Raspberry Pi, through an FTDI USB breakout cable.

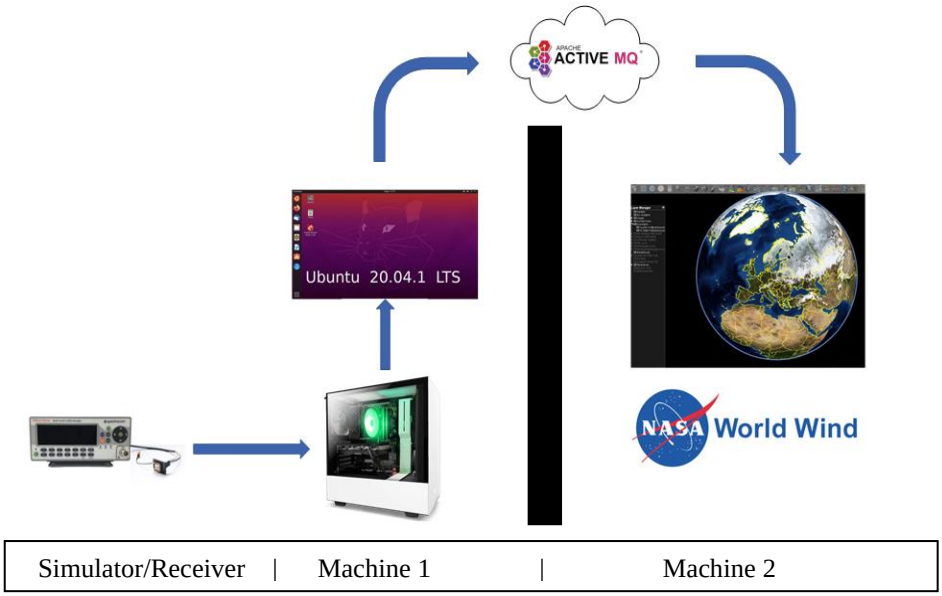


Figure 3. GPS Sensor System

The Testbed in Figure 3 simulates a VUAV that contains both hardware and virtual software like the OpenPLC components. A GSG-5 simulates the actual hardware RF signals that contain simulated GPS data from up to 16 satellites. These signals can either be transmitted through the air via GPS antennas, or through SMA cables. In this testbed, we transmitted GPS signals between two virtual machines via a cable. The Adafruit GPS breakout board receives the GPS data and then transmits it to Machine 1 through an FTDI USB cable and provides the conversion point between our hardware and software based VUAV.

On the software receiving side, we use Navigation Service (NavService) to ingest the GPS data through a socket and packages it into a common DoD format. NavService then publishes the packaged data to the topic “position report” in ActiveMQ running on Machine 1. NASAs WorldWind display running on Machine 2 subscribes to the position report

topic in ActiveMQ to track the UAV movement. WorldWind is Java-based open-source source virtual globe Application Programming Interface (API) for visual interaction of globe geographical information.

Machine 1 also simulates a virtual gas tank through OpenPLC for the VUAV. When the VUAV runs low on gas, a virtual switch connected to a virtual Programmable Logic Controller (PLCs) is triggered. The PLC is used to transmit the low gas signal through ActiveMQ to a low gas monitor on Machine 2. A PLC implementation for the low gas signal was chosen because they are found in many DoD systems for data acquisition and control [9]. The manufacturing industry also heavily uses PLCs to support easier interfacing among machines, increase part repeatability, and generally bring in the latest factory automation concepts [10].

4.2 Methodology

The research methodology is a standardized guide that states how the research is to be prepared and why we select or adapt specific methods and tools for this research. The research methodologies used in computing science is as follows [11]:

1) The formal methodology:
   A mathematically based technique is used to specify, verify, and implement systems or algorithms under development and is widely used by software and hardware engineering practitioners.
2) The experimental methodology:
   It is used to evaluate a new solution proposed for a problem by questioning the system and its objectives to thus identify overall system requirements.
3) The building methodology:
   Involves constructing a complete system or part of a system that shows the potential and effectiveness of the proposed solution. That includes suggesting new features and techniques which have not been included in the system before.
4) The process methodology:
   Process methodology explains all processes that are used to accomplish system's objectives. It is continuously applied in agile software engineering improvement and development.
5) The model methodology:
   The model methodology presents a model to mimic a real complex system which cannot be implemented in the real world due to the cost or accessibility issues. This methodology most often relies on simulation, which shows the validity and usability of the proposed solution.

The above methodologies can be combined with other methodologies. For example, in software development, the formal methodology is combined with the build methodology to implement and embed a subsystem to the main system. Likewise, we aim to develop a methodology that combines features of the above methodologies for understanding the CE/SCE parameter space, optimizing the CE/SCE methods within that parameter space, and applying CE/SCE to simulated military systems.

To do this we combine formal and experimental methodologies to propose and perform CE/SCE experiments against the designated system, as shown in Figure 4. The CE/SCE methodology cycle is as follows:

1) Understand System Behavior. Initially, a selection of metrics is chosen based on current understanding of system functionality. These selected metrics are then used to establish the steady state of system.
2) Create New Hypothesis. A CE hypothesis is created by asking "What if?" questions about the system. These questions need to be as specific as possible based on the selected metrics for the system. Only create hypotheses on parts of the system already seen as resilient
3) Design Experiment. Only one hypothesis should be selected for experimentation at a time to simplify results analysis. Metrics of the system exercised by the experiment need to be identified along with defining the experiment's scope. Start with the smallest scope to minimize the detrimental effects of a failed experiment, also known as the blast radius.
4) Run Experiment. The system should be running in production mode while experiments are run. This is what differentiates CE from more conventional techniques like penetration testing. Being in production mode also elevates the risks associated with the blast radius. A way to stop your experiment and get back to the normal steady state as fast as possible needs to be identified before kicking off the experiment.
5) Analyze Results. One of the critical components of the CE methodology is the time it takes to detect a failure. If none of the selected metrics for the system indicates that a failure has occurred, then understanding of system behavior must be revisited, restarting the cycle. If the metrics do detect a failure, then the length of time must be within an acceptable range or possible improvements should be considered.
6) Improve System. If improvements are developed in the previous step, they may now be implemented in the system.

If multiple improvements are possible for the given experiment, only the one deemed to have the largest impact should be implemented. Step 7 is skipped, and the experiment is then re-run.

7) Increase Blast Radius. If the time to detect the failure does fall within acceptable ranges, then step 6 is skipped and the scope of the original experiment is expanded and re-run with the updated scope. Once the maximum scope is reached and either the time to detect the failure is acceptable or the failure is eliminated from the system, then the system resiliency is said to be improved and the next hypothesis can be tested.

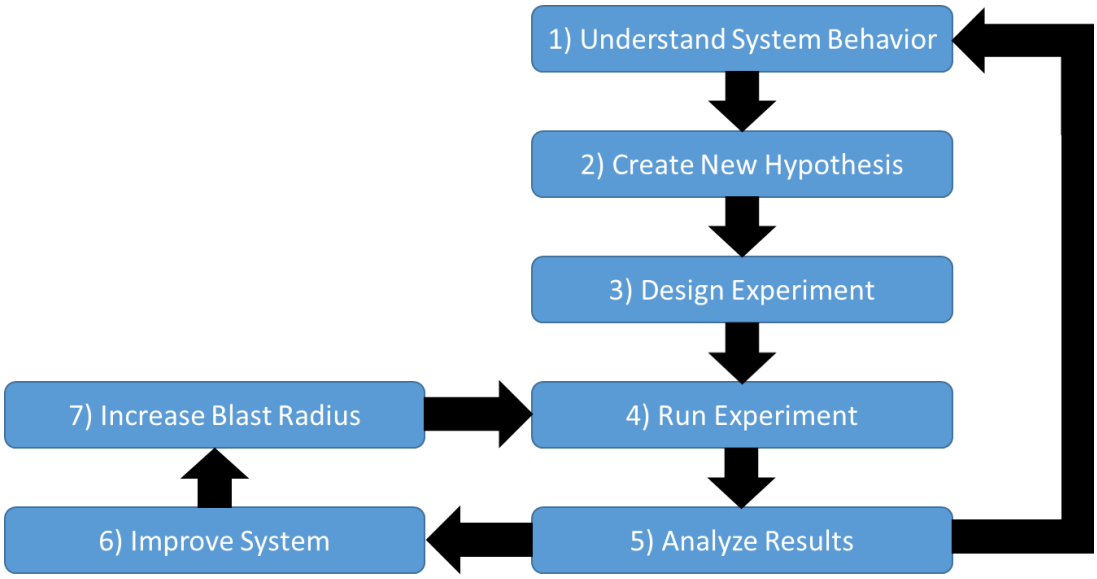


Figure 4. CE Methodology

4.3 Attack Tree and Threat Model Assumptions

The Attack Tree is one of the formal and methodical processes we used to describe the security of the system. An attack tree is a way of enumerating all possible attack surfaces within a system. The first step in developing the attack tree for SCE is by describing the threat model of the attacker. The threat model presents assumptions about what levels of access an attacker has for a specified system before an attack is launched. The following assumptions were considered for the attack trees developed in this paper:

1) The threat model assumes the attacker has access to the network our system is connected to. In our testbed, this means that the attacker either has a hardline connection into a network switch or is in range of the WiFi access point into the network and knows the password.
2) Another assumption in our threat model is that remote access into the system running the target process has remote access enabled. For our testbed, we assume that the target system either has unsecured protocols like Telnet or Remote Shell (rsh) enabled, or secure ones like Secure Shell (ssh) without strict security enforcements enabled.
3) The final threat model assumption is that the attacker has user level access to the system that the process is running on. This could mean that the Telnet protocol has guest access enabled or that the attacker already knows the name of a configured ssh user that lacks a specified password.

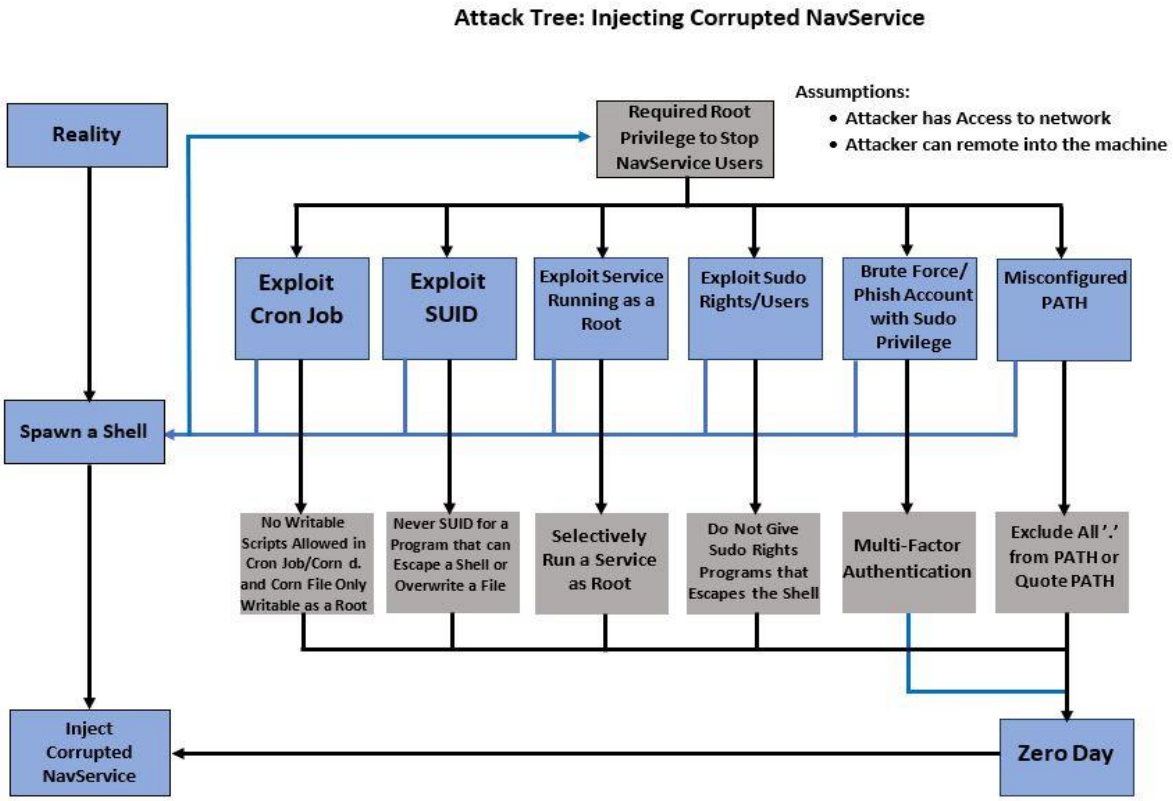


Figure 5: Injecting Corrupted Navigation Service Attack Tree

An attack tree begins with the target system having minimal protections enabled. Therefore, the entry point into the attack tree begins with an attacker establishing a remote shell on the target system. The attack described in Figure 5 assumes that the remote shell is being established on the UAV itself. Normally, the UAVs onboard navigation service (NavService) receives GPS data from an antenna, packages it into a common messaging format within the system and sends it out to a message broker for distribution. The navigation service is initially assumed to be running with only user level access. This allows the attacker to stop the original navigation service and start an attacker created one. In our testbed, the corrupted navigation service injected into the system ignores GPS data from the UAV antenna and sends false GPS coordinates to the message broker. The attack is now considered successful.

From there, the attack tree now considers elevated security postures from the system as the graph moves toward the right. From the spawned attacker shell, the elevated system security now assumes that the navigation service is running with root level privileges. This means that an attacker would need to perform a privilege escalation attack from the established shell to gain the privileges necessary to stop the running navigation service. The attacker could still run the corrupted navigation service alongside the correct one at this point, but the attack would only be considered a partial success as conflicting GPS coordinates would quickly lead to attack detection.

The attack tree in Figure 5 uses grey background boxes to indicate system protections, while blue boxes indicate attack actions. The arrows show the system evolving from an unprotected to a protected state. A system with all possible security features enabled leads to the zero-day attack. A zero-day attack is one that has not been identified and reported widely. However, zero-day attacks cannot be defended against and must be included in the iteration of the CE process.

Figures 6 show an examples of attack tree of Killing Active MQ/WorldWind

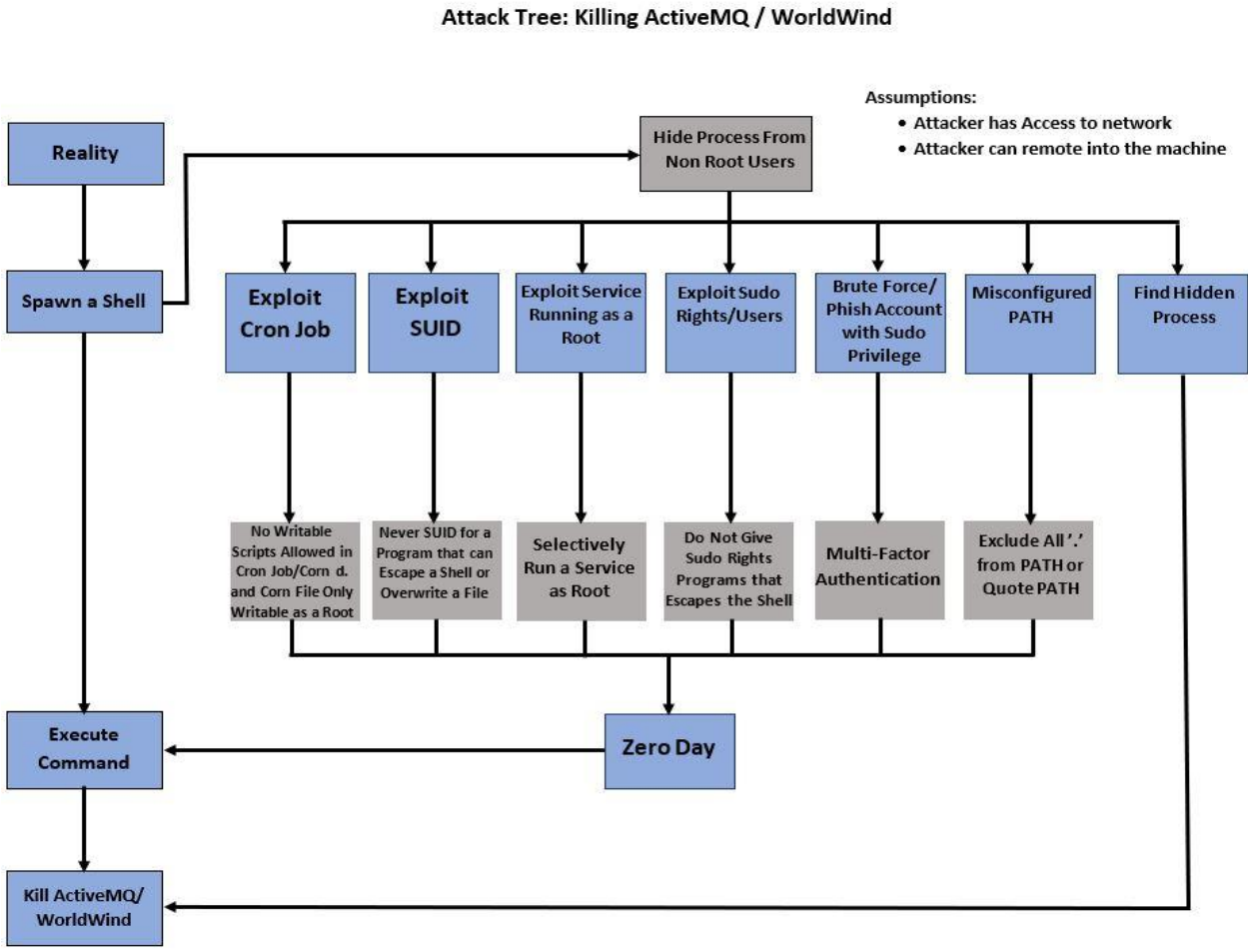


Figure 6: Killing ActiveMQ / WorldWind Attack Tree

5. EXPERIMENT RESULTS AND DISCUSSION

5.1 CE Experiments

We use our attack trees as the basis for our CE/SCE experiments that this section discusses as the relate to the VUAV testbed. We run consistent CE/SCE experiments on our testbed to evaluate its performance. These experiments include terminating ActiveMQ (recall provides the message service), ActiveMQ single queue multi-message flooding, ActiveMQ multi-queue single message flooding, terminating WorldWind, and injecting corrupted NavService as shown in Figure 5. For the first three experiments, we run each experiment (using single virtual machine) for 15 minutes, then we switch to the second, and third respectively. We run the fourth and fifth experiment separately (among two virtual machines) due the requirement of NavService and WorldWind. We evaluate the performance of testbed by measuring the CPU load and RAM Usage of the system.

We follow the same formal methodology described earlier in our experiments. To adopt that methodology in CE/SCE we streamline it to Observability, Steady State, and Hypothesis. The results of these experiments are discussed in the next

section and finally the expansion of tests is left for future research. The experiments we implemented on the VUAV testbed are as follows:

5.1.1 Terminating ActiveMQ Experiment

1) *Observability*: This experiment returns normally if ActiveMQ was shut off and restarted correctly. If any part of the experiment would fail, then the output from Chaos Toolkit would state how and where it failed and ActiveMQ's website would also be unreachable. The observable output would not being able to access the ActiveMQ local website.
2) *Steady State*: The steady state is ActiveMQ being able to queue and dequeue messages from publishers and subscribers.
3) *Hypothesis*: Our hypothesis is that ActiveMQ will gracefully handle being shut off and restarted with no significant issues for the overall system.

5.1.2 ActiveMQ Single Queue Multi-Message Flooding Experiment

1) *Observability*: This experiment returns normally if we are able to flood a single queue within ActiveMQ and ensure the service is still running. If there are issues, Chaos Toolkit tells the user whether the experiment deviated from its intended instructions or failed outright. The observable output of this experiment is a queue within ActiveMQ called 'FloodingQueue' with 10 million messages enqueued and none dequeued.
2) *Steady State*: The steady state for this experiment is whether our system can continue to send messages over ActiveMQ while a single queue is being flooded with test messages. We will monitor whether this flooding causes a drop in performance and see if it causes spikes in CPU load or RAM usage.
3) *Hypothesis:* Our hypothesis is that ActiveMQ will be able to gracefully handle the flooding of test messages. We also hypothesize that CPU load and potentially RAM usage will spike due to this experiment.

5.1.3 ActiveMQ Multi-Queue Single Message Flooding Experiment

1) *Observability*: This experiment returns normally if we are able to flood multiple queues within ActiveMQ and ensure the service is still running. If there are issues, Chaos Toolkit tells the user whether the experiment deviated from its intended instructions or failed outright. The observable output of this experiment is 10 million queues with incrementally larger names within ActiveMQ, each having a single message enqueued and none dequeued.
2) *Steady State*: The steady state for this experiment is whether our system can continue to send messages over ActiveMQ while multiple queues are being flooded with a single test message. We will monitor whether this flooding causes a drop in performance and see if it causes spikes in CPU load or RAM usage.
3) *Hypothesis*: Our hypothesis is that this experiment will significantly slow down the performance of our system. We hypothesize that CPU load and potentially RAM usage will spike due to this experiment.

5.1.4 Terminating WorldWind Experiment

1) *Observability*: This experiment returns normally if we are able to terminate the WorldWind process and ensure that it restarts properly. If there are issues, Chaos Toolkit tells the user whether the experiment deviated from its intended instructions or failed outright. The observable output of this experiment is the termination of WorldWind, and the subsequent restart of the WorldWind service.
2) *Steady State*: The steady state for this experiment is whether the remainder of our system can continue sending messages and functioning without the WorldWind display functioning. We also want to ensure that coordinates continue being displayed after a restart of WorldWind.
3) *Hypothesis*: Our hypothesis is that our system will be able to gracefully handle the termination of WorldWind and subsequent restart. We expect to see a dip in CPU load and RAM usage when WorldWind is terminated, followed by a return to steady state upon its restart.

5.1.5 Injecting Corrupted NavSrvc Experiment

1) *Observability*: This experiment returns normally if we are able to inject and run the corrupted NavSrvc into our system, and subsequently return to our good NavSrvc. If there are issues, Chaos Toolkit tells the user whether the experiment deviated from its intended instructions or failed outright. The observable output of this experiment is the corrupted coordinates being displayed, and the subsequent restart of the known good NavSrvc.
2) *Steady State*: The steady state for this experiment is whether our system can return to correctly displaying known good coordinates after being fed corrupted coordinates.

3) *Hypothesis*: Our hypothesis is that our system will be able to gracefully handle switching between being fed corrupted packets and known good packets. We hypothesize that we will see coordinates in two very different places on our display, but that our system will be able to correctly handle the jump between them.

5.2 Results and Discussion

After first baseline the system performance. We follow the CE/SCE process and run our experiments sequentially for experiments 1, 2, & 3 and independently for 4 and 5. We evaluate the performance of testbed by measuring the CPU load and RAM Usage. Also, we monitor and detect the position coordinates of the UAV after injecting corrupted NavSrvc experiment.

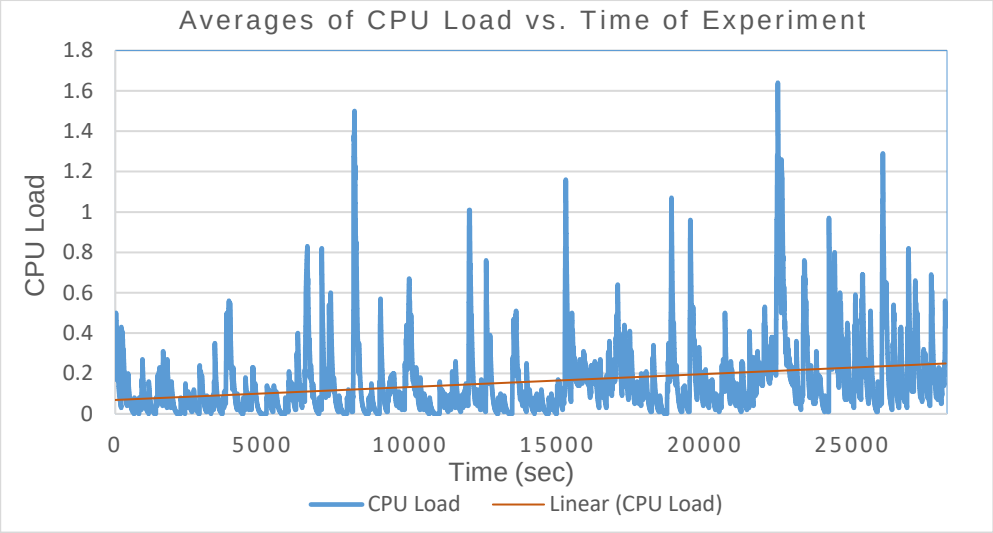


Figure 7. Averages of CPU Load vs. Time of Experiment

Figure 7 shows the average rate of CPU load versus the time for the production environment while the experiments were being run. Each major spike in the CPU load coincides with the execution of an experiment on the production environment. The trendline shows that there is a slight growth in the CPU load over the entire duration of experiments.

After running all our experiments on our simulated VUAV testbed, we were able to gather the data seen above. If we look at the data for CPU load, we can see that most of our data is within the range of 0 to 0.3. There are many spikes happening in this data, both small and large. We believe that these spikes occur whenever an experiment is run. The dips in the data can be correlated to the times whenever a service is terminated via an experiment.

The largest spikes out of all the experiments occurred during the Flood ActiveMQ experiment with the highest CPU load value reaching 1.64. This can be attributed to the two million placeholder messages being sent to ActiveMQ in quick succession. There were not any major drops in performance for the whole production testbed with no lags or unexpected whole system crashes.

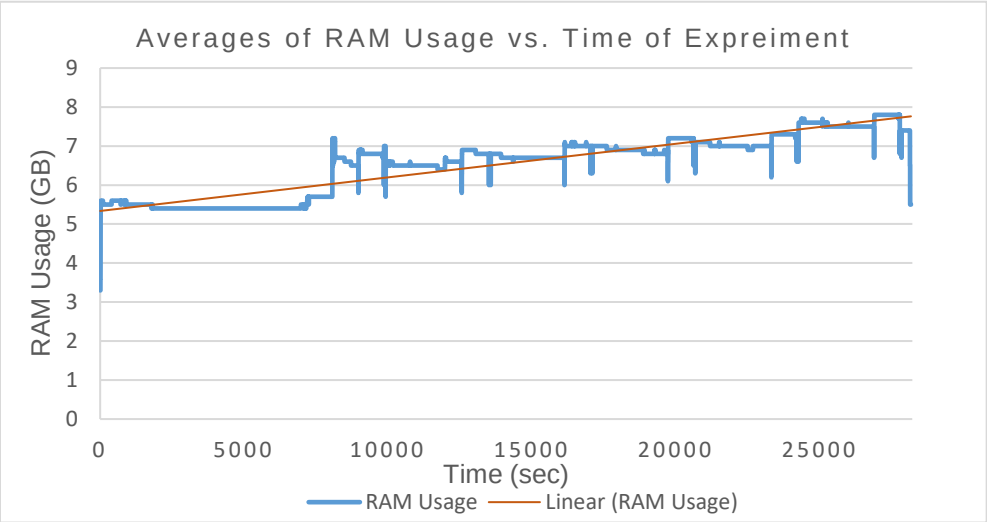
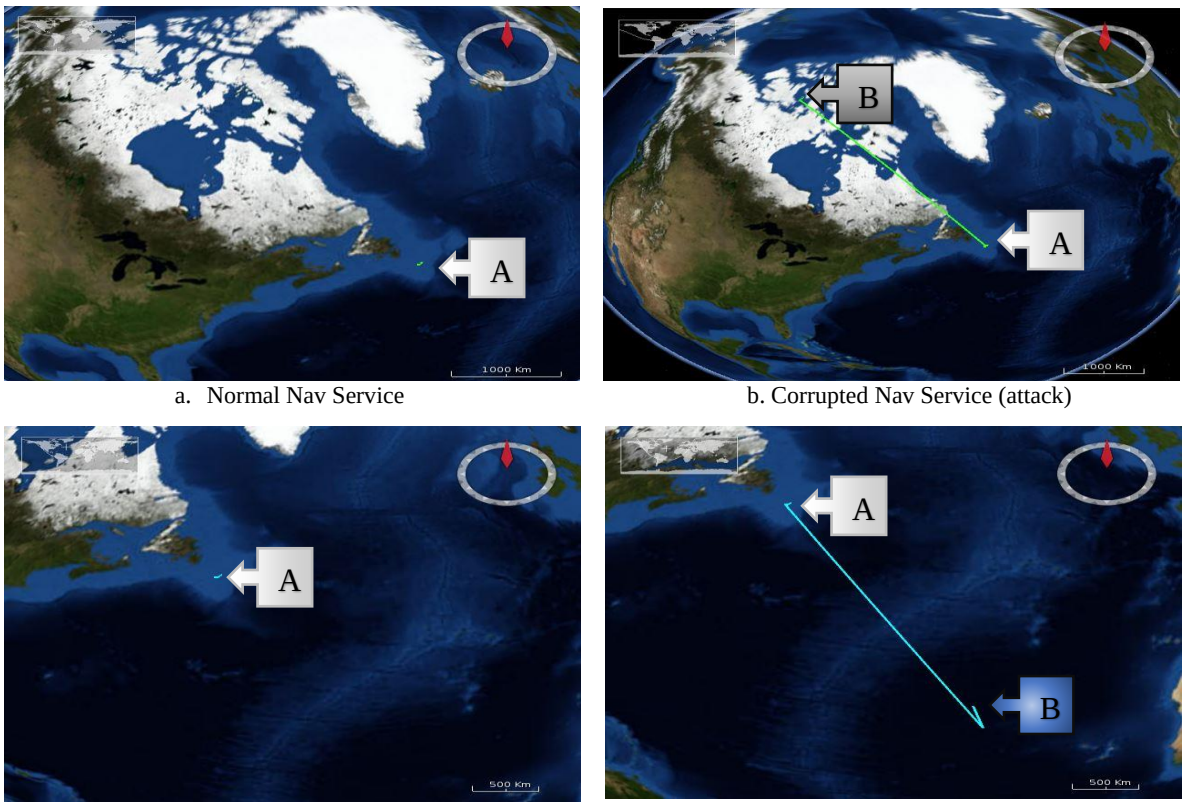


Figure 8. Averages of RAM Usage vs. Time of Experiment

Figure 8 shows the average of RAM Usage versus the time for of the production environment while the experiments were run. At this point we are still investigating what the data means, as the drops in RAM Usage do not coincide with the runtime of the experiments. There is not a uniform reaction from the production environment for experiment two as one Flood ActiveMQ experiment might drop the RAM Usage, but another run of the same experiment will not. Our future research will look to confirm that RAM Usage may be a metric that does not provide meaningful results in this use case.



a. Normal Nav Service

b. Corrupted Nav Service (attack)

c. Normal Nav Service

d. Corrupted Nav Service (attack)

Figure 9. UAV coordinates changed from normal Nav Service (a &c) vs the attack corrupted Nav Service (b & d)

Using Corrupted Nav Services, we interrupt normal Nav Services of UAV and attempt to change the UAV coordinates, and we show the change of coordinates using WorldWind. Figure 9 shows two different attacks on UAV using Corrupted Nav Services. Figure 9a shows normal UAV operation in the southeast while 9b shows the first attack via corrupted Nav Services, which changes the UAV coordinates to the north of the United States. Figure 9c shows normal UAV operation moving southeast while 9d shows a second attack that changes the UAV coordinates to the deep south into the Atlantic Ocean.

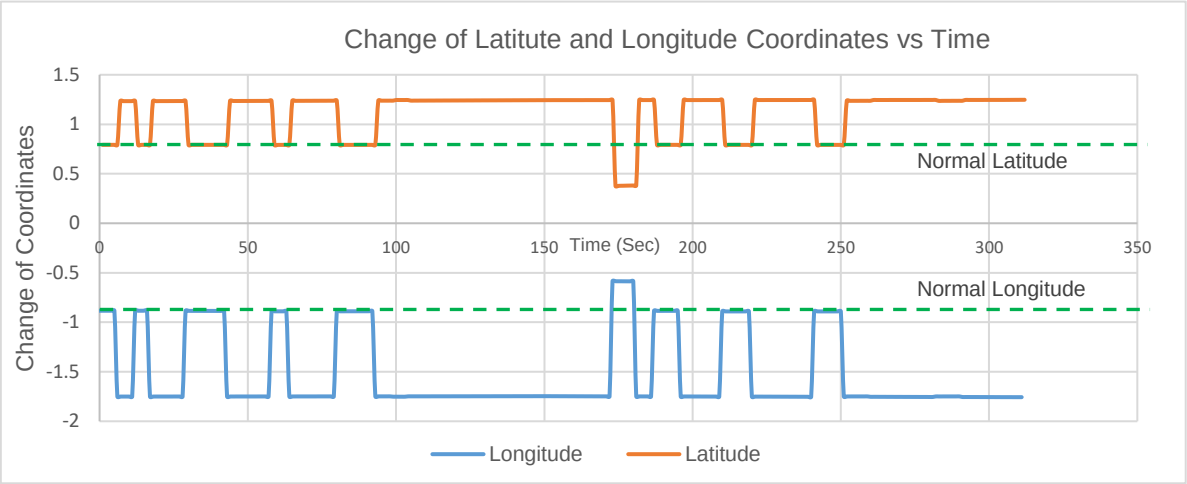


Figure 10. The Change for latitude and longitude of UAV after different attacks by corrupted Nav Service

We run different Corrupted Nav Services experiments during regular UAV operations, and we would be able to monitor the change of coordinates, including latitude and longitude values. Figure 10 shows the fluctuation of latitude and longitude above and below the normal values of latitude and longitude of UAV (indicated by the dotted green line). We can see that the UAV was under different attacks. We used a specific threshold value to detect the difference among latitude/longitude values. Also, we use different times of attack as we can see the different lengths of latitude (red line) and longitude (blue line).

6. CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

The goal of the VUAV testbed is to implement CE experiments to test and assess the resiliency of the SoS as shown in Figure 3 using the open-source CE tool (Chaos Toolkit). Chaos Toolkit supports and performs different CE experiments of the designated Virtual Unmanned Aerial Vehicle (VUAV) SoS. we use formal attack tree methodologies to specify and verify the proposed CE/SCE experiments against the SOS testbed. Notably, the selection of metrics for system performance is the most difficult portion of the CE/SCE methodology. This is apparent when we utilize CE in conjunction with Security Chaos Engineering. However, we believe these baseline experiments provide an introductory understanding of designing and selecting CE/SCE testbeds and parameters that document how systems experience turbulence and return back to a steady state.

6.2 Recommendations

We believe that self-healing software is the future outcome from CE/SCE research. For example, a user can inject chaos into their system and allow self-healing software to identify any issues that arise and protect against them in the future. Zhang et al. present an example of self-healing software with their tool [12]. As of current, most fault-injection models of Chaos Engineering utilize the Java programming language. We hope to change this in the future and create language-agnostic tools for fault injection in languages such as C++ and Python. In doing so, more robust, and secure technical infrastructure can be created across a variety of platforms.

Along with growing the available languages for fault injection models, the metrics being measured can also be increased. With a larger, more distributed system of systems, throughput is an additional key metric to be considered and investigated. The throughput of a system is crucial to the functionality of the system and is also a critical target for attacker to focus on to disrupt the system. In the future, we hope to utilize CE to work with more DoD complex systems and harden their resilience.

REFERENCES

[1] K. A. Torkura, M. I. H. Sukmana, F. Cheng and C. Meinel.2020. "CloudStrike: Chaos Engineering for Security and Resiliency in Cloud Infrastructure," in IEEE Access, vol. 8, pp. 123044-123060, 2020, doi: 10.1109/ACCESS.2020.3007338.

[2] J. O. Kephart and D. M. Chess. 2003. ‘‘The vision of autonomic computing,’’ Computer, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[3] Hugo Jernberg, Per Runeson, and Emelie Engström. 2020. Getting Started with Chaos Engineering - design of an implementation framework in practice. In Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '20). Association for Computing Machinery, New York, NY, USA, Article 43, 1–10. DOI: https://doi.org/10.1145/3382494.3421464

[4] Kazuyuki Aihara and Ryu Katayama. 1995. Chaos engineering in Japan. Commun. ACM 38, 11 (Nov. 1995), 103–107. DOI: https://doi.org/10.1145/219717.219801

[5] Jamie Lewis and Chenxi Wang. 2019. Chaos Engineering: New Approaches To Security . A Rain Capital Research. Accessed On May 12, 2021 From: https://static1.squarespace.com/static/5a927314b27e397ab120694b/t/5d5ef7e43be7270001c0e9fb/1566504938565/Rain+Capital+Chaos+Enginneering.pdf

[6] L. Zhang, B. Morin, P. Haller, B. Baudry and M. Monperrus, "A Chaos Engineering System for Live Analysis and Falsification of Exception-handling in the JVM," in IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2019.2954871.

[7] Tony Pierce, Jason Schanck, Alex Groeger, Raed Salih, Michael R. Clark, "Chaos engineering experiments in middleware systems using targeted network degradation and automatic fault injection," Proc. SPIE 11753, Open Architecture/Open Business Model Net-Centric Systems and Defense Transformation 2021, 117530A (12 April 2021); https://doi.org/10.1117/12.2584986

[8] User Manual with SCPI Guide, GSG-5/6 Series GNSS Simulator, Access May 2021 from: https://www.orolia.com/document/user-manual-with-scpi-guide-gsg-5-6-series-gnss-simulator/

[9] Anderson, C. M., Greenblatt, J. E., & Lively, K. A. (1998). The introduction of programmable logic controllers into US Navy Machinery Control Systems. Naval engineers journal, 110(1), 217-223.

[10] Ziemba, R. "Use of a Programmable Logic Controller (PLC) for Temperature, Position, Velocity and Pressure Control of Injection Molding Machinery." Conference Record of the 1988 IEEE Industry Applications Society Annual Meeting (1988): 1397-404 Vol.2. Web.

[11] R. Elio, J. Hoover, I. Nikolaidis, M. Salavatipour, and L. Stewart, K. Wong, “About Computing Science Research Methodology,” material for class CMPUT 603: Teaching and research methods Panned by J.N Amaral and M. Buro), University of Alberta 2001.

[12] S. Sharieh and A. Ferworn, "Securing APIs and Chaos Engineering," 2021 IEEE Conference on Communications and Network Security (CNS), 2021, pp. 290-294, doi: 10.1109/CNS53000.2021.9705049.

[13] The Chaos Engineering toolkit for developer, Access March 2021 from: https://github.com/chaostoolkit/chaostoolkit

[14] The Chaos Monkey git hub, accessed March, 2021 form: https://netflix.github.io/chaosmonkey/

[15] The Netflix Simian Army, The Netflix Tech Blog, accessed March 2022 form: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116

[16] AWS Faut Injection Simulator, AWS, Amazon, accessed March 2022 form: https://aws.amazon.com/fis/

[17] Kessel Run Delivers Chaos Engineering Practices to Black Pearl, Kessel Run, accessed November 2021 form: https://kesselrun.af.mil/news/CHAOS-Engineering.html

[18] B. Vu and O. Huang, “Chaos Engineering and the Disruption Platform”, Applied Resilience for Mission Systems, accessed March, 2022 form from: https://www.ll.mit.edu/sites/default/files/project/doc/2020-07/2Page-ChaosAndDisruption-v.10.pdf