A Potentially Specious Cyber Security Offering for 5G/B5G/6G

Software Supply Chain Vulnerabilities within Certain Fuzzing Modules

Steve Chan
Decision Engineering Analysis Laboratory, VT
San Diego, USA
e-mail: schan@dengineering.org

***Abstract*—A plethora of fuzzing Tactics, Techniques, and Procedures (TTPs) have been either proposed or described in the literature for the purpose of discerning software vulnerabilities with efficacy. The benefits of fuzzing have been well documented, such as when researchers found dozens of vulnerabilities in 4G LTE wireless networks, and fuzzing has become prevalent among the disparate actors within the wireless network ecosystem (to include 5G). However, fuzzing implementations are varied, and ironically, in some cases, implementations have utilized software bundles that have contained known "High Severity" Common Vulnerabilities and Exposures (CVE). On the surface, it seems that fuzzing the fuzzing module itself would constitute a simple solution to this issue. However, prototypical fuzzers have coverage issues (i.e., they only fuzz certain lines of code or sections of the software program). In addition, as numerous fuzzers utilize Docker containers, which are essentially inert when not in use, the complexity of the challenge is non-trivial. This paper introduces a fuzzing framework that capitalizes upon a sequence of bespoke grey-box concolic (i.e., hybridized symbolic and concrete execution) fuzzers (one set that fuzzes the next) to better address the coverage issue (as well as more likely to discern CVEs) and leverage their hybridized nature to overcome the disadvantages of black-box (higher computational performance, but lower coverage) and white-box fuzzers (e.g., lower computational performance, but higher coverage). The introduced bespoke grey-box concolic fuzzer architecture has certain advantages over other Coverage-based Grey-box Fuzzers (CGF) via the numerical stability-centric approach by which it selects seeds, undertakes seed scheduling, and operationalizes the seed pool.**

Keywords-cyber security; fuzzing; wireless networks; 5G; autonomous vehicles; grey-box concolic fuzzer.

I. INTRODUCTION

The growth within the 5G arena is well documented in the literature. According to TeleGeography, "nine 5G networks went live globally in Q1 2021, bringing the global total up to 172 networks" [1], and according to the Global Mobile Suppliers Association (GSA), there are now "511 commercially available 5G devices as of June 2021" [1]. To date, the rollout of 5G has occurred by way of three core service categories (a.k.a., "5G triangle"): Enhanced Mobile Broadband (eMBB), Ultra-Reliable Low-Latency Communications (URLLC), and massive Machine-Type Communications (mMTC). These service categories support a wide range of Quality of Service (QoS) needs. The QoS needs differ by application (e.g., fixed wireless access, connected machinery/equipment, video monitoring/ detection, as well as connected/autonomous vehicles) [2]. QoS needs are constantly evolving as existing applications become more sophisticated and emergent applications are designed for the envisioned capabilities of 5G, Beyond 5G (B5G), and the 6G ecosystem.

A key aspect of the 5G/B5G/6G ecosystem is that hardware is principally supplanted with software so that future upgrades will be software-centric. However, this increased utilization of Software-Defined Networking (SDN) within the network core also expands the attack surface opportunities [3][4]. In fact, the literature shows that cyber security researchers have found a plethora of security vulnerabilities (e.g., improper handling of procedures, invalid integrity protection, and security procedure bypasses), via fuzz testing (a.k.a., fuzzing), within wireless networks [5].

It should be of no surprise that governments and industries around the world are concerned about availability (a key aspect of the cyber notion pertaining to the Confidentiality, Integrity, and Availability Triad) being compromised, particularly as pertains to critical/strategic infrastructure and mission-critical applications [6]. Given the recent surge in issued directives, such as the "Improving the Nation's Cybersecurity" (Executive Order 14028, which was issued on 12 May 2021 and proceeded to direct the National Institute of Standards and Technology or NIST to enhance software supply chain security guidelines), it seems ironic that there remains software supply chain vulnerabilities within certain mission-critical software fuzzing paradigms; after all, these are the very mechanisms that are supposed to discern cyber vulnerabilities and enhance the cyber posture. The main contribution of the paper is to introduce a bespoke fuzzing framework that addresses the issues of limited coverage and inadvertent inherent vulnerabilities within certain fuzzing paradigms.

This paper is structured as follows. Section I introduces the problem space. Section II presents background information and discusses the operating environment and the state of the challenge. Section III delineates the referenced software supply chain challenge and presents some experimental findings derived from scrutinizing a particular architectural stack, which supports a mainstay of the 5G network core — the family of Fast Fourier Transform (FFT)-related functions for signal processing. Section IV posits a potential mitigation pathway for the discussed cyber

 ISBN: 978-1-61208-893-8

exposure. Section V concludes with some observations, puts forth envisioned future work, and the acknowledgements close the paper.

II. BACKGROUND INFORMATION

Within the 5G/B5G/6G ecosystems, maximizing spectrum efficiency by optimal allocation of frequency/time/power resources is vital, and the orchestration of the involved waveforms is complex. Exemplar waveforms include Generalized Frequency Division Multiplexing (GFDM), Filter Bank Multicarrier (FBMC), Orthogonal Frequency Division Multiplexing (OFDM), Universal Filtered Multi-Carrier Modulation (UFMC), etc. In turn, there are variants of these waveform types. For example, FBMC has two principal variants: Quadrature Amplitude Modulation (QAM) and real-valued Offset QAM (OQAM) (a.k.a. FBMC/OQAM). OFDM, which conjoins the advantages of QAM and Frequency Division Multiplexing (FDM), has an even greater number of variants. UFCM (a generalization of FBMC and OFDM) has greater variants still.

The library of FFT-related functions for signal processing is of critical import, and as just one example, the library is used for spectrum enhancement of the previously referenced Orthogonal Frequency Division Multiplexing (OFDM)-based waveforms within Fifth Generation New Radio (5G NR) development [7]; 5G NR is, in essence, a new Radio Access Technology (RAT) for cellular networks. The involved functions include not only FFT, but also Inverse FFT (IFFT), Real-Valued FFT (RFFT), Inverse RFFT (IRFFT), Short-Time Fourier Transform (STFT), and Inverse STFT (ISTFT), among others. In particular, STFT is a key requisite functionality within the 5G/B5G/6G ecosystem.

Prior research has indicated that the selection and utilization of, by way of example, specific STFT implementations from the available machine learning libraries/toolkits is critical; it is vital for the 5G/BFG/6G researcher/programmer to understand and contend with the implementation intricacies of the numerical algorithms being utilized for the involved functions. For example, signature consistency and dependency intricacies have been shown to result in errors and/or incorrect results, and these issues can cause a non-graceful degradation of the involved system [8]. Clearly, this would be unacceptable, particularly for those applications (e.g., autonomous vehicles), which have mission critical requirements that necessitate a certain QoS (and even Quality of Experience or QoE for some cases). In particular, those applications with mission critical requirements would be extremely sensitive to the issues of data rate (the data packet transfer rate per unit time), latency (the delay before the mandated transfer of data packets begins), and jitter (the variation in the time between data packets arriving).

Network Slicing (NS) is often utilized to satisfy varied NS QoS requirements (e.g., data rate, latency, jitter). Typically, a Service Function Chain (SFC) handles specific traffic within each NS. As each NS has its own cyber characteristics, each SFC will encounter varied cyber requirements. Consequently, the involved fuzzing modules will have varied implementations; each implementation will have its own set of potential cyber vulnerabilities. This challenge is more fully described in subsections A through D below.

A. Network Slice (NS)

To support a wide range of applications with varying QoS requirements (and particularly for mission critical QoS requirements), 5G/B5G/6G networks endeavor to provide high data rates with low end-to-end (E2E) latency and minimal jitter. To achieve this, among a myriad of approaches, NS is often utilized. In essence, each NS QoS requirement is met for the particular involved application while the overall involved 5G/B5G/6G network resources are still, ideally, optimally distributed for all involved NS [9].

B. Service Function Chain (SFC)

Operationally, NS leverages both Software Defined Networking (SDN) and Network Function Virtualization (NFV). In essence, NFV is the de-coupling of Network Functions (NFs) from a myriad of hardware appliances and the running of NFs as software in Virtual Machines (VMs). The various NFs (e.g., traffic control), which consist of the involved core network and Radio Access Network (RAN) component, are referred to as Virtual Network Functions (VNFs). Each SFC handles specific traffic within the NS, over varied technological and administrative ecosystems, and is an ecosystem in it of itself [10],[11].

C. Cyber Implications of using SFCs

The varied ecosystems can equate to physically dispersed, low-cost, short-range, small-cell antennas (e.g., low-power femtocells, picocells, and microcells). Functionally, each of these small-cell antennas leverages the 5G/B5G/6G dynamic spectrum sharing capability, wherein multiple streams of information share the available bandwidth, via a NS. In turn, each NS has its own varying degree of cyber risk [12][13]. To continually evaluate the ongoing risks, oftentimes a fuzzing module (which intentionally injects malformed inputs into the involved software, so as to ascertain failure/vulnerability points) is utilized.

D. Potential Cyber Vulnerabilities within the Fuzzing Module Itself

Given that 5G/B5G/6G protocols/specifications are still evolving and actively being defined by standards bodies, (e.g., 3rd Generation Partnership Project or 3GPP, Internet Engineering Task Force or IETF, International Telecommunication Union or ITU), and since each NS has its own associated cyber risks, varying implementations of fuzzing modules exist within 5G/B5G/6G architectural frameworks [14]. On the surface, it seems that the very use of a fuzzing module is in keeping with the spirit of cyber hygiene best practices. However, upon scrutinization of varied implementations, potential cyber vulnerabilities have been uncovered within the fuzzing module itself. In these cases, the fuzzing module represents a potentially specious

 ISBN: 978-1-61208-893-8

cyber security offering for 5G/B5G/6G, as it itself is subject to compromise.

Overall, the work presented in this paper differs from prior research in that a particular sequence of bespoke grey-box concolic fuzzers is utilized to mitigate against the known coverage issue and better discern known CVEs. The chosen sequence shows promise in that it overcomes some of the disadvantages of prototypical black-box and white-box fuzzers.

III. EXPERIMENTATION FINDINGS

This paper examined a 5G/B5G/6G architectural framework, which was used in a Technology Readiness Level (TRL) 5 (i.e., laboratory environment) and 6 (i.e., relevant environment). Typically, fuzz testing is conducted in a controlled, isolated laboratory environment (such as in the case of TRL 5), and isolation is often provided, via containerization. The notion of utilizing containers (as a testing target) is predicated upon the notion that it provides enhanced consistency and reproducibility (particularly when using container images) [15].

The previously discussed implementation intricacies (e.g., signature consistency, dependencies) that result in inadvertent errors and/or incorrect results are already problematic enough; however, this paradigm can be exacerbated when it is intentionally exploited. To better delineate this point, first, the containerization aspect is described. Second, some identified vulnerabilities related to the containerization paradigm are presented. Third, further vulnerabilities are identified within underlying legacy supply chains.

A. Containerization Aspect of Fuzzing

Traditionally, containerization has provided the desired isolation paradigm for fuzzing. The often-used workflow for containerization (e.g., specifying configuration, building a Dockerfile — a text file that contains all the commands required to build a Docker images — for each desired image, and using Docker Compose to assemble the images) facilitates reproducible/consistent testing results. Typical fuzzing architectures might utilize, by way of example, either of two container orchestration platforms for containerization: Docker (referring to either Docker Swarm or Classic Swarm, which was initially released in 2014, or Swarmkit, which was initially released in 2016; of note, Swarmkit stemmed from Docker's acquisition of SocketPlane, an SDN technology firm, in March 2015) and Kubernetes. Generally speaking, Docker, by default, prioritizes isolation between containers; this is construed by some to represent higher security. In contradistinction, Kubernetes prioritizes communication between multiple containers within the same pod; this is construed by some to represent higher efficacy, but lower security. Depending upon the specific requirement, the choice of orchestrator (i.e., Docker or Kubernetes) can be made explicitly.

Over the past several years, the leadership for container orchestration, potentially, has shifted from Docker to Kubernetes. In fact, Docker itself adopted Kubernetes and announced native support for Kubernetes at DockerCon Europe in Copenhagen on 17 October 2017. Yet, Docker's architecture enables users to select the desired orchestration engine (Docker, Kubernetes) at runtime. On the Kubernetes side, as of Kubernetes v1.20, Docker (specifically, Dockershim, which communicates with Docker Engine, which was renamed to Docker Community Edition or Docker CE in March 2017) has been deprecated as a container runtime.

Whatever the case may be with regards to the leadership for container orchestration, Docker images remain a mainstay within the development ecosystem. In addition, Docker Compose still remains in wide use for building Dockerfiles. While container images can indeed be built with tools, such as Kaniko (an open-source tool for building container images from a Dockerfile) [16], Podman, Buildah, and Buildkit, etc., Docker images assembled with Docker Compose may be more prevalent for certain facets of 5G/B5G/6G architectural stacks (particularly those utilizing GNU Octave). Indeed, there is a plethora of GNU Octave-related experimentation (e.g., [17]). By way of background information, whereas the utilization of docker run can indeed start up a container, Docker Compose is often utilized to automatically start up multi-container applications. Historically, Docker Compose has been the configuration component of the Docker ecosystem (whereas Docker Swarm was the scheduling component of the Docker ecosystem and determined where to place the containers within the cluster of Docker hosts, which were in the form of physical computer systems or VMs running Linux). Overall, containerization remains an accepted methodology for consistency/reproducibility; yet, this containerization paradigm for fuzzing modules introduces a new set of potential vulnerabilities.

B. Identified Vulnerabilities Related to certain Containerization Paradigms of Fuzzing Modules

The system of Common Vulnerabilities and Exposures (CVE) is a compilation of Information Security (InfoSec) issues. For example, CVE-2020-1971 identified an OpenSSL (wherein the identity of an involved website/web service is validated, and the information flowing between the website/web service and user is encrypted) "High Severity" issue, which had been reported on 8 December 2020 [18]. The Cybersecurity & Infrastructure Security Agency (CISA) noted that, "OpenSSL has released a security update to address a vulnerability affecting all versions of 1.0.2 and 1.1.1 released before version 1.1.1i. An attacker could exploit this vulnerability to cause a denial-of-service condition" [19]. In brief, the vulnerability issue simply affected OpenSSL v1.0.2, which was out of support and no longer receiving public updates; theoretically, OpenSSL v1.1.1i and beyond are no longer vulnerable to the referenced issue, and the matter should be closed. However, the key issue is not simply that there was a vulnerability issue in a deprecated version; more importantly, various

 ISBN: 978-1-61208-893-8

Github commentators/contributors had noted, such as years prior (e.g., [20]) that various bundled OpenSSL versions had been out of date. Yet, various offerings (e.g., various Docker Compose offerings) still continued to incorporate outdated OpenSSL versions, such as can be seen in Table I below.

TABLE I. DOCKER COMPOSE WITH BUNDLED OPENSSL VERSION

Exemplars	Constituent Components of the Bundle			
	Docker Compose Version	***Docker-Py Version***	***CPython Version***	***OpenSSL Version***
Case #1[a]	1.23.2, build 1110ad0	3.7.3	2.7.16	1.1.1c 28 May 2019
Case #2[b]	1.26.0-rc3, build 46118bc5	4.2.0	3.7.6	1.1.1d 10 Sep 2019
Case #3[c]	1.26.2, build eefe0d31	4.2.2	3.7.7	1.1.0l 10 Sep 2019

a. https://dockerlabs.collabnix.com/intermediate/workshop/DockerCompose/version_Command.html
b. https://github.com/docker/compose/issues/7348
c. https://github.com/docker/compose/issues/7686

Cases #1 through #3 represent just a small sample set of the implications of CVE-2020-1971. Additional OpenSSL CVEs are available at the OpenSSL portal (e.g., [21]), and other CVEs are available at the National Vulnerability Database (NVD) (https://nvd.nist.gov) as well as the U.S. Computer Emergency Response Team (CERT)-CISA (https://us-cert.cisa.gov) portals. As discussed, OpenSSL versions affected by CVE-2020-1971, among others, had been bundled with Docker Compose. Yet, despite the published CVEs, it should be noted that certain images of Docker Compose v1.28.0 might still be deploying with OpenSSL v1.1.1h (i.e., vulnerable to CVE-2020-1971); OpenSSL v1.1.1i and above have the security update for CVE-2020-1971. As of the writing of this paper in January 2021, v1.28.2 was the current version of Docker Compose; as of the finalization of this paper in July 2021, v1.29.2 is the current version of Docker Compose. The latest OpenSSL tarball source files can be found here: https://www.openssl.org/source/; this page asserts that the latest stable version is the 1.1.1 series, which is the OpenSSL.org's Long Term Support (LTS) version, which is slated to be supported until 11 September 2023. For convenience, please refer to Table 2 below. The current version (v1.29.2) is bolded for convenience as are the referenced v1.28.0, v1.28.2, and v1.26.2.

TABLE II. DOCKER COMPOSE RELEASE VERSIONS WITH DATES

Release Version	Release Date
1.29.2	2021-05-10
1.29.1	2021-04-13
1.29.0	2021-04-06
1.28.6	2021-03-23
1.28.5	2021-02-26
1.28.4	2021-02-18
1.28.3	2021-02-17
1.28.2	2021-01-26
1.28.0	2021-01-20
1.27.4	2020-09-24
1.27.3	2020-09-16
1.27.2	2020-09-10
1.27.1	2020-09-10
1.27.0	2020-09-07
1.26.2 (Case #3 from Table I)	2020-07-02

Source: https://docs.docker.com/compose/release-notes/

The Docker Compose bundling issue has persisted for quite some time — as far back as 25 June 2015 (e.g., https://github.com/docker/compose/issues/1601) (e.g., [22]). The bundling issue has also been noted in Linux binaries (as well as Mac binaries). Despite the delineated bundling issue, many vendors, who bundle OpenSSL, "will selectively retrofit urgent fixes to an older version of code, in order to maintain [Application Programming Interface] API stability/predictability. This is especially true for 'long-term release' and appliance platforms" [23]. This trend is significant, for in the current environment, 5G App developers are actively leveraging the faster speeds and lower latencies to innovate and release next-generation Augmented Reality (AR) and other immersive experience Applications (a.k.a., apps), which are used by healthcare providers (e.g., collaborative apps for sharing high-resolution medical images for telemedicine triaging), manufacturers (e.g., inspection apps to assist in identifying defects more quickly), and others. As the involved industries require mission-critical QoS, API stability/predictability is paramount. Therein resides the dilemma; many vendors have utilized older software release versions to maintain the requisite stability/predictability. However, this has resulted in the described bundling issues, which contain — in many cases — deprecated versions of various constituent components of the bundle.

For the architectural stack scrutinized, it was found that the containerization implementation, for an extended period of time (until flagged by the author), contained "High Severity" versions of OpenSSL and other components utilized for the involved fuzzing modules. The implication is that the entire fuzzing TTP could have been compromised, and discovered vulnerabilities within the fuzzing target might have been non-logged; the significance of logs has been previously illuminated by various reports, such as the "Verizon Data Breach Report: Detective Controls by Percent of Breach Victims" (which highlighted the fact that 71% of breach victims relied predominantly upon System Device Logs, 20% for Automated Log Analysis, and 11% for Log Review Process), and extensively discussed in the literature [24]; in essence, logs remain a mainstay of a cyber framework.

C. *Further Legacy Vulnerabilities within the Fuzzing Module Supply Chain*

Given the possibility of discovered vulnerabilities being non-logged, among other paradigms, the notion of a script, which serves to ensure the bundling of an apropos (e.g., patched) OpenSSL version, as just one example of a bundled component, and the notion of a vetted installer (e.g., PyInstaller) that links to an apropos OpenSSL version dynamically, have been discussed extensively [25]. The notion of such a script to check for CVEs on an ongoing basis seems quite simple; however, because Docker

containers are essentially inert when not in use, a systematic evaluation of what CVEs are present is non-trivial. In addition, as the constituent components of a bundle are ever-evolving, and as certain sub-components become deprecated, legacy issues are ongoing. For example, as each network operator has its own 5G/B5G/6G network roll-out plan, at least for the interim, each operator's network is actually a patchwork of 2G, 3G, 4G, and 5G networks. Since 5G networks will, for the foreseeable future, continue to interoperate with legacy networks, they will be subject to prototypical legacy vulnerabilities (e.g., spoofing, denial-of-service, etc.). By way of example, network operators will continue to rely upon General Packet Radio Service (GPRS) Tunneling Protocol (GTP) (designed to facilitate data packets moving backing and forth between the wireless networks of different operators, such as when a user is roaming). Furthermore, in addition to fuzzing modules within the 5G/B5G/6G ecosystem containing vulnerable constituent components within their bundles, fuzzing modules within the 4G ecosystem, etc. have also been found to contain the described vulnerabilities. Hence, even if the 5G/B5G/6G fuzzing modules are robustly scrutinized, the fuzzing modules (a.k.a., fuzzers) of the underlying patchwork (i.e., 2G, 3G, 4G) need commensurate scrutinization.

The implications of this underlying legacy patchwork are profound, particularly in the matter of mission-critical QoS and 5G-enabled software defined networks, such as vehicular networks (5G-SDVN), which have been burgeoning (to support the ever-growing autonomous vehicle market). However, the cyber security issues surrounding 5G-SDVN are complex not only because of the underlying legacy patchwork issues, but also because conventional fuzzers are comprised of varied classes — each with certain advantages/disadvantages: black-box (coverage information is not considered and inputs are randomly generated), white-box (coverage is maximized by considering the data structure/logical constraints of the internal implementation, and inputs are crafted, but the time requirement is higher), and grey-box (in contrast to black-box fuzzing, coverage information is considered, but perhaps not to the extent of white-box fuzzing so as to save on time). Among other distinctions, coverage-based evaluation metrics are difficult to ascertain as it is difficult to determine "which parts of a [software] program a fuzzer actually visits and how consistently it does so," and the lack of a standardized methodology for evaluating coverage remains a challenge [26].

IV. A PROSPECTIVE MITIGATION PATHWAY

As discussed, white-box fuzzers produce quality inputs, but the computational overhead is much higher, while black-box fuzzers that focus upon random mutation have computational overhead that is much lower, but have difficulty producing quality inputs [27]. Even state-of-the-art fuzzers are sub-optimal at discerning " 'hard-to-trigger' bugs in applications that expect highly structured inputs" [28]. While grammar-based fuzzers (capable of generating syntactically correct inputs) can indeed be effective, the computational overhead is high and feature engineering is required.

To address these challenges, in this paper, we present a bespoke grey-box concolic fuzzing module, which is comprised of four differing bespoke grey-box concolic fuzzers. A primary grey-box concolic fuzzer is able to achieve higher coverage (on average) and able to more robustly discern which parts of a software program it visits and how consistent it is in doing so; the primary fuzzer is complemented by a secondary fuzzer, which utilizes different classes (from that of the primary fuzzer) for mutating a seed. Together, they comprise an aggregate fuzzer for the test target; this is an improvement upon prototypical fuzzers, which might simply report on the number of lines or basic blocks (a straight-line code sequence that has no branches except to the entry and from the exit), but does not indicate whether it missed visiting certain sectors of the software program. By having a myriad of distinct and disparate classes (e.g., Class 1a Family, Class 2a Family, etc) for mutating a seed and by utilizing differing seed schedules (i.e., varying distributions of the fuzzing time spent among the seeds) for coverage (e.g., Class 1b Seed Schedule, Class 2b Seed Schedule, etc.), the primary and secondary fuzzers constitute a complementary set. In turn, this set is fuzzed by tertiary and quaternary grey-box concolic fuzzers, so as to mitigate against inadvertently not discerning vulnerabilities within the primary and secondary fuzzers themselves. The utilization of distinct and disparate tertiary and quaternary fuzzers (which utilize different classes for mutating a seed as well as seeding schedules) increases the likelihood of increased coverage (on average). The described paradigm is shown in Figures 1 and 2 below, wherein the entirety of Figure 1 is situated within the yellow box of Figure 2, which is roughly based upon [29].

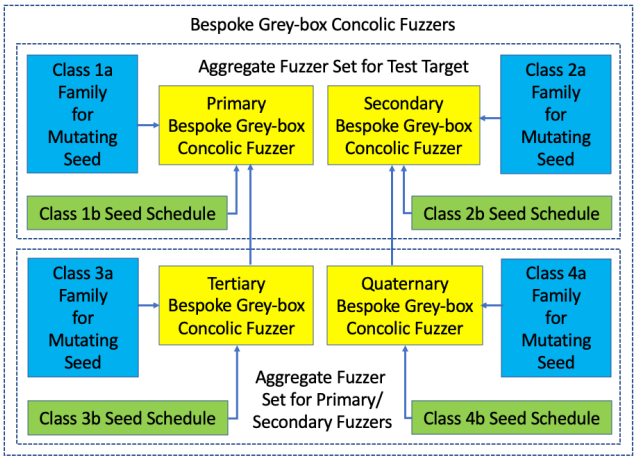

Figure 1. Bespoke Grey-box Concolic Fuzzers

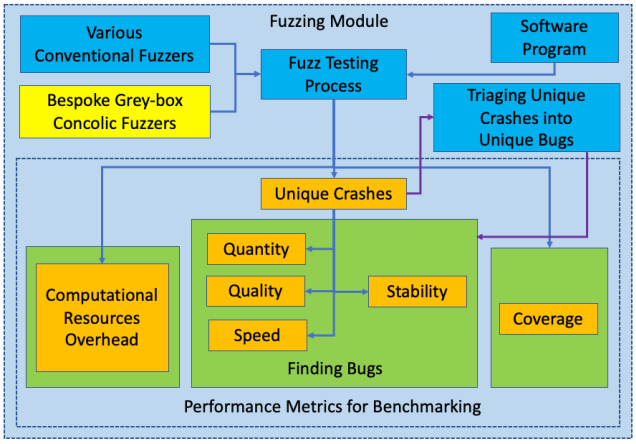

Figure 2. Grey-box Concolic Fuzzing Module

The coverage feedback derived by both primary and secondary fuzzers help to operationalize an underpinning numerical stability-centric Deep [Learning] Convolutional Generative Adversarial [Neural] Network (DCGAN)-facilitated Enhanced Context Module (ECM). The ECM is comprised of a Numerical Stability-Centric Module (NSCM), which in turn contains two Convolutional Adversarial Neural Networks (CANNs), each with a different implementation and version of PyTorch; PyTorch v0.4.1 (more numerically stable) is used in CANN #1, and PyTorch v1.7.0 (less numerically stable) is used CANN #2. The ECM's NSCM, which is shown in light purple in Figure 3 below, directs the aggregate fuzzer set for the test target (a.k.a., directed grey-box concolic fuzzing) to progress more rapidly into deeper code sectors.

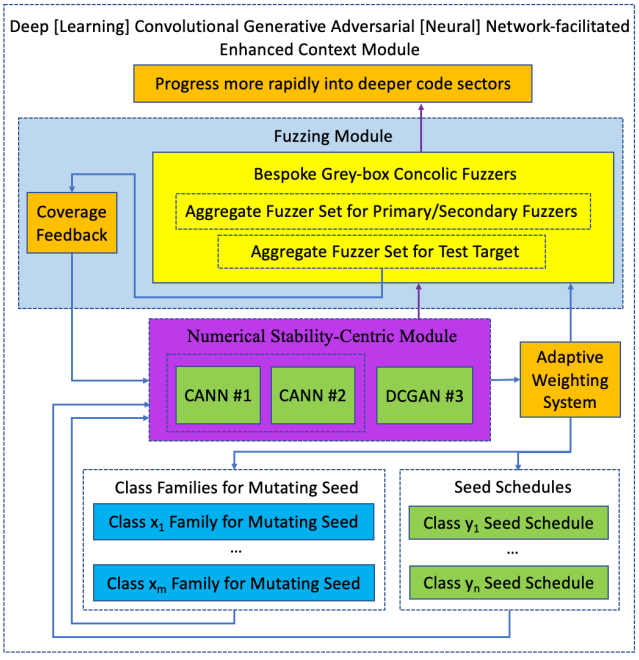

Figure 3. Numerical Stability-Centric Module (NSCM) which Leverages Coverage Feedback and an Adaptive Weighting System for Assigning Relative Weights to Fuzzer Seed Mutations/Schedules

The NSCM architecture is loosely based upon work that had been previously articulated in [8], but this version leverages coverage feedback and utilizes an adaptive weighting system to assign a relative weight to the fuzzer seeds *rw(fs)* that have the potential to achieve greater coverage, as shown in (1) below,

$$rw(fs) = 1/fr(pw(fs))^e, \quad (1)$$

where *pw(fs)* is the pathway identifier selected by *fs*, *fr(pw)* is the frequency at which the pathway *pw* is actually selected by the generated inputs, and *e* is a given exponent. The schedule is dynamically updated depending upon the frequency for which each pathway *fr(pw)* is utilized.

Overall, the feedback orientation (e.g., coverage, schedule) provided by the ECM well lends to a runtime coverage feedback paradigm, which in turn lends to, interestingly, enhanced thread-context. In this way, feedback can be operationalized, via the dynamic seed selection, mutation, weighting, and ensuing execution so as to better discern vulnerabilities within even a multi-threaded context [30]. With regards to the inner workings of the ECM of Figure 3, the entirety of Figure 1 is situated within the yellow box of Figure 2, the entirety of Figure 2 is situated within the light blue box of Figure 3, and the various components are described as follows.

A. *Grey-box Concolic Fuzzing Module*

Standard performance metrics for assessing the Grey-box Fuzzing Module (GFM) include, but are not limited to: (1) Unique Crashes, (2) Computational Resources Overhead, and (3) Coverage. First, Unique Crashes are further translated into unique bugs; it can also further be subdivided into quantity of unique bugs found, quality of bugs found (e.g., common or rare, CVE severity level, etc), speed at which bugs are found (i.e., Time-to-Exposure or TTE), and performance stability for finding bugs (i.e., Relative Standard Deviation or RSD for the number of unique bugs found for the fuzzing iterations; a lower RSD implies higher performance stability) [29]. Second, Computational Resources Overhead reflects the computing resources required by the involved fuzzing paradigm; if a particular paradigm is effective at finding more bugs, but the resources consumed are disproportionate, then that must be taken into consideration. Third, Coverage signifies the intrinsic ability of the fuzzer for exploring new pathways; this is of import for pursuing the desired pathway, which leads to the vulnerable code (i.e., quality versus quantity).

B. *Bespoke Grey-box Concolic Fuzzers*

The GFM is, in essence, powered by an amalgam of four bespoke grey-box concolic fuzzers: primary, secondary, tertiary and quaternary. Each utilizes distinct class families for mutating the seed as well as a multi-objective optimization seed schedule (whose aim is to decrease the TTE). Furthermore, the seed schedule is further subdivided into seed pool states (e.g., wide area search, targeted search, and assessment). First, for the wide area search, the aim is to seek high-promise pathways. Second, for the targeted search,

 ISBN: 978-1-61208-893-8

the aim is to allocate increased weighting (via the Adaptive Weighting System) towards those identified high-promise pathways. Third, for the assessment, the aim is to ascertain and assess promising seeds. The primary and secondary fuzzers form an aggregate fuzzer set for the test target. The tertiary and quaternary fuzzers are tasked with fuzzing the primary and secondary fuzzers. This amalgam of fuzzers comprise the Fuzzing Module.

C. Deep Learning Convolutional Generative Adversarial Neural Network-facilitated Enhanced Context Module

The Enhanced Context Module (ECM) encompasses the Fuzzing Module; its purpose is to serve as a macro feedback loop. In essence the ECM selects a seed, mutates it, and serves it as input to the test target. If the input causes a crash, it will be added to the ECM's crash set. Alternatively, if the input segues to new coverage, it will be added to the search seed pool. In turn, the Fuzzing Module derives Coverage Feedback from the Aggregate Fuzzer Set for the Test Target. This then serves an input to the NSCM, which processes the information and informs the Adaptive Weighting System, which dynamically weights the Class Families for Mutating Seed and Seed Schedules. This should segue to a more optimal Seed Schedule for decreasing TTE as well as RSD; the resulting lower RSD/higher performance stability can be attributed to the NSCM and Adaptive Weighting System.

Given the page limitations of this paper, future work will include more quantitative comparison with various CGFs, such as AFLGo (generates input to reach specified target test sectors) [31], FairFuzz (discerns rare branches within the target test and adapts mutation strategies to enhance coverage) [32], MOPT (utilizes Particle Swarm Optimization or PSO to optimize the mutation schedule and reduce TTE) [33], etc.

ACKNOWLEDGMENT

This research is supported by the Decision Engineering Analysis Laboratory (DEAL), an Underwatch initiative. This is part of a VT white paper series on 5G-enabled defense applications, via proxy use cases, to help inform Project Enabler.

V. CONCLUSION

In a not insignificant portion of the 5G/B5G/6G ecosystem cyber cases, the more serious security problems are implementation imperfections (e.g., network protocols); these constitute attack surface areas, which are often exploited. In the case for which 5G/B5G/6G protocols are still evolving and being defined, these implementation imperfections can be amplified. Conventional software cyber security frameworks, which involve code review, risk analysis, penetration testing, and prototypical fuzzing, do not currently suffice for robustly addressing a domain space, such as the 5G/B5G/6G ecosystem, wherein the protocols are evolving at a rapid pace. Indeed, prototypical fuzzers are challenged by the coverage issue, and conventional CGFs are as well. In an endeavor to provide a mitigation pathway, this paper presented an architectural stack comprised of a sequence of bespoke grey-box concolic fuzzers; as the primary grey-box concolic fuzzer (used against the testing target) is designed to work in conjunction with a secondary grey-box concolic fuzzer, so as to better mitigate against coverage issues (e.g., increasing the probability of visiting certain blocks/lines of code of the software program), and both are fuzzed by tertiary and quaternary grey-box concolic fuzzers (which utilize different classes for mutating a seed as well as seeding schedules), so as to mitigate against inadvertently not discerning vulnerabilities within the primary and secondary fuzzers themselves, the likelihood of increased coverage (on average) is enhanced. The feedback for coverage and adaptive weighting, as well as seed scheduling schemas, contribute to the efficacy. Future work will involve more quantitative experimentation.

REFERENCES

[1] "5G Uptake Progresses Across the Globe: Global 5G Connections Reach 298M in Q1 2021, 5G Connections Added in 2021 Nearly Triple that of 2020, 172 5G Commercial Networks Deployed Worldwide," 5G Americas, June 2021.

[2] H. Remmert, "5G Applications and Use Cases," Digi, November 2019.

[3] K. Fysarakis et al., "A Reactive Security Framework for operational wind parks using Service Function Chaining,"2017 IEEE Symposium on Computers and Communications (ISCC), 2017, pp. 663-668, doi: 10.1109/ISCC.2017.8024604

[4] H. Xu, M. Dong, K. Ota, J. Wu, and J. Li, "Toward Software Defined Dynamic Defense as a Service for 5G-Enabled Vehicular Networks," 2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2019, pp. 880-887, doi: 10.1109/iThings/GreenCom/CPSCom/SmartData.2019.00158

[5] C. Cimpanu, "South Korean researchers apply fuzzing techniques to LTE protocol and find 51 vulnerabilities, of which 36 were new," Zdnet, March 2019.

[6] "Department of Defense (DoD) 5G Strategy (U)," Accessed on: Aug 27, 2021. [Online]. Available: https://www.cto.mil/wp-content/uploads/2020/05/DoD_5G_Strategy_May_2020.pdf.

[7] J. Yli-Kaakinen, T. Levanen, M. Renfors, M. Valkama, and K. Pajukoski, "FFT-Domain Signal Processing for Spectrally-Enhanced CP-OFDM Waveforms in 5G New Radio," 2018 52nd Asilomar Conference on Signals, Systems, and Computers, 2018, pp. 1049-1056, doi: 10.1109/ACSSC.2018.8645100

[8] S. Chan, M. Krunz, and B. Griffin, "AI-based Robust Convex Relaxations for Supporting Diverse QoS in Next-Generation Wireless Systems," Proc. of the IEEE ICDCS Workshop - Next-Generation Mobile Networking and Computing (NGMobile 2021), July 2021, pp. 1-8.

[9] B. Han, J. Lianghai, and H. D. Schotten, "Slice as an Evolutionary Service: Genetic Optimization for Inter-Slice Resource Management in 5G Networks," in IEEE Access, vol. 6, pp. 33137-33147, 2018, doi: 10.1109/ACCESS.2018.2846543.

[10] L. U. Khan, I. Yaqoob, N. H. Tran, Z. Han and C. S. Hong, "Network Slicing: Recent Advances, Taxonomy, Requirements, and Open Research Challenges," in IEEE Access, vol. 8, pp. 36009-36028, 2020, doi: 10.1109/ACCESS.2020.2975072

 ISBN: 978-1-61208-893-8

[11] A. Farrel, “Recent Developments in Service Function Chaining (SFC) and Network Slicing in Backhaul and Metro Networks in Support of 5G,” 2018 20th International Conference on Transparent Optical Networks (ICTON), 2018, pp. 1-4, doi: 10.1109/ICTON.2018.8473624.

[12] A. J. Gonzalez et al., “The Isolation Concept in the 5G Network Slicing,” 2020 European Conference on Networks and Communications (EuCNC), 2020, pp. 12-16, doi: 10.1109/EuCNC48522.2020.9200939

[13] B. L. Parne, S. Gupta, K. Gandhi and S. Meena, “PPSE: Privacy Preservation and Security Efficient AKA Protocol for 5G Communication Networks,” 2020 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS), 2020, pp. 1-6, doi: 10.1109/ANTS50601.2020.9342780.

[14] J. Knudsen, “How to cyber security: containerizing fuzzing targets,” Synopsys, February 2021.

[15] R. Nagler, D. Bruhwiler, P. Moeller, and S. Webb, “Sustainability and Reproducibility via Containerized Computing,” 2015, pp. 1-2, arXiv:1509.08789 [cs.SE].

[16] "Use Kaniko to build Docker images," Accessed on: Aug 27, 2021. [Online]. Available: https://docs.gitlab.com/ee/ci/docker/using_kaniko.html.

[17] "Octave-x11-novnc-docker," Accessed on: Aug 27, 2021. [Online]. Available: https://github.com/epfl-sti/octave-x11-novnc-docker

[18] “OpenSSL Release Security Update [08 December 2020],” Accessed on: Aug 27, 2021. [Online]. Available: https://www.openssl.org/news/secadv/20201208.txt.

[19] “OpenSSL Release Security Update [25 August 2021],” Accessed on: Jul 19, 2021. [Online]. Available: https://us-cert.cisa.gov/ncas/current-activity/2021/08/25/openssl-releases-security-update

[20] "Update the bundled OpenSSL version #1834," Accessed on: Aug 27, 2021. [Online]. Available: https://github.com/docker/compose/issues/1834.

[21] "[OpenSSL] Vulnerabilities," Accessed on: Aug 27, 2021. [Online]. Available: https://www.openssl.org/news/vulnerabilities.html

[22] "Openssl version used is insecure #1601," Accessed on: Aug 27, 2021. [Online]. Available: https://github.com/docker/compose/issues/1601

[23] “Heartbleed: how to reliably and portably check the OpenSSL version,” Accessed on: Jul 19, 2021. [Online]. Available: https://serverfault.com/questions/587324/heartbleed-how-to-reliably-and-portably-check-the-openssl-version text 10

[24] S. Chan, “Prototype Orchestration Framework as a High Exposure Dimension Cyber Defense Accelerant Amidst Ever-Increasing Cycles of Adaptation by Attackers,” The Third International Conference on Cyber-Technologies and Cyber-Systems, November 2018, pp. 28-38.

[25] “Dynamically linking OpenSSL #1304,” Accessed on: Jul 19, 2021. [Online]. Available: https://github.com/bitshares/bitshares-core/issues/1304

[26] L. Simon and A. Verma, “Improving Fuzzing through Controlled Compilation,” 2020 IEEE European Symposium on Security and Privacy (EuroS&P), 2020, pp. 34-52, doi: 10.1109/EuroSP48549.2020.00011.

[27] P. Chen and H. Chen, “Angora: Efficient Fuzzing by Principled Search,” 2018 IEEE Symposium on Security and Privacy (SP), 2018, pp. 711-725, doi: 10.1109/SP.2018.00046.

[28] X. Wang, C. Hu, R. Ma, B. Li and X. Wang, "LAFuzz: Neural Network for Efficient Fuzzing," *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2020, pp. 603-611, doi: 10.1109/ICTAI50040.2020.00098.

[29] Y. Li et al., “UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers,” 2020, pp. 1-18, arXiv:2010.01785 [cs.CR]

[30] H. Chen et al., “MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs,” 2020, arXiv:2007.15943v1 [cs.SE].

[31] M. Bohem, V. Pham, M. Nguyen, and A. Roychoudhury, “Directed Greybox Fuzzing,” ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 2329-2344.

[32] C. Lemieux and K. Sen, “Fairfuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage,” 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 475-485.

[33] C. Lyu, S. Ji, C. Zhang, Y. Li, W. Lee, Y. Song, and R. Beyah, “MOPT: Optimized Mutation Scheduling for Fuzzers,” 28th USENIX Security Symposium, 2019, pp. 1949-1966.

 ISBN: 978-1-61208-893-8