Internet of Things 24 (2023) 100916

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

Moving Target Defense for the cloud/edge Telco environments

Pedro Escaleira [*], Vitor A. Cunha, Diogo Gomes, João P. Barraca, Rui L. Aguiar

Instituto de Telecomunicações, Campus Universitário de Santiago, Aveiro, 3810-193, Portugal
Department of Electronics, Telecommunications and Informatics, University of Aveiro, Campus Universitário de Santiago, Aveiro, 3810-193, Portugal

A R T I C L E   I N F O

Keywords:
Moving Target Defense (MTD)
MTD as a Service (MTDaaS)
Network Functions Virtualization (NFV)
Cloud security
Security as a Service (SECaaS)
Zero-day vulnerabilities

A B S T R A C T

The Internet of Things (IoT) paradigm has been one of the main contributors, in recent years, to the growth in the number of connected equipment. This fact has predominantly contributed to IoT being constrained by the 5th Generation Mobile Network (5G) progress and the promises this technology brings. However, this can be a double-edged sword. On the one hand, it will benefit from those progresses, but on the other, it will also be impacted by any security risk associated with 5G. One of the more serious security problems associated with it is the new wave of virtualization and *softwarization* of networks and analogous appliances, brought to light by paradigms such as Network Functions Virtualization (NFV) and Multi-access Edge Computing (MEC). Considering these predicaments, we propose a state-of-the-art Moving Target Defense (MTD) approach that defends Cloud-based Network Functions (CNFs) launched within MEC and NFV environments. Furthermore, our mechanism follows the famous Everything as a Service (XaaS) ideology, allowing any CNF provider to use this protection system, working agonistically. In the end, we created a Proof of Concept (PoC) of our proposed methodology, which we then used to conduct an extensive practical security analysis against the multiple phases of the Intrusion Kill Chain. Our final results have proven that our MTD as a Service (MTDaaS) approach can effectively delay and, in some cases, stop an attacker from achieving its objectives when trying to attack a CNF, even if the related vulnerability is a zero-day.

1. Introduction

In recent years, the proliferation of Internet of Things (IoT) devices has led to challenging amounts of produced data that require improved processing. Traditionally, IoT systems would send it to the cloud for further handling, where processing is cheaper and easier to scale. However, security, privacy, and latency concerns require different approaches that keep data and its processing closer to where it is produced. Therefore, recent research has been developing new technologies that extend the cloud to the network's edge near the equipment creating the data [1]. One of these approaches is Edge Computing, which promises to reduce service latency and improve the Quality of Experience and of Service (QoE and QoS) by having the computing power, storage, and bandwidth closer to these devices. Therefore, Cloud and Edge Computing, and their interplay, are essential in achieving the IoT's data processing necessities.

On the communication side, IoT will significantly benefit from the 5th Generation Mobile Network (5G) and beyond-related technologies [2]. Primarily, it will provide a high data rate, very low latency, reliability, resilience, and highly scalable and fine-grained networks, among other benefits. 5G will be built upon several pillars to achieve these disruptive requirements, including

* Corresponding author at: Instituto de Telecomunicações, Campus Universitário de Santiago, Aveiro, 3810-193, Portugal.
E-mail addresses: escaleira@av.it.pt (P. Escaleira), vitorcunha@av.it.pt (V.A. Cunha), dgomes@ua.pt (D. Gomes), jpbarraca@ua.pt (J.P. Barraca), ruilaa@ua.pt (R.L. Aguiar).

https://doi.org/10.1016/j.iot.2023.100916
Received 31 May 2023; Received in revised form 12 August 2023; Accepted 15 August 2023
Available online 22 August 2023

the network's *softwarization* and virtualization [3]. These are realized through many key technologies, such as Network Functions Virtualization (NFV) and Multi-access Edge Computing (MEC) [4].

NFV is a recent paradigm in telecommunications that aims to decouple Network Functions (NFs), such as routers or switches, from their physical equipment and virtualizes them in generic cloud servers [5]. This characteristic allows for separating each NF from its tightly coupled hardware and virtualizing it into Virtual Network Functions (VNFs), which can be executed in generic hardware as Virtual Machines (VMs) or even containers. It also facilitates flexible deployment and dynamic scaling of NFs. This new paradigm has gathered the attention of both standardization bodies and academia. In the first case, the European Telecommunications Standards Institute (ETSI) has led NFV's conceptualization and standardization since 2012 [6–8]. In the 5G-IoT sphere, Li et al. [2] defend that NFV will grant scalable and flexible NFs so that the network can adapt according to the needs of the IoT devices.

As for MEC, it is the ETSI response for Edge Computing adapted for the telecommunications world. Its central idea is relocating computation from the cloud to the network's edge. This characteristic is essential since User Equipments (UEs) are placed within the Radio Access Network (RAN). Therefore, it is at this location where the equipment that produces the data is located. Theoretically, this closeness between servers and appliances decreases latency and increases bandwidth while minimizing traffic congestion and communications costs associated with data transmission in the network's backbone [9,10].

Despite these advantages, new security risks and considerations come with this network paradigm shift [11–13]. They are still in the early stages of development, are partly associated with borderless computing, and depend on many moving parts. These peculiarities make in-depth defense challenging, where gathering all their security issues is difficult. Consequently, not only do these security risks end up impacting 5G-IoT scenarios, but introducing IoT within NFV and MEC environments institutes new attack surfaces towards those systems. With this in mind, academia and ETSI have been preoccupied with studying these risks to create protective mechanisms against them [12–16].

Nonetheless, software vulnerabilities are among these new systems' most worrying security risks [11]. Accordingly, they can be subdivided into unknown or zero-day vulnerabilities and known vulnerabilities. Although we already have well-known best practices for protecting and securing a system against the former, the same cannot be said for the latter. Most zero-day protection-related research has focused on previous vulnerability detection to defend the system against it. These mechanisms rely on data obtained concerning past attacks, for example, to create anomaly detection or signature-based systems [17]. This trait led to many works tackling this issue using Machine Learning (ML)-based methods [18]. However, these kinds of protection heavily rely on the system's accuracy, which is affected by the training data [17]. Since zero-day vulnerabilities are characterized by their unpredictable nature, the reliability of a system, as one based on ML, that detects them based on past events might be questionable. Moreover, these ML-based methodologies are reactive, meaning an attack must happen for an action against them to occur.

Considering these issues, Moving Target Defense (MTD) emerges as a potential solution to protect systems against zero-day vulnerabilities. It is a proactive defense methodology that moves a system's attack surface to increase its assault difficulty, continuously disorienting any potential attacker. Its proactivity means that MTD constantly protects the system, regardless of whether an attack is happening. Furthermore, this methodology also does not rely on datasets and long training sessions, as ML usually does, and consequently, does not depend on observations of past events to be effective.

In light of this methodology, this paper proposes a new MTD-based protection technique that targets these new 5G enabler technologies. More specifically, we propose an innovative defense system that a Telecommunications Operator can offer its clients as a security product to protect their deployed services. In other words, any VNF provider that uses an Operator's infrastructure to launch their products can use this methodology to enhance their defense. More specifically, we targeted the protection of Cloud-based Network Functions (CNFs) based on Operating System (OS)-level instead of hardware-level virtualization. Therefore, they are more lightweight than VNFs, consuming fewer resources and usually faster than their counterparts [19]. Consequently, we defend that they will have a prominent role in the NFV world, notably in MEC in NFV scenarios, where computing resources are scarcer [20].

Furthermore, since our system can be offered as a security product to defend any CNF, we named it MTDaaS, following the Everything as a Service (XaaS) ideology. Therefore, our MTDaaS is a novel Security as a Service (SECaaS) [21] strategy to enhance NFV-based ecosystem security. The following paragraphs condense these and other of this work's main novelties by area of applicability.

Solution's Flexibility: (i) As far as we know, we propose the first MTD system in the literature that can be provided as a plug-and-play module attached to the object of protection, i.e., provided as a security service; (ii) since we designed our main proposal towards the NFV ecosystem within a cloud environment, it can be easily generalized for any NFV-based domain, including MEC in NFV, and; (iii) our approach gives providers the power to easily define the MTD system properties, namely the mutation interval, redundancy level, diversity level, and application's attributes to shift.

Areas of applicability: (i) Since our MTD solution is aligned with the ETSI's NFV standards and has the potential to evolve to meet the MEC in NFV references, it is, as far as we can tell, the first work where MTD is used to protect CNFs and MEC Applications; (ii) in this work, we felt the need to propose the evolution of some of the NFV standards and the used Management and Orchestration (MANO) framework, which, in this last case, led to an open-source feature contribution to the ETSI OSM project, and; (iii) as NFV and MEC are some of the pillars that promise to bring 5G and beyond to life, our proposal emerges as a state-of-the-art mechanism to enhance the overall ecosystem security.

Cybersecurity significance: (i) This work's approach appears not only as a response against known vulnerabilities but also against zero-day or unknown vulnerabilities, against which there is not yet a precise response mechanism in the cybersecurity area, and; (ii) the system we introduce guarantees some level of protection in almost all Intrusion Kill Chain phases, which sets it apart from most MTD works, where authors usually focus on enhancing the security of one step at most.

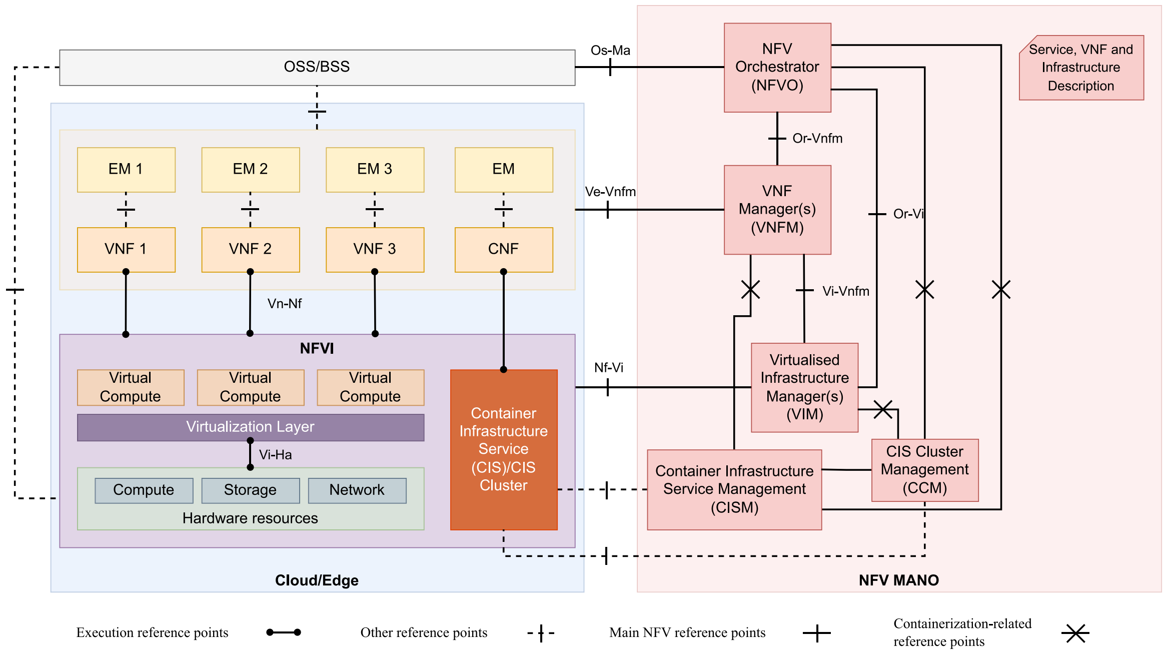

OSS/BSS
Os-Ma
NFV Orchestrator (NFVO)
Service, VNF and Infrastructure Description
Or-Vnfm
EM 1
EM 2
EM 3
EM
VNF 1
VNF 2
VNF 3
CNF
Ve-Vnfm
VNF Manager(s) (VNFM)
Or-Vi
Vi-Vnfm
Vn-Nf
NFVI
Virtual Compute
Virtual Compute
Virtual Compute
Virtualization Layer
Vi-Ha
Compute
Storage
Network
Hardware resources
Container Infrastructure Service (CIS)/CIS Cluster
Nf-Vi
Virtualised Infrastructure Manager(s) (VIM)
CIS Cluster Management (CCM)
Container Infrastructure Service Management (CISM)
Cloud/Edge
NFV MANO
Execution reference points
Other reference points
Main NFV reference points
Containerization-related reference points

Fig. 1. NFV architectural framework with support for CNFs.
Source: Adapted from [24].

Thorough practical evaluation: (i) The proposed system deviates from the norm of MTD works that stop at a purely mathematical or simulated analysis by effectively demonstrating the effects of the MTDaaS in protecting a real CNF in an NFV-based environment, and; (ii) we prove the system's effectiveness in delaying and confusing an attacker in each step of the Intrusion Kill Chain, except for the Weaponization and Delivery phases, when protecting an application with a zero-day vulnerability.

The rest of this document is organized as follows. We start by outlining some NFV background context and some of the most noteworthy MTD works in Section 2. Then, in Section 3, we present our MTDaaS solution from a theoretical standpoint. Subsequently, to properly assess the protection of our proposed system, we evaluate our MTD solution against each Intrusion Kill Chain phase in Section 4. Section 5 follows with an early performance analysis of our MTD system's impact at the management and user plane levels. We finally close this manuscript in Section 6 with a summary of our proposal, the results we achieved, lessons learned, and future directions.

2. Background and related work

The rapid growth of the IoT has, and will continue to, contribute to the growth of connected devices in various environments. These environments include Cloud and Edge Computing, which are particularly vulnerable due to their reliance on virtualized resources. As a result, there is a growing need for robust security mechanisms that can protect these computing platforms against cyber threats.

In this work, we have committed to secure fully virtualized elements, i.e., VNFs, and containerized elements, i.e., CNFs, alike. Therefore, we have followed the ETSI's NFV-MANO and MEC architectures [22,23] for the Telco world. Going further, we explore the MANO architecture adapted to work with CNFs in Section 2.1. Then, in Section 2.2, we will present the most notorious MTD works whose proposals could be readjusted for the scenario in hands.

2.1. Motivating framework

As we are proposing specific to 5G-IoT, which will rely heavily on NFV and MEC, we focused on creating a defense mechanism to protect CNFs. Therefore, we established our defense mechanism on the architectural frameworks ETSI standardized for these two paradigms. More concretely, we based our solution on the new NFV architectural framework supporting OS-level virtualization or containers [24], whose generic form can be analyzed in Fig. 1. Note, however, that this support is relatively recent, as ETSI originally designed this architecture to only support hardware-level virtualization. Nevertheless, the new support for CNFs allows the deployment of lightweight NFs. Furthermore, it is better aligned with the current interest in cloud-based applications, usually based on stateless applications.

The central entity of the NFV framework represented in Fig. 1 is the MANO. Its primary block is the NFV Orchestrator (NFVO), which, as the name implies, orchestrates the overall NFV ecosystem [22]. Then, it is also composed of one or multiple VNF

Managers (VNFMs) [25], which are pluggable modules that manage their attached VNFs' lifecycle. In the NFV framework, the central entity is the MANO, as represented in Fig. 1. As for the NFV Infrastructure (NFVI) and Container Infrastructure Service (CIS) virtualized resources, they are governed by the Virtualised Infrastructure Manager (VIM) and Container Infrastructure Service Management (CISM), respectively.

Furthermore, although the NFV-MANO is usually used to launch and manage VNFs in a cloud setting, it can be generalized to be applied in an edge environment, using an NFVI in an edge location. In those cases, we can launch MEC Applications as VNFs or CNFs. Therefore, they will be orchestrated and managed by the NFV-MANO. This enhanced NFV framework for MEC, first proposed by Sciancalepore et al. [26], has already been standardized by ETSI and can be further studied in the ETSI's MEC framework and reference architecture document [23]. Despite this document's current release not specifying the presented container's integration, MEC Applications might leverage from being deployed as CNFs instead of VNFs since computational resources are usually more scarce at the edge [20].

2.2. Moving target defense mechanisms

MTD was a concept first introduced at the *National Cyber Leap Year Summit* and summarized in the Networking and Information Technology Research and Development (NITRD) report on that event [27]. At the time, MTD was instigated as a proactive defensive security mechanism to continually move the attack surface of the protected system to confuse possible adversaries. Traditionally, attackers had time on their side, gathering the maximum amount of information about the targeted system, to create the most opportune attack delivery. With this novel technique, the adversary is constantly disoriented, and the information gathered during the reconnaissance of one system's state loses its value once a new state is set up. This way, the probability of an attack happening with success is diminished.

In recent years, we can narrow an MTD system definition to three base questions from the many proposed frameworks: what, how, and when to move the system [28,29]. The first refers to the system's property to be moved, such as the CPU architecture, the IP address where some server is listening, or the OS versions where an application is executed. Then, the how-to-move question is related to the decision on how to change those properties, and there are three options: shuffling, diversity, and redundancy. Finally, the when-to-move question relies on how the method chooses the best moment to change the system's properties. Therefore, there are two approaches of when-to-move: proactively, where the system moves between states from time to time, usually depending on some predefined mutation window period, and; reactively, where the MTD is dependent on detection mechanisms and only moves the system when an attack is detected.

Then, MTD can also be defined by its type, which depends on the movement target, i.e., the what-to-move question. The most predominant kind in the literature is Dynamic Networks which, as the name implies, focus on techniques that move network-related properties, such as the IP address or port where some service is listening. An example of such a study is the Time-based One-Time Password (TOTP) MTD, proposed by Cunha et al. [30], where the authors proposed an architecture that leveraged the widely used TOTPs algorithm to calculate the next port where the protected server was listening. Another interesting network-based MTD methodology was proposed by Sattar et al. [31]. While, as referred previously, most works in this area involve IP address and port hopping, this work presents a different target of movement: the time it takes for the server to send a response. This characteristic was considered an approach for dissolving slow-rate denial of service attacks aimed at firewalls, where an attacker can discover the rules by sending random packets and observing the response times. With this defense mechanism, the authors have proven that the attacker cannot guess these packets since the system waits for a random period to send all responses.

However, our work focused on another kind of MTD: Dynamic Platforms. This type is based on the movement of the platform-related attributes, such as the OS, software libraries, or CPU architecture. Therefore, proposed techniques in this area are usually based on migrating an application or service between hosts with different platform properties. Recent advances in virtualization techniques have made this possible. These techniques are hardware-level and OS-level virtualization [32]. The first, which targets the deployment of VMs, has the advantage of allowing migrating the application status but has the disadvantage of consuming more computational resources. On the other hand, OS-level virtualization, or containerization, is usually a better fit for stateless migration, as it has less support for state migration but is much more lightweight.

Consequently, works in this area can, for the most part, be categorized for the type of virtualization they employ. Although, as referred, OS-level virtualization is mainly associated with stateless migration, there are multiple proposals where authors present an MTD framework based on the stateful migration of containerized applications [33–35]. In this case, authors usually resort to one of two methods of maintaining the state. On the one hand, there is the method of taking memory dumps of the container's processes at some moment [33,34]. Then, this snapshot will be used the next time the container is launched. Therefore, processes can continue from the moment they are stopped. The most recognizable utility to achieve this is CRIU.[1] However, one of the biggest problems of such a method is the impossibility of allowing migration between machines with significantly different hardware, mainly in terms of different CPU architectures. On the other hand, some authors achieve state maintenance through checkpoints. This peculiarity is the case with the Trusted Dynamic Logical Heterogeneity System (TALENT) [35] system, where applications are compiled using a particular code compiler called Portable Checkpoint Compiler (PCC). This compiler will introduce multiple checkpoints throughout the original code. Then, when one of these applications is running, and the process reaches one of these checkpoints, its variables and other state-related data are saved in a checkpoint file. Therefore, when the application is migrated, the new process will use this

[1] CRIU's project page: https://criu.org/Main_Page.

file to obtain its current state. Despite this methodology having the benefit of functioning in any hardware and CPU architecture, as long as the compiler can compile for those architectures, it poses some other challenges and problems. First, it is not guaranteed that there is a PCC compiler for each and any possible programming language. Then, introducing new pieces of code within the original might introduce new and unexpected vulnerabilities, which would not be present if the programmer used “a normal compiler”.

A solution to all these problems would be to use hardware-level virtualization. However, this virtualization technique has three significant issues [32]. First, container images are more lightweight than VM images since they do not require packaging a whole OS. Therefore, migrating containers have less impact on the bandwidth, which in MEC systems, where hosts may be distant from one another, may be a necessary particularity to consider. Second, launching a container is faster than launching a VM for the same reasons, which may be problematic in proactive MTD systems that use relatively short time windows or in reactive MTD when an attack is detected. Finally, the execution of a VM also requires, in most cases, more compute resources, which might be scarce in environments such as the Edge.

Nevertheless, in addition to stateful applications, there are also works that addressed Dynamic Platforms MTD for stateless applications. For instance, Ahmed and Bhargava [36] proposed a proactive MTD system, where a pool of multiple VMs of the same application would be pre-launched in various hosts. These hosts would have different characteristics, such as the hardware, OS, and hypervisor software. In each mutation window, only one VM would have an interface attached, and when the mutation window ends, that interface would be detached from the previous VM and attached to another one. Therefore, it would be like the application moved between distinct hardware and software stacks. Then, this work was enhanced by Villarreal-Vasquez et al. [37], where the authors combined the proactive method with a reactive one, which used a machine learning system to detect potential attacks and move the system at those points as well. Thompson et al. [38] also proposed a similar approach using a pool of VMs, for which the corresponding application was mutated. However, this work had a different target of mutation: instead of the underlying software and hardware stack, the movement occurred between VMs with distinct running OSs, i.e., the application was packaged within particular VM images with different OSs. These authors later expanded the idea of this work [39], but instead of rotating among VMs, the movement transpired between distinct web server frameworks. In any case, all these works have one specificity in common: they all reduce the probability of fingerprinting separate stacks of a platform and increase the difficulty of an attacker taking advantage of specific vulnerabilities that might be present on those stacks.

Although these proposals, for the most part, could probably be adapted to enhance the protection of an NFV environment, there are not a lot of works that do so, with exceptions such as the one proposed by Cunha et al. [30]. Because of this, there are potential problems adapting these solutions to the NFV world that still need to be profoundly studied in the literature. One such issue is how the MTD logic could be integrated and work in parallel with the orchestration and management functionalities of the NFV-MANO. Therefore, our proposal not only concentrates on giving a new MTD methodology, focusing on an application’s version movement, but also on addressing these issues.

3. Proposed solution

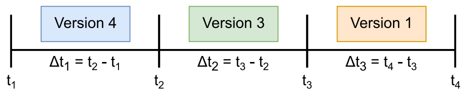
Considering our use case, where an Operator would offer the MTD system as a security product, the MTD methodology we conceptualized needed to work for any of the providers’ CNFs. In other words, the MTDaaS would need to be CNF agnostic, as customer-specific security solutions would be unjustifiable. Following a black box approach, this characteristic implied that the Operator would not change or observe any of the provided CNFs.

In this context, we start by outlining the MTD design elements in Section 3.1, following the key principles discussed by Cai et al. [28] and later emphasized by Cho et al. [29], answering the questions of what, when, and how to move the system. Then, considering this work’s defense context, we present the integration of our MTD system with the NFV framework in Section 3.2. To that section follows Section 3.3, which describes this system’s internal and external logic and interactions. Finally, Section 3.4 discusses some network considerations related to the proposed solution.

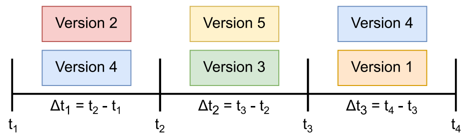
3.1. Design considerations

The first characteristic of an MTD system we need to consider is its type since it will affect the key design decisions. In our proposal, we followed the Dynamic Platforms kind, where the movement occurs at the platform level. Furthermore, the choice made in this proposal was to modify the application’s versions being executed at each moment. In other words, the proposed MTD methodology changes the platform where CNF is being run by modifying, at runtime, its version. This property implies that a provider intending to use this mechanism must supply at least two application versions. The theory behind the proposed approach is that software developers, by nature, can produce code with errors. These errors can lead to vulnerabilities, which in some scenarios, may “open a door” to their exploration. However, suppose an application has multiple implementations, ideally created by different developer teams. In that case, we can theoretically reduce the probability of an attacker taking advantage of a vulnerability in one of those versions by continually shifting the version being run.

Moreover, besides different implementations, clients can create multiple versions of the same application by changing, perhaps, the software libraries’ versions their application is using or even the associated container’s base image. These approaches also increase the system’s heterogeneity and reduce the probability of an attacker exploiting a version based on a library or base image that potentially can have an undisclosed vulnerability. Although these were some of the mutation parameters we thought of for the given scenario, we assume there are many more.



(a) Execution of one version per time window of an application managed by the proposed MTD Controller.



(b) Execution of two versions per time window of an application managed by the proposed MTD Controller.

Fig. 2. Graphic representation of the proposed MTD system controlled time windows.

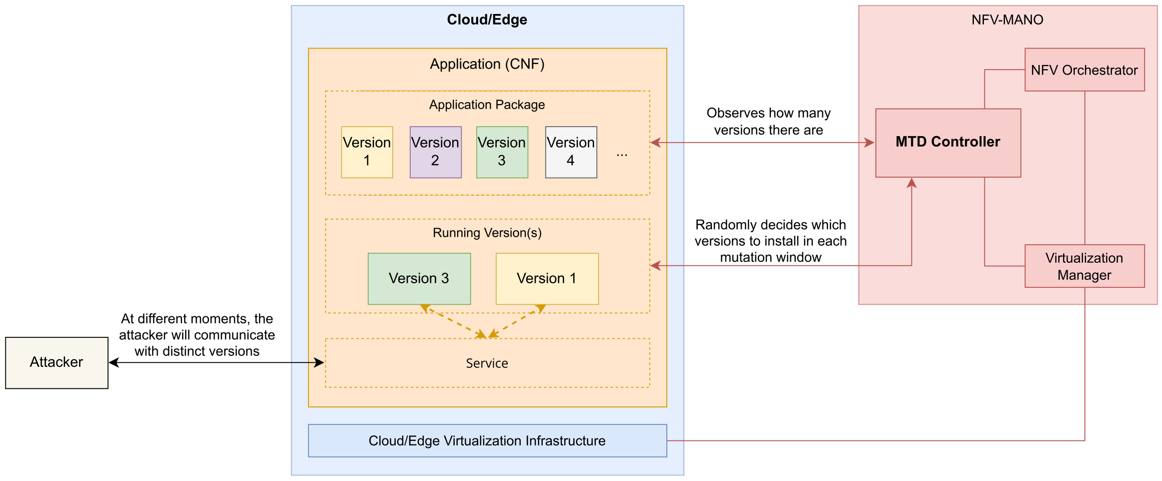

Cloud/Edge
Application (CNF)
Application Package
Version 1
Version 2
Version 3
Version 4
...
Running Version(s)
Version 3
Version 1
Service
Cloud/Edge Virtualization Infrastructure
Attacker
At different moments, the attacker will communicate with distinct versions
Observes how many versions there are
Randomly decides which versions to install in each mutation window
NFV-MANO
NFV Orchestrator
MTD Controller
Virtualization Manager

Fig. 3. Proposed MTD methodology high-level view.

Since the MTD system will move a CNF's versions over time, this methodology is limited to stateless applications. Although there is the option of saving the current running status of a container when stopping it, it is still a limited technology. Moreover, the described MTD poses a challenge when storing application statuses when its executing version is changed. Tools such as CRIU can keep a checkpoint of a container, but this checkpoint will not be valid for another implementation, even if the same application. Although there is the possibility of having applications that save their status on disk or in a database, to which we think this method would still be able to function correctly, the effects of this methodology would have on such scenarios needs further study in future works.

Regarding when-to-move the system's parameters, another key design of an MTD system, the choice was to follow a Proactive MTD methodology to offer a higher level of defense since this method considers that the system can constantly be under attack. Finally, we applied all three how-to-move-related techniques. First, shuffling and diversity are present. The former is because, in two distinct moments, the application's version running is different, and the latter is due to the presence of multiple application versions instead of just one. As for redundancy, we adjusted our technique to allow running multiple versions simultaneously, as will be explained further.

Furthermore, Fig. 2 graphically expresses the execution flow of the discussed system. In Fig. 2(a), we represented the case where only one version is executed simultaneously, i.e., the scenario where redundancy of one would be used. In contrast, Fig. 2(b) represents the scenario where redundancy is two. As observed, each mutation window exists for a well-defined period. When this period ends, the MTD system arbitrarily decides a new CNF version (or two new versions, if the redundancy is two) to be executed in the new mutation window.

Everything considered, we present in Fig. 3 a high-level sketch of our MTD methodology base idea. To achieve all the necessary MTD logic, we envision the necessity of an MTD Controller. This Controller will automatically decide when a mutation window needs to be launched or to end and will have an overall view of all the application's existing versions. On its end, this application, which in our scenario is a CNF, will be deployed in a cloud or edge server, depending on the targeted environment. On each mutation window, the MTD Controller will randomly decide, from all existing versions, the one or ones that will be executed depending on the redundancy level. Finally, depending on the moment an attacker tries to communicate with the application, it will communicate with a different version. This characteristic will potentially confuse the intruder while it tries to gather information about the application and later when trying to intrude into the system, as the version it is trying to attack might not be available.

Considering that CNFs were the target of protection, launched, managed, and orchestrated in the scope of NFV, we decided a VNFM would conduct the MTD-related decisions. According to the ETSI's report on the NFV-MANO framework [22], a VNFM is a component attached to a VNF responsible for its lifecycle management. Therefore, we believe a specially crafted VNFM can conduct all the MTD decisions related to each CNF. This VNFM, corresponding to the presented MTD Controller presented below, can trigger

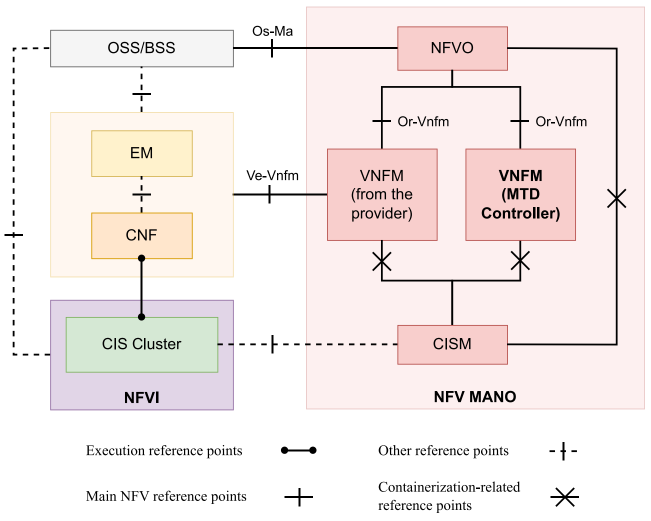


Fig. 4. Integration of the proposed MTD Controller within the NFV architecture.

or stop the execution of a CNF-specific version at some particular moment through its lifecycle control over that CNF. The following sections will discuss the integration between the MTD Controller and the NFV framework and its functioning.

3.2. Integration with the NFV-MANO framework

As mentioned below, the MTD method to follow in this work would lead to creating an MTD Controller, which interacts with the NFV-MANO, to manage the corresponding CNF under protection. Therefore, as depicted in Fig. 4, this Controller was considered a VNFM, which will be connected to the NFVO and the CISM, managing part of the CNF's lifecycle. Consequently, any Operator client wanting to use this defense mechanism will have a Controller MTD VNFM attached to its CNF.

Besides the MTD Controller VNFM, there is also another VNFM represented in the schema of Fig. 4. The client can provide this VNFM when its CNF has a more complex lifecycle which will be executed alongside the MTD Controller VNFM. An important detail to consider is that this architectural decision of combining multiple VNFMs to manage the same VNF or CNF is something that ETSI does not contemplate in its NFV-related references. In the NFV architectural options report [25], ETSI extensively discussed two architectural options related to VNFMs. The first is the possibility of using a generic VNFM to manage multiple VNFs. The other is where a provider can create a proprietary VNFM, distinct from other providers' VNFMs. However, in all these discussions, the VNFM was always seen as managing just one or multiple VNFs, not the other way around. Therefore, our proposal foresees an augmentation to scenarios proposed by ETSI.

Furthermore, we decided that the MTD Controller would manage that movement through scaling operations. We based this decision on scaling operations being part of the lifecycle management functionalities assigned to the VNFM. Another essential characteristic of our architecture is its interoperability with the ETSI MEC. As we presented previously, launching, orchestrating, and managing a MEC ecosystem within an NFV ecosystem is possible. In that environment, MEC Applications can be viewed as VNFs or CNFs. Therefore, this MTD methodology can be used to protect those MEC Applications, where our MTD Controller will defend them.

3.3. MTD controller logic and functioning

As discussed, the MTD methodology developed in this work focuses on moving an application's versions. In other words, in some mutation window, the application's execution corresponds to some of its versions. However, in the upcoming mutation window, there is another version running. This behavior reflects the proposed MTD system's proactivity: it continually triggers the system's movement, corresponding to the executing version mutation. Therefore, when a mutation window ends, the proposed MTD Controller will begin the scale-out of another target application's version and the scale-in of the previous running version.

However, this is a complex MTD system since we needed to integrate it within the NFV framework. As demonstrated in this section, this MTD system will correspond to an MTD Controller VNFM. Furthermore, we also defined that the MTD Controller would manage the version being executed at each mutation window through scaling operations. Therefore, we needed to define a VNFM capable of doing such functionalities while following the delineated MTD design considerations.

Something noteworthy looking back is that the NFV-MANO proposed architecture possesses some reference points not yet standardized by ETSI. These are related to the interactions between the CISM, the CIS, the NFVO, and the VNFM. Since these are important to consider at this point, we needed to make some assumptions: the reference point between the NFVO and the CISM was considered similar, in functionalities, to the *Or-Vi*, which originally connects the NFVO to the VIM, and; the reference point between the VNFM and the CISM was considered similar to the *Vi-Vnfm*, which originally connects the VNFM to the VIM.

On the current reference for the NFV-MANO [22], in *Annex B.4.4*, ETSI presented the execution flow for automatic expansion and contraction of a VNF, including its scale-out and -in operations, respectively. Although they are given as being triggered by the VNF performance measurements, since the VNFM is the actual entity to start the scaling, they can be generalized for the present scenario. Therefore, the MTD Controller VNFM will trigger the analogous CNF scaling without analyzing its performance measurements but instead will have an internal "intelligence" related to its MTD functionalities. Furthermore, due to the assumptions made above for the reference points, we also adapted these flows for the CNFs scaling from existing VNFs.

Moreover, in the *Or-Vnfm* specification document [40], ETSI presented two ways of starting a VNF scaling: it can be initiated by the NFVO, commanding the VNFM, or; by the VNFM itself, corresponding to the automatic scaling. Likewise, it is also specified two distinct ways the VNFM can execute the necessary resource management operations: directly towards the VIM or; towards the NFVO, which in turn, will redirect them to the corresponding VIM. These operation management modes are called "VNF related resource management in direct mode" and "VNF related resource management in indirect mode", respectively. Since ETSI does not consider the usage of multiple VNFMs to manage the same VNF, we generalized the indirect mode so that the MTD Controller would first request the scaling operation towards the NFVO. Then, instead of allocating or releasing the resources towards the CISM, the NFVO will trigger the execution of the provider's CNF VNFM. Therefore, this proprietary VNFM will be the one that conducts the scaling operations towards the CISM and configures, when necessary, the new CNF instances.

Considering these assumptions, we summarized all the stated interactions in the sequence diagram of Fig. 5, where we took into account the scenario of the MTD Controller changing a CNF running version from one to two. As discussed above, this diagram was adapted for our strategy from the scaling flows discussed by ETSI for the auto-scaling of VNFs. One crucial detail that can be perceived in the schema is the scaling operations' order. The scaling-out of the new version, presented by the resources' allocation triggered by the MTD Controller towards the NFVO, takes place before the resources' release corresponding to the scaling-in of the previous running version. This order is of the utmost importance. If it were to be conducted in reverse, there would be a small window where the application would not have any running version, which would provoke its downtime. However, this process will lead to a short period where double the expected number of versions run simultaneously.

In Fig. 5, we also considered using multiple VNFMs to control the corresponding CNF lifecycle. The MTD Controller only requests grants for the scaling operations and redirects the related resource's allocation or release to the NFVO in indirect mode. Finally, our considered extensions to this mode can also be perceived in the diagram: the NFVO triggers the scaling of the CNF towards the provider's VNFM, which will allocate/release all the necessary resources in direct mode and configure the CNF according to the specifications given by the application provider, using the CNF descriptor.

Another important detail we must consider from these interactions is that there will be a delay between the MTD Controller's decision to launch a new version and this version being instantiated. This delay is because there are multiple entities in play, with interactions among them that need to occur for a new version to be launched. Therefore, a specific mutation window period might be executed for more or less time than initially expected, i.e., from these interactions, we predict that the mutation windows we will get in practice will have a different dimension than the provider initially specified. Consequently, there will be an error between the experienced mutation window periods and the theoretical one, which the application provider must consider when deciding the best mutation window period.

Apart from the relations discussed above, it is also essential to highlight the MTD Controller's internal functioning, whose decisions cause the described sequence to happen and, therefore, the targeted CNF to be secured. In Fig. 6, we present the Controller execution flow. First, two variables are displayed in this schema: the *number_simultaneous_versions* and *time_window*. The application provider can control these two variables' values when it instantiates its CNF. The first controls the number of versions the provider wants to be executed simultaneously, hence controlling the redundancy level. The second relates to the mutation window size, i.e., the total time each version, or conjunction of versions, will be executed until the system moves.

Considering this flow, no CNF version is running when the MTD Controller initializes, albeit the CNF is already instantiated. This behavior is because the MTD Controller is the entity that decides when and what versions to deploy. When the Controller obtains the current non-running versions, there will be none, meaning that in the next execution step, it will select *number_simultaneous_versions* of all the existing application's versions. At this moment, the process will split in two, where the first will be responsible for the MANO interactions, as depicted in the diagram of Fig. 5. In contrast, the second will hold for *time_window* seconds, corresponding to the mutation window size. When this delay ends, the Controller will once again obtain all the non-running version(s) and choose *number_simultaneous_versions* randomly to replace the one(s) executed in the previous mutation window. Another property of the proposed Controller is that it has no finishing point, being in an endless loop until the CNF is destroyed, which marks all its VNFMs removal.

3.4. Network considerations

Despite all these internal and external functioning details, there was still an issue to consider. One of this MTD system requirements was that it should not affect the normal functioning of the protected application, which meant that the MTD mechanism should be transparent for any end client interacting with the protected CNF. In practice, this raises two problems to be solved: the application cannot have any downtime while moving, and; any Transmission Control Protocol (TCP) open session should not break unexpectedly. We did not consider User Datagram Protocol (UDP) since it does not establish sessions, so it will not present any of the described issues. Starting with the first problem, we have already provided a solution previously. To avoid downtime while the

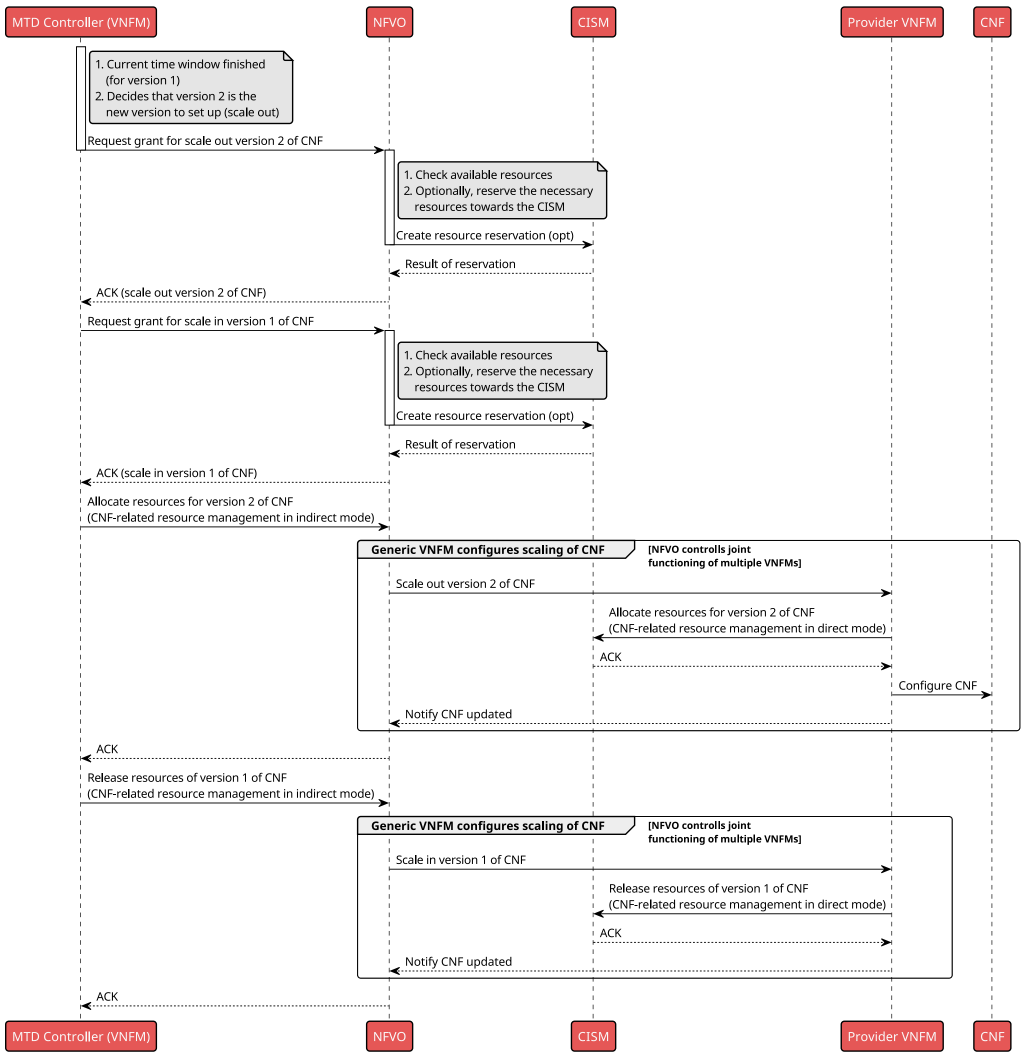

MTD Controller (VNFM)
NFVO
CISM
Provider VNFM
CNF
1. Current time window finished (for version 1)
2. Decides that version 2 is the new version to set up (scale out)
Request grant for scale out version 2 of CNF
1. Check available resources
2. Optionally, reserve the necessary resources towards the CISM
Create resource reservation (opt)
Result of reservation
ACK (scale out version 2 of CNF)
Request grant for scale in version 1 of CNF
1. Check available resources
2. Optionally, reserve the necessary resources towards the CISM
Create resource reservation (opt)
Result of reservation
ACK (scale in version 1 of CNF)
Allocate resources for version 2 of CNF (CNF-related resource management in indirect mode)
Generic VNFM configures scaling of CNF
[NFVO controlls joint functioning of multiple VNFMs]
Scale out version 2 of CNF
Allocate resources for version 2 of CNF (CNF-related resource management in direct mode)
ACK
Configure CNF
Notify CNF updated
ACK
Release resources of version 1 of CNF (CNF-related resource management in indirect mode)
Generic VNFM configures scaling of CNF
[NFVO controlls joint functioning of multiple VNFMs]
Scale in version 1 of CNF
Release resources of version 1 of CNF (CNF-related resource management in direct mode)
ACK
Notify CNF updated
ACK

Fig. 5. Sequence diagram of the interactions between the proposed MTD Controller and the MANO entities.

system moves, the MTD Controller shall first scale out the new version, and only after will it scale in the previous one. However, this solution is not an answer to the second problem.

The second problem's cause is that if a connection is made with some running version, that connection will be lost when a new version replaces the older one. Our hypothesized solution was the usage of a reverse proxy provided by the Operator to each of its clients. This proxy could receive any connection from an end client and redirect them to the analogous running version. Suppose, at some moment, the MTD system modified the executing version. In that case, the reverse proxy can maintain the same TCP session with each end client and create the necessary connections to the new version in the backend.

Another issue to consider is managing the network traffic when more than 1 version is being executed at some point. This behavior appears in two different scenarios. The first is when a mutation window finishes and a new one occurs: our design imposes that the MTD system scales out the latest version before scaling in the previous one. This aspect will lead to a short period where double the predicted versions are being run, meaning that even for redundancy of one, two versions will be executed in that short period. The other scenario where the described behavior is observed is for redundancy of more than one since there is always more than one version running. Under those circumstances, we propose to use a load balancer. This software will sit between the end client and the running versions and balance coming requests through them. Furthermore, we suggest that this load balancer selects the version randomly, even if the end client making the request is the same on two distinct occasions. Our argument for this choice is that this random behavior will introduce a new level of confusion into the overall MTD system.

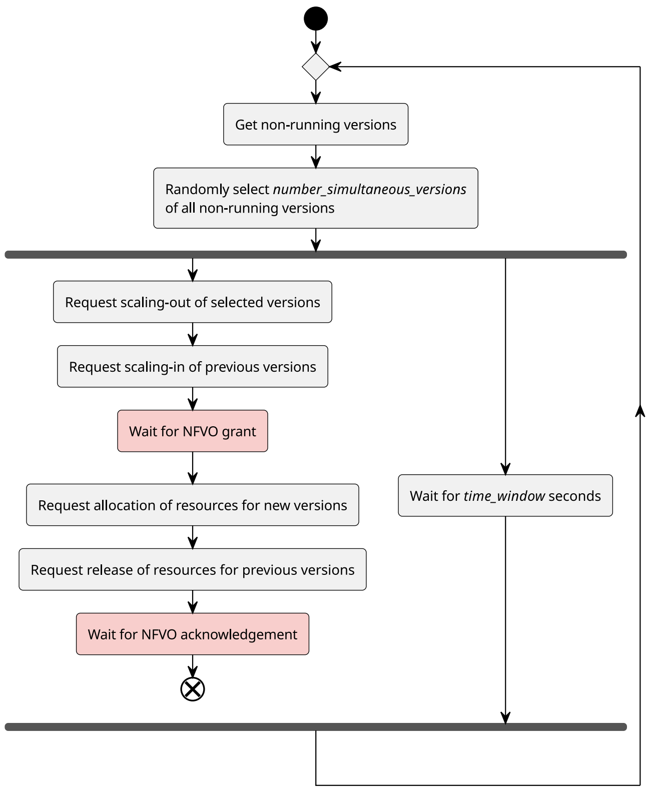

Get non-running versions
Randomly select *number_simultaneous_versions* of all non-running versions
Request scaling-out of selected versions
Request scaling-in of previous versions
Wait for NFVO grant
Request allocation of resources for new versions
Request release of resources for previous versions
Wait for NFVO acknowledgement
Wait for *time_window* seconds

Fig. 6. Proposed MTD Controller execution flow.

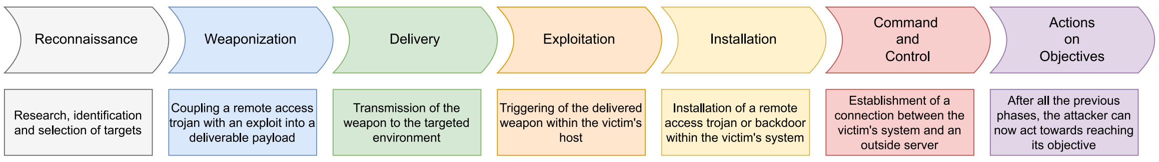

Reconnaissance
Weaponization
Delivery
Exploitation
Installation
Command and Control
Actions on Objectives
Research, identification and selection of targets
Coupling a remote access trojan with an exploit into a deliverable payload
Transmission of the weapon to the targeted environment
Triggering of the delivered weapon within the victim's host
Installation of a remote access trojan or backdoor within the victim's system
Establishment of a connection between the victim's system and an outside server
After all the previous phases, the attacker can now act towards reaching its objective

Fig. 7. Intrusion Kill Chain phases.
Source: Adapted from the information in the Lockheed Martin Corporation's white paper [41].

4. Security evaluation

Having finished the MTD proposal, we created a Proof of Concept (PoC) implementation, whose details can be followed in the appendix of this document, in Appendix A. With this PoC, we could empirically assess the security our methodology provides. Therefore, we conducted a security study, examining the MTD system against a step-by-step attack on an exemplary vulnerable CNF. With this in mind, we start this section by reviewing the specifications of the VMs we used and the evaluation methodology, ending it by providing the results.

Furthermore, we followed the Intrusion Kill Chain process to study how a real aggressor would attack this application. This methodology, developed by Lockheed Martin Corporation and presented publicly in a white paper [41], is based on the theory that a malicious actor needs to follow a set of well-defined, sequential phases to breach a system successfully. We have summarized these seven phases in Fig. 7.

4.1. Hosts specifications

For this evaluation, we used four distinct VMs: one that executed OSM, two for the Kubernetes cluster, composed of a Kubernetes Control Plane and a Worker Node, and another used to perform the test scripts. The first machine had 12 Virtual CPUs (VCPUs) and 16 GiBs of RAM. Then, the two Kubernetes nodes had similar characteristics to one another: four VCPUs and eight GBs of RAM. As for the VM used to run the tests, it had two VCPUs and eight GBs of RAM.

4.2. Methodology

This subsection explains how we evaluated the security of the proposed MTD system. We start by outlining our test scenario, where we created a CNF with five versions to be protected by the built MTD PoC. Then, we describe the analysis methodology itself, with the environment setup and how we conducted the experiment.

4.2.1. Scenario

To evaluate the security added by the proposed MTD system, we developed a CNF, which packaged a web application with multiple versions, one of which had a well-known vulnerability. In this work, the vulnerable version had a Zip Slip vulnerability,[2] a well-known and usually critical vulnerability that commonly allows potential attackers to achieve remote code execution. It is characterized by allowing a path traversal when extracting files from an archive, such as a zip file. Therefore, an application with this vulnerability enables, in theory, an attacker to send a specially crafted archive containing an exploit, which will be saved in a directory outside the one where the uncompressed contents typically would be stored.

Our sample application was an image repository where any end client could store images. Besides allowing storing photos, it also admits sending a zip file, enabling a client to send multiple images simultaneously easily. All the created versions used the same HTML template to create the frontend page for the users requesting access. We followed this pattern to elude any user accessing the page at different moments into believing that the application was always the same. Therefore, the difference between each version was in their backend.

We have created five versions of this application since we conducted tests for redundancy of one and two. Since the minimum number of running versions of an application concerning the number of simultaneous versions, s, without compromising the randomness introduced by the MTD methodology is $2 * s + 1$, and the maximum value of s used on the tests below, as referred, is two, the minimum number of required versions, given by this formula, is five, so this was the chosen number. Also, to showcase a provider's multiple possibilities of creating multiple versions, we used three distinct programming languages to create them. Considering this, we developed two versions in Python, two in Java, and one in PHP. The distinction between both Python versions was the web framework used, where we used Flask[3] for one and CherryPy for the other. As for the Java versions, the difference was based on the version of the library used to manage the uploaded zip files. On one, we used version 1.12 of ZeroTurnaround ZIP,[4] and on the other, version 1.15. The first, version 1.12, is vulnerable to the Zip Slip vulnerability. Considering these implementation details, we named our versions as follows: *java*, *java-zip-slip*, *flask*, *cherrypy*, and *php*. Besides these versions sharing the same HTML template, they also share a volume. This volume allows the images each user uploads to persist between system movements. Therefore, if a user uploaded a photo while version one was running, the page will present that image when the user accesses it later when another version is running.

4.2.2. Analysis setup

To conduct this security analysis, we attached our created PoC, detailed in Appendix A, to this CNF. Then, to launch the CNF, we used OSM as the MANO and Kubernetes Control Plane as the CISM. Moreover, we created a testing script and launched it in a different host than the one used to run OSM or the Kubernetes nodes. We designed it to administer OSM to instantiate the Network Service (NS), which packaged our sample CNF. The analogous instantiation command was then sent in a loop, in which each cycle corresponded to each mutation window size studied. For each mutation window size, the script waited for a specific period and then instructed OSM to destroy the corresponding NS instance. This period was dependent on the test we were doing. We used a one-hour interval for the Reconnaissance phase, whereas, for the Exploitation, Installation, and Command and Control (C2), we used a 20-min period. Then, depending on the phase, the script started the experiment again for the same mutation window size. For the Reconnaissance, we only instructed the script to run one experiment of one hour for each mutation window since our objective with this analysis was to verify the variability of responses that fingerprint tools gave. As for the others, we did ten runs of 20 min for each mutation window to obtain results with the lowest possible error. Finally, we also executed this script for one and two simultaneous versions.

Furthermore, in each experiment, this script launched multiple bots whose functionality was based on the Intrusion Kill Chain phase to which they were targeted. For the Reconnaissance phase, we used two distinct bots: one that executed Nmap against the target server and another that we used to make HTTP connections to the port where the application was listening on that server. For the other phases, we created an "attack bot" that delivered the weapon created within the zip file, as discussed in Appendix B.2, to the target application and then tried to access the corresponding web shell. Fig. 8 presents the basic functioning of this bot. It mainly has two mods. The first is to detect when the Exploitation and Installation phases were successful by continually sending the crafted exploit, followed by a shell command, and observe when the server returns the correct response for that command. When this behavior happens, the bot perceives that these Intrusion Kill Chain phases were completed and moves to the next mod. This mod is related to detecting the period the attacker can access the C2 channel. Therefore, the bot continues sending shell commands until it receives the wrong response, i.e., not the expected result for the shell command, but instead, the original webpage. This behavior happens when the MTD causes the system to move and, therefore, the vulnerable version to be destroyed and, with it, the installed exploit. Thus, the bot signals the C2 phase as finished and tries sending the exploit repeatedly until a new success.

[2] Zip Slip Vulnerability introduction by Snyc: https://security.snyk.io/research/zip-slip-vulnerability.

[3] Flask project page: https://flask.palletsprojects.com/.

[4] ZeroTurnaround ZIP project repository: https://github.com/zeroturnaround/zt-zip.

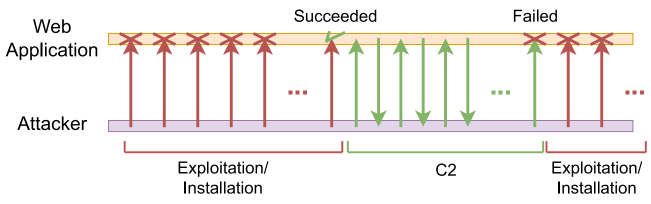
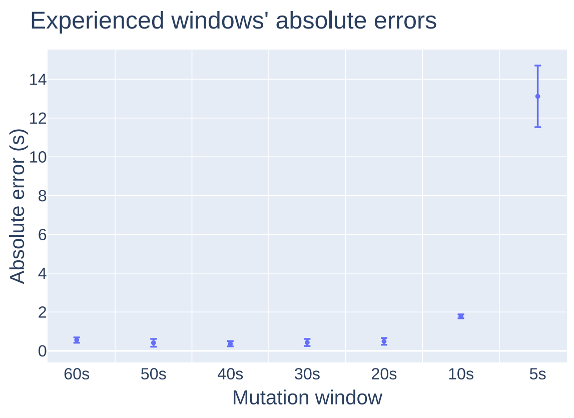
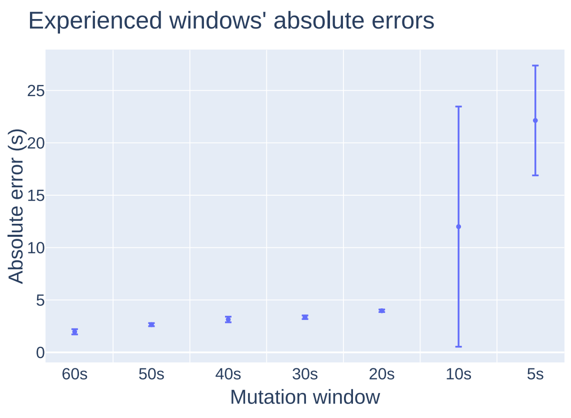


Fig. 8. Attack bot interactions respecting the Exploitation and C2 phases with the web application.



(**a**) For one simultaneous version.



(**b**) For two simultaneous versions.

Fig. 9. Absolute error obtained from the observed windows.

4.3. *Results and discussion*

In this subsection, we exhibit the MTD evaluation results over the multiple steps of the Intrusion Kill Chain, except for the Weaponization and Delivery, as our MTD mechanism, by design, does not affect them. Furthermore, we also provided how an attacker would probably intrude on the vulnerable web application without any MTD defense in this document's appendix, in Appendix B.

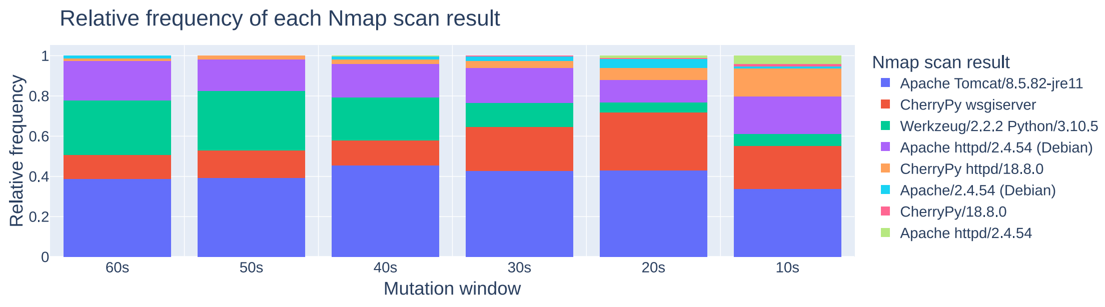
4.3.1. *Observed mutation windows' absolute error*

As we referred to in Section 3.3, since multiple entities interact to achieve the MTD methodology's protection, there will be a delay between the moment the MTD Controller makes a decision, and that decision has a practical effect. Therefore, we calculated this delay and considered it when conducting the tests presented in this section. More specifically, we calculated the mutation window sizes we obtained in practice for each run of 20 min and, from them, the absolute error, i.e., how much they deviated from the theoretical mutation window of that experiment. In the end, we calculated the average of those absolute errors for the ten experiments of each mutation window size tested, with the respecting standard deviation. We represented the results obtained in Fig. 9 plots, each for each number of simultaneous versions tested.

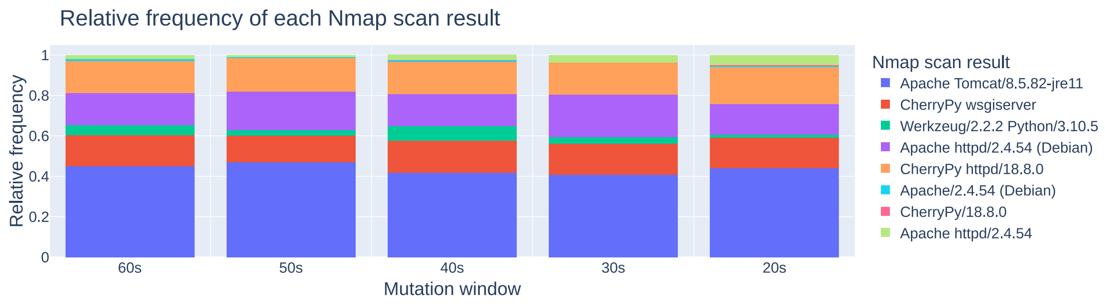
From the graph of Fig. 9(a), we can conclude that, for one simultaneous version, the absolute error maintained itself more or less stable from the 60 to the 20 s mutation windows, only being greater than 0.5 s for the 60 s mutation window, where it was of 0.55 ± 0.14 s. However, this error was significant for the 10 and 5 s mutation windows, where we obtained values of 1.77 ± 0.10 and 13.12 ± 1.59 s, respectively. Considering these results, we can conclude that, for the created examination setup and one simultaneous version, the system cannot handle well mutation window sizes below 10 s where, for 5 s, the error was much larger than the mutation window size. Therefore, in the results concerning one simultaneous version we will present below, we only considered mutation window sizes between 60 and 10 s, inclusively.

As for the obtained absolute error for two simultaneous versions, shown in the graph of Fig. 9(b), the results show a slight increase from 1.67 ± 0.25 to 3.97 ± 0.12 s with the decrease in mutation window size from 60 s to the 20 s mutation windows, respectively. However, for the 10 and 5 mutation window sizes, the average absolute error was much greater than the mutation window size. Consequently, we will only consider the mutation window sizes between 60 and 20 s in the results we present for two simultaneous versions.

This discrepancy between absolute errors obtained between greater and smaller mutation window sizes can be mainly attributed to using the OSM framework as our chosen MANO. We observed that, as we decreased the mutation window size, OSM became more overwhelmed with the incoming tasks from our MTD Controller. After some time, these tasks started accumulating in its internal queue, resulting in experiencing mutation windows longer than the theoretical ones. Furthermore, one of the main reasons for OSM



(a) For one simultaneous version.



(b) For two simultaneous versions.

Fig. 10. Relative frequency of each Nmap scan result, obtained throughout an experiment of one hour.

not being able to take care of said tasks rapidly is due to OSM not handling Kubernetes-related activities asynchronously. Therefore, even if it had, for instance, three scale-out requests in its queue, it could only take one at a time. This last reason is why the absolute error is greater for two than for one simultaneous version.

4.3.2. Reconnaissance

The first step in this phase was to use Nmap to detect what services were available on the server, similar to what we did for the application without protection. Since the system under analysis was the MTD system, we did a study over time to verify the Nmap tool's results, considering the modification of the version being executed over time. As we were modifying the application's version, the expectation was that Nmap would retrieve a scan result dependent on the mutation window being used, i.e., considering the version instantiated at the moment, Nmap should always retrieve a result related to that version. However, after running Nmap for each standalone version, some results were excitingly different from the expectation. More specifically, it gave the same response for both versions built using Java. This outcome happened because their only difference stemmed from a distinct ZeroTurnaround ZIP library version.

Since Nmap could detect all the other versions, we anticipated that, when running it iteratively during the extension of each trial with the MTD system, its scan results would be similar to the standalone ones, depending on the version executed at the moment it was run. However, the actual results were surprisingly different, as summarized in the histograms of Fig. 10. These histograms were obtained from the frequency of each obtained Nmap result throughout the one-hour experiment for each mutation window size. Each plotted graph corresponds to each tested redundancy level, Fig. 10(a) being relative to one simultaneous and Fig. 10(b) to two concurrent versions. Since each Nmap scan was made right after the previous one finished, the interval between each scan was considered negligible. In these histograms, there was no mutation window where the number of Nmap distinct scan results was four throughout each observation extension. Furthermore, there were some mutation windows, the 20 and 10 s for one simultaneous version and the 40 and 20 s for two simultaneous versions, with eight different Nmap scan results.

Moreover, we would expect each version's analogous results to be obtained one-fifth of the time, except for the Java-related results, where they would be obtained two-fifths of the time. This expectation of getting each result one-fifth of the time can be explained by the probability of hitting a specific version at any given moment, demonstrated in Appendix C.1. Since each Nmap scan can be interpreted as a unique attempt, the chance of obtaining each version will equal the probability of hitting that version for only one attempt. Furthermore, the assumption of obtaining each result one-fifth of the time, except for the Java-related result, is supported not only in theory but also in practice, as demonstrated by the results showcased in the histograms of Fig. 25 of this document's appendix in Appendix D. These histograms represent the relative frequency of each installed version in the same experiment, each one for each number of simultaneous versions tested. Although the Apache Tomcat, corresponding to the Java-based versions, is broadly obtained two-fifths of the time, our expectations for the other versions were not met, primarily for the smaller mutation window sizes. By comparing the results we got with Nmap, demonstrated in Fig. 10, with the expected results, demonstrated in Fig. 25, we can notice this apparent deviation. Furthermore, when comparing the results obtained for one and two simultaneous versions, we witness a considerable difference between the frequency of some results obtained with Nmap. Some results, such as the CherryPy httpd, are obtained more frequently with two simultaneous versions. In contrast, others, such as the Werkzeug Python or the CherryPy wsgiserver, are obtained many more times.

In the end, the explanation for these unexpected results was found to be related to Nmap's internal behavior. Its documentation[5] states that after Nmap succeeds in connecting to a TCP port, it will listen, without making any requests, for about 5 s. Then, if it does not receive anything, as with a web server, it will send multiple probes related to the service most commonly served on that port. After that, it compares the obtained response for each probe with a list of signatures, each dependent on the match level obtained from the previous ones. Although the documentation does not specify the most common probes sent to each port, we captured the packets during our tests to ascertain what Nmap was doing. The results obtained showed that, for each scan, Nmap sends multiple GET requests to the 8000 port. For one simultaneous version, when Nmap was executed in the middle of a mutation window, the obtained response almost always had the HTTP header *Server* value equal to the service providing that web application. However, in intervals where the mutation windows were being exchanged, the header's value went back and forth between scan responses. This result happened because of our load balancer, which connects the client to one random running version when there are multiple. Moreover, we observed that the HTTP header *Server* value almost always changed within the same Nmap scan for two simultaneous versions. With this in mind, when Nmap tries to define the most probable service being executed in the scanned port in these conditions, its algorithm, most probably, tries to make a deliberation considering the "weight" of each match.

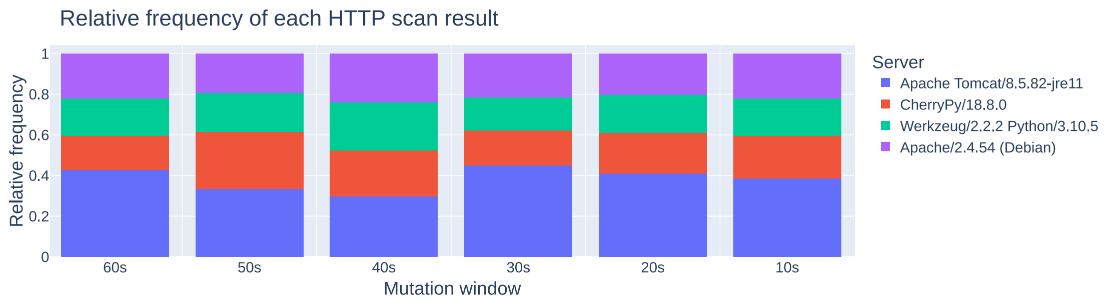
Considering that usually, the first step an attacker takes when confronted with a server is to scan all open ports and respective services, generally using tools like Nmap, we demonstrated that the created MTD mechanism could mislead the attacker in this phase. Not only does the attacker obtain information about the system that goes out of date when it moves, but it might also be erroneous. However, an intruder might try to use other simpler fingerprint mechanisms. Since the application under study is web-based, it usually sends responses to clients containing the HTTP *Server* header. If not obfuscated, this header usually provides information regarding the software running on the server side. Therefore, when the attacker discovers the opened port and that it is an HTTP service, it can create a simple script to obtain the *Server* header value with only one request and, with it, fingerprint the web application. This methodology is more straightforward and faster and only requires one request and one response, being less evasive than methods similar to Nmap. With this in mind, we also tested the usage of this type of mechanism. To achieve that, we created a simple script to send a request to port 8000 and, from the response, store the *Server* header value. Then, similarly to what we did with the Nmap tests, this script was executed in a loop for one hour for each mutation window size under study for one and two simultaneous versions. From the obtained results, we created the histograms of Fig. 11. Fig. 11(a) represents the results for one concurrent version, and Fig. 11(b) for two. As before, the histograms represent the frequency of each scan result, and we considered the period between each scan negligible.

The results obtained from this second fingerprint methodology had more quality than those from Nmap. The most noticeable difference is that in this case, unsurprisingly, there are no deviant results: as expected, the number of different Server software "versions" obtained was four. Also, in these histograms, it is more apparent that we got the Apache Tomcat server version more or less double the time each of the other server versions. This affirmation holds especially true for two simultaneous versions where the error between the theoretical and obtained values is diluted since more versions are installed in the same period. Furthermore, let us compare each HTTP scan result with the frequency at which each version was installed, whose results we present in Fig. 26 in the appendix of this document in Appendix D. We can conclude that this method is much more accurate, as each installed version's frequency almost perfectly matches the corresponding scan result frequency.

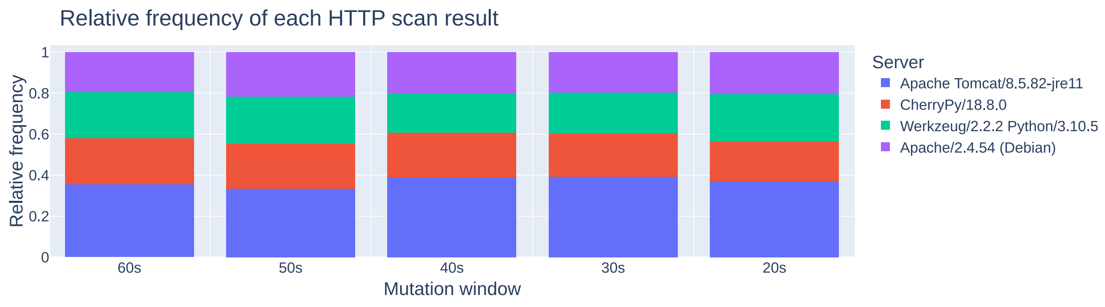
Considering these results, we can say that a threat actor will have a more difficult time in this phase of the Intrusion Kill Chain with the presence of this MTD system. An attacker might start by scanning all open ports and, from that analysis, will have a false sense of what is being executed in the target system. That is because, with only that scan, the attacker will conclude that those ports are running some web application version and software all the time. Furthermore, as proven by the above Nmap results, this first scan can give an erroneous guess as to what version was running when it was executed. This lack of information that the application is not running just the version given by the first scan, or the obtained misleading results, might lead the intruder to continue the attack by skipping to the following Intrusion Kill Chain phases. In this example, that malicious actor could even "be lucky" to the point where it obtained from the first scan that the application running on the server side was based on the Tomcat web server and, with that, found the GitHub repository with the vulnerable code. However, when trying to exploit the application, the attacker would have some adversity in succeeding with the attack, for instance, if it was trying to deliver the crafted weapon when a non-vulnerable version was being executed. At this moment, the intruder might return to the first phase and analyze what is happening with this web application in more detail. However, this episode marks the success of using an MTD methodology: confuse and delay the attacker.

We can also suppose that the intruder returns to this phase for extended analysis and finds out, using Nmap or a more straightforward tool such as the presented script, that there are multiple outputs for the application scan. In this case, the attacker might conclude that the application provider is using some method to trick potential intruders by, from time to time, changing the *Server* header value or changing the response to probes commonly sent by tools such as Nmap. In such a case, the attacker might even guess that the repository it found was an older application version that now does not have the vulnerability and gives up. However, it might also discover that there is a mechanism that is changing the running application's version. In this case, the attacker can try fingerprinting not only the versions used by the application but also the mutation window size, probably using a

[5] Service and application version detection technique Nmap documentation: https://nmap.org/book/vscan-technique.html.



(a) For one simultaneous version.



(b) For two simultaneous versions.

Fig. 11. Relative frequency of each HTTP scan result obtained throughout an experiment of one hour.

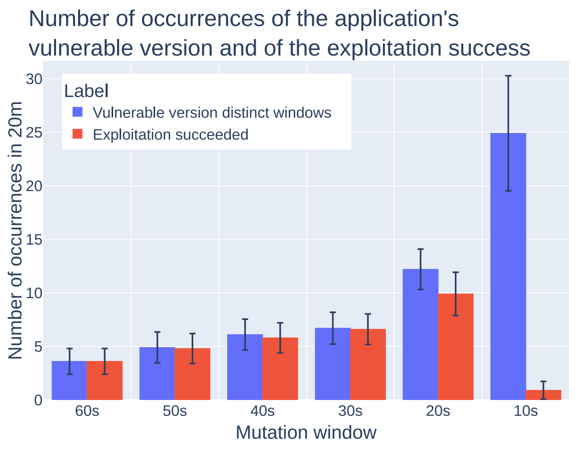
less overwhelming mechanism, such as the HTTP scan demonstrated in this section. However, this method has a problem: if the attacker decides to use tools such as Nmap, or even if it creates one to make the same request for an extended period continuously, there might be an Intrusion Detection System (IDS) that flags that client's behavior as suspicious. In such a case, the IDS might block that attacker from doing any new interaction, ceasing the attack. It is necessary to, once again, point out that MTD, as any defense mechanism, is not meant to be a solution for all security problems. This joint usage of an MTD mechanism and an IDS shall be something to study in the future.

4.3.3. Exploitation, installation, command and control, and actions on objectives

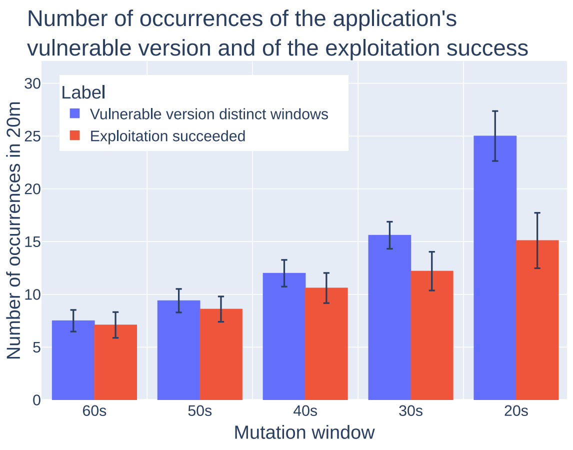
The following phases of the Intrusion Kill Chain are more or less interconnected. This affirmation holds, considering that the weapon we produced for the presented vulnerability is already a backdoor. Therefore, the Installation ends when the Exploitation finishes, as the attacker already has a backdoor installed on the target system. Accordingly, for this test, we considered that these phases co-occur. With this in mind, when we refer to the Exploitation phase below, we also refer to the Installation. An important note to consider is that here, we discarded the possibility of the application's provider or the Operator noticing the existence of this backdoor.

To assess the added security given by the proposed MTD system, we conducted an experiment with the two following purposes. First, verify the success of the Exploitation phase for each mutation window size. Second, for each successful Exploitation, analyze the time an attacker has to conduct the C2 and Actions on Objectives phases. To achieve this, we created an "attack bot" that tried to continually send the weapon to the web application, regardless of the version being executed at each moment. Then, after each weapon delivery, it tried, for a brief period, to perform a given command by attempting to access the established backdoor. If it successfully accessed the web shell during this period, it marked that trial as an Exploitation phase completed. In that case, it proceeded for the experiment's second purpose: send the same command to analyze the period where the C2 phase could occur successfully. Furthermore, when each mutation window finishes, the related version is destroyed, which leads to dismantling any established exploit or backdoor, terminating the settled C2 channel. Therefore, each time the communication channel ended, the bot would try to send the weapon again continuously, and the described process started again.

From the success level results obtained for the Exploitation phase, we created the graphs of Fig. 12 for one and two simultaneous versions. These plots represent the total number of vulnerable version deployments for each mutation window size experiment and the corresponding number of exploitation phases that succeeded throughout the ten runs of 20 min. Therefore, the total number of exploitations in each mutation window size trial has a superior limit equal to the number of vulnerable version deployments. Generally speaking, the exploitation phase success rate grows with the increase in the mutation window size, i.e., when trying to conduct the Exploitation, an attacker will succeed more when the mutation window size is bigger. Also, as the mutation window size lowers, versions are instantiated for a smaller period. This characteristic means that within an equal interval, the attacker has to begin the task of sending the exploit again more times for smaller mutation windows. Therefore, although, as shown in Fig. 12,



(a) For one simultaneous version.



(b) For two simultaneous versions.

Fig. 12. Occurrences of the application's vulnerable version and the exploitation success.

the gross number of exploitation successes increases with the decrease in the mutation window size, that only happens because the attacker has to start the process again, as the vulnerable version is removed more often.

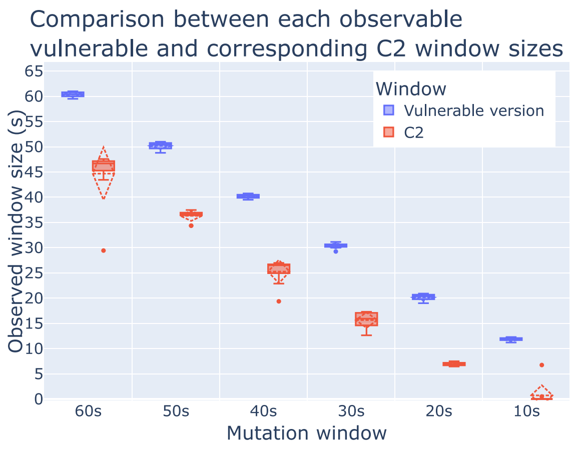
Regarding the differences between the redundancy levels used, the results of Fig. 12(a) show that the success rate of the Exploitation was roughly stable and close to 100% between the 60 and 30-s windows, inclusively. Then, for the 20-s mutation window, it was considerably less than the other longer windows, reaching 9.9 ± 2.02 successes out of 12.20 ± 1.89 vulnerable windows. Finally, the 10-s mutation window dropped to 0.90 ± 0.83 successes out of 24.90 ± 5.37 vulnerable windows. Moreover, in the 10-s mutation windows test, there were experiments where the bot had no success finishing the Exploitation over 20 min. Consequently, we can say with some assurance that, for the application we created for this test and for the specific vulnerability it had, a window size roughly below 10 s will make the Exploitation almost impossible to achieve. Naturally, when we approach a mutation window size close to 0 in a system such as this one, the MTD methodology will provide the best safety. However, this comes at the cost of performance loss (we further discuss this problem in Section 5). Therefore, we usually want a good balance between protection and performance. From the results we obtained and under this analysis' conditions, a window size close to 10 s will be the size that gives almost optimal protection for the least negative performance impact for one simultaneous version.

As for using two simultaneous versions, whose results are shown in Fig. 12(b), the Exploitation success rate was less for each mutation window size compared to one concurrent version. This difference was as pronounced as the window size decreased. For the 20-s window, we obtained close to 60% success in exploiting the application, with 15.10 ± 3.58 successes out of 25.00 ± 2.37 vulnerable windows. Furthermore, for each mutation window size, the attacker had to send the exploit more times than when using redundancy of one because, in this case, the vulnerable version is removed more times. For these reasons, we can conclude that the redundancy of two instead of one confers more protection to the CNF for mutation windows between 60 and 20 s.

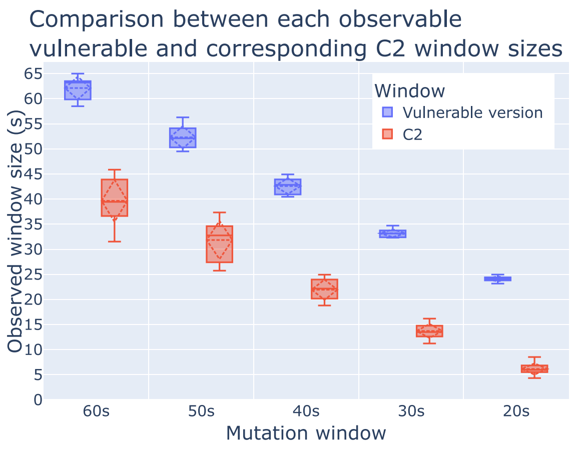
This experiment is related to the probability we calculated in Appendix C.2 in the appendix of this document. This relation is because we are observing the results obtained only when the vulnerable version was installed. However, comparing the results of Fig. 12 with the probability values highlighted in Fig. 24, we can notice some distinctions. First, the empirical results show that the probability of achieving Exploitation with success lowers with the increase of the mutation window. However, in Appendix C.2, we proved that the likelihood of reaching the Exploitation did not depend on the mutation window size but only on the number of concurrent versions. The explanation for this difference between theoretical and practical results is twofold. On the one hand, the empirical results depend highly on the attacker, or in this case, the attack bot, strategies, which our theoretical models did not consider. On the other hand, and more importantly for redundancy of more than one, the smaller the mutation window, the smaller the number of possible attempts an attacker can make. Therefore, although an attacker with an infinite number of trials could reach the Exploitation phase, as proven by the formulas showcased in Appendix C.2, when the number of attempts is finite and decreases with the mutation window size, the Exploitation success also decreases with the mutation window size. This explanation is why we had almost 100% success for a 60-s mutation window, but it was close to 60% for the 20-s one for the two simultaneous versions' results. Furthermore, it is necessary to denote that our bot, failing at achieving the Exploitation, would take more than two seconds to retry.

Another considerable difference between these results and the theoretical assumptions was that we obtained almost 0% success for a mutation window size, the 10-s one for one simultaneous version. However, in our theoretical models, we did not consider the impact of the time the vulnerable version would take to execute the exploit. As demonstrated further within the C2 results, the used exploit seems to take about 13 s to be executed. Therefore, mutation window sizes closer to this value will lead to an Exploitation success near 0% for this specific scenario.

Regarding the C2 phase, although it usually can begin right after the Installation, a C2 channel is usually kept stealthily for the intruder to achieve its objectives in the future. As explained in Appendix B.6, without MTD protection, an attacker can maintain a C2



(**a**) For one simultaneous version.



(**b**) For two simultaneous versions.

Fig. 13. Comparison between each observable vulnerable and the corresponding C2 window sizes.

connection as long as the Operator or the provider does not realize the attack or if the targeted container never goes down. However, with the proposed methodology, in theory, the intruder can only maintain a C2 channel if the exploited version's mutation window is maintained. When that window finishes, that version will be replaced by another, destroying the old container and creating a new one. In practice, this behavior was proven true, as the box plots of Fig. 13, made from the results of the experimentations discussed above, demonstrate. These plots show that with the decrease of the used mutation window size, the C2 window size, i.e., the time the attacker can access the created C2 channel, also decreases. Furthermore, in these plots, we can also notice a big difference between the vulnerable window size, corresponding to the time the vulnerable version is instantiated throughout each experiment, and the C2 window size. This outcome was expected since the Exploitation phase takes some time to be completed.

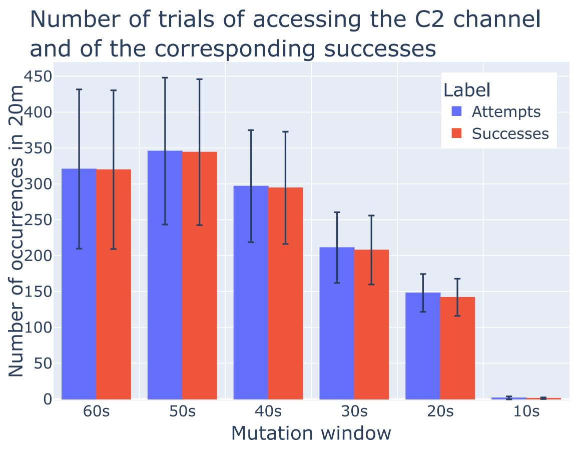
Moreover, from the graph of Fig. 13(a), we can notice the difference between the averages of the vulnerable version and C2 window sizes for one simultaneous version was always greater than 13 s for the mutation window sizes between 60 and 20 s. This outcome further proves the 10-s mutation window to be close to the limit where an attacker has more difficulty achieving any Exploitation successfully. Since there are usually more than 13 s between the window starting and the exploit being executed, for the most part, the attacker will not have the necessary time to accomplish these phases for windows close to 13 s. This reason is also why we obtained a C2 window near 0 s for the 10-s window size. Moreover, this period seems related to the time it takes to execute the exploit and install our backdoor, as nothing else affected our bot's ability to access the C2 window after attempting to send the exploit. The only time between sending the exploit and trying to access the backdoor was a 0.5-s wait.

We can draw a similar conclusion for the two simultaneous versions from the results in Fig. 13(b). From the tested window sizes, we consistently obtained a difference between the averages of the vulnerable version and C2 window sizes greater than 17 s. Therefore, if we want to achieve the best protection for the most negligible performance impact, the best window size for redundancy of two would be close to 17 s for this application in these settings. However, this conclusion is difficult to prove with the performance degradation caused by OSM. Nevertheless, this 17-s interval appears to contrast with the assumption that the exploit would take about 13 s to be executed. However, we also need to consider that when there are more concurrent versions, the attacker has less success communicating with the vulnerable version, as demonstrated in Appendix C.2. With this in mind, when our bot did not successfully communicate with the backdoor, it would wait for some time before retrying.

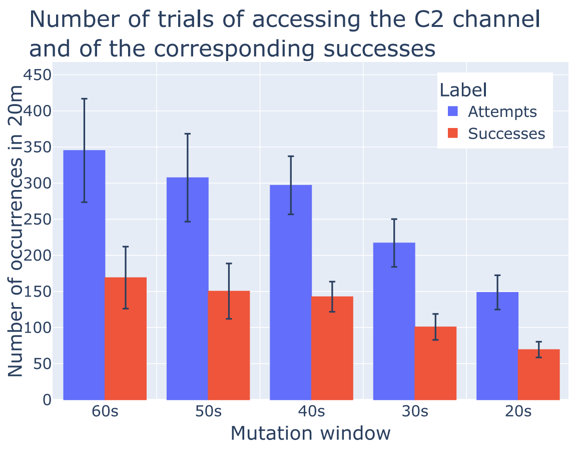
However, we have calculated these C2 windows from the first to the last time the bot had access to the web shell on the corresponding mutation window without considering what happens in the between. Inherently to our system design, there are moments when more than one version is running. In those cases, the version the client will communicate with is unclear, *a priori*, since Kubernetes balances those connections throughout all the running versions. Therefore, and especially when using more than one simultaneous version, we would anticipate the attacker would not be able to access an installed backdoor with 100% certainty, even during the C2 window. To verify this hypothesis, we created the graphs of Fig. 14. These charts have considered the number of attempts and their successes when accessing the backdoor for each mutation window size.

In this case, there is a noticeable difference between the one and two simultaneous versions' experiment results. For one concurrent version, the success rate of accessing the C2 channel is close to 100% for all mutation window sizes. However, the success rate is always less than 50% and slowly decreases with the mutation window size for two simultaneous versions. This difference stems from the existence of the Kubernetes load balancer. At two different moments, the version responding to a client can be different. Therefore, the presence of two simultaneous versions increases the provided protection since the attacker has less than half the success accessing an established backdoor.

Moreover, these results align well with the theoretical model highlighted in Appendix C.2 of this document's appendix. Since the graphs of Fig. 14 showcase the successes of singular trials, the number of attempts they represent is one. From the plot of Fig. 24 for the likelihood of hitting a specific version if that version is installed, we can conclude that the probability of communicating with



(a) For one simultaneous version.



(b) For two simultaneous versions.

Fig. 14. Number of successes and failures when accessing the C2 channel.

the backdoor is 100% for one concurrent version and 50% for two when the attacker only attempts accessing once. These values are roughly what we obtained for all mutation window sizes in practice.

The results obtained in this section have shown that the defense of our MTD system increases the difficulty a malevolent actor has accessing a backdoor installed within an attacked system. Because of this, the attacker will have a more challenging time achieving its goals. Furthermore, our MTD system removes any possible persistence of any installed backdoor. Consequently, when the backdoor is removed before an attacker uses it, it cannot achieve its objectives and, in the best scenario, will have to redo most of the Intrusion Kill Chain to gain access to a new C2 window.

5. Performance evaluation

Apart from the security assessment presented above, we also did a preliminary performance evaluation. However, it is necessary to consider that this evaluation relies significantly on the functioning of our PoC, far from being at industry-level expectations. Not only that, but the results we will present below also depended on the NFV-MANO solution used, which in our case was OSM.

With this in mind, we start this section by specifying the methodology we used to attain our results. Then, those results are subdivided into two subsections, each for each "realm" tested. In the first, we present our system's impact on the management plane, while in the last one, we showcase some results regarding the user plane.

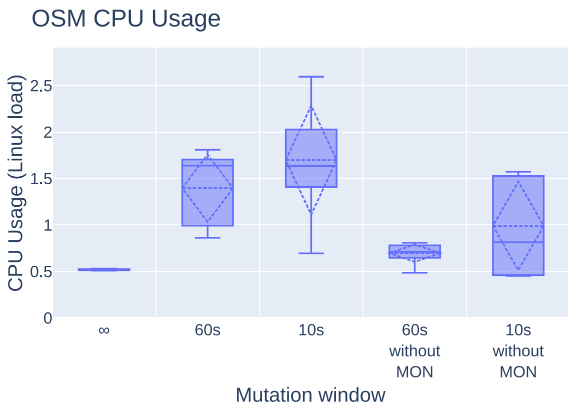
5.1. Methodology

In this set of tests, we reused some elements of the security evaluation methodology, as specified in Section 4.2. We used the same MTD PoC, as well as the same machines' setup, whose specifications we presented in Section 4.1. Furthermore, we also adapted the CNF corresponding to the web application with multiple distinct versions. However, we modified it to have five versions of the Java implementation. This decision stems from this evaluation's objective of analyzing the performance issues our MTD mechanism might introduce. Having multiple distinct versions would introduce an undesirable variable into our tests. Since each version impacts the overall system's performance differently, separating each version's impact from the MTD system would be difficult. Therefore, only using one allows us to easily distinguish between what is caused by the application and the MTD mechanism.

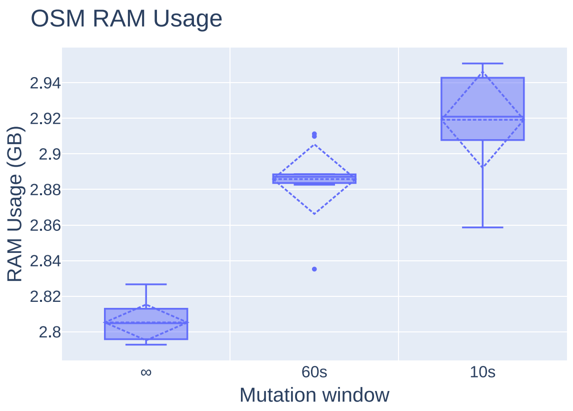
In these evaluations, we also created a process responsible for setting up all of our environment, commanding OSM to launch our CNF when necessary. Therefore, for each plane's results, our process loops throughout each mutation window size and conducts a total of 10 trials for each one. Each trial starts by commanding OSM to launch the CNF with the MTD system defined with the proper mutation window size. Then, it waits 10 min, corresponding to each trial's period. For the user plane performance evaluation, this script launched Locust,[6] a Python-based tool that facilitates running stress tests on web systems. In this evaluation, our Locust setup simulated 100 users spawning at a rate of 0.1 s, making continuous requests to the web application, i.e., making each request right after the previous one finishes. As for the management plane evaluation, our process gathered all the metrics related to the CPU, RAM, and network usage from two kube-prometheus[7] installed on both Kubernetes clusters. Kube-prometheus is an open-source project that allows collecting metrics of a Kubernetes cluster. At the end of those 10 min, the script reloaded OSM, relaunching all its necessary components, waited 5 min, and started a new experiment cycle.

6 Locust project page: https://locust.io/.

7 Kube-prometheus project page: https://github.com/prometheus-operator/kube-prometheus.



(**a**) CPU usage considering and not considering the OSM's Monitoring (MON) module.



(**b**) RAM usage.

Fig. 15. Comparative computation performance metrics between using and not using MTD.

5.2. *Management plane*

On the management plane evaluation, we obtained metrics related to OSM's computation performance and the network impact on the Kubernetes cluster where our CNF was installed. Starting with the first set, we present the average CPU and RAM usage of OSM throughout 10 min on the graphs of Fig. 15. On each graph of this figure, we provide an impact comparison between using and not using MTD (represented by the infinite mutation window) to protect a CNF for one simultaneous version. Furthermore, we only tested the extreme cases for one simultaneous version because we directly obtained relevant results with them. Therefore, considering the results from Section 4.3.1, we tested the 60 and 10-s mutation window sizes.

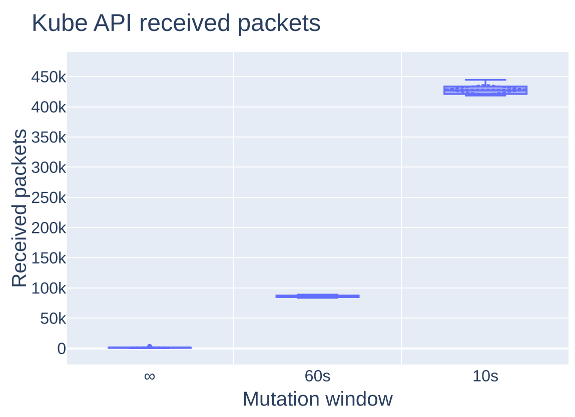
From the box plot of Fig. 15(a) relative to the CPU usage, we understood that there was a considerable negative impact when using our MTD PoC. For a window of 60 s, OSM required about 169.23% more processing power than when not using MTD. As for the 10 s mutation window, we only observed a 21.43% increase in relation to the 60 s window. After analyzing each OSM component's individual contribution to the CPU total usage, we noticed that most of the processing power was being consumed by the Monitoring module. We found that this behavior was due to the way this module works. When we launch some CNF with autoscaling based on metrics values, which was the case with our MTD PoC, the Monitoring module is instructed to verify any changes to that metric's value. However, the way it does so is rather intensive. From time to time (in our case, each second, since we needed as much precision as possible), it analyzes the Prometheus system for any modification. Then, when it detects an important change, it alerts the OSM module responsible for scaling the CNF. Furthermore, it will continue sending this alert until it observes that the metric value changes to a new one. The combination of these factors leads to this performance inefficiency. As demonstrated in Fig. 15(a), if we discard the CPU usage of this module, our MTD PoC has far less impact: the MTD set with a 60 s mutation window only consumes about 24.64% more CPU than the non-usage of MTD, while the 10 s window uses about 30.30% more processing power than the 60.

A potential solution to fix this problem on the Monitoring module would be to use a broker to receive all metrics, which may then be persisted through agents. Using a broker is a better solution than Prometheus, at least for this case, as it is the best fit for handling live data, while Prometheus is more adapted to store historical data. OSM could still use Prometheus as a time series database, which would obtain the necessary data from the broker. Nevertheless, its benefit would be that this module would not need to periodically query Prometheus directly for changes, as it would be alerted by the broker when a new value was received.

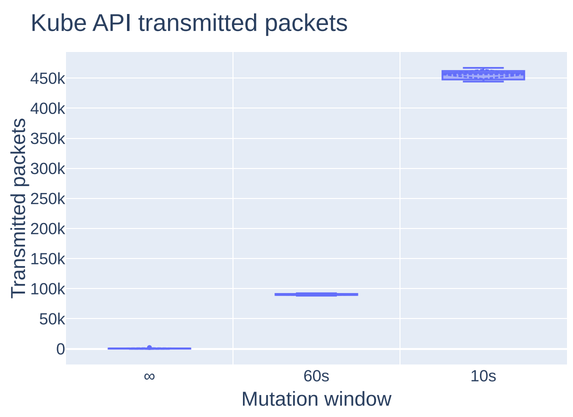
As for RAM usage, whose results we presented in the plot of Fig. 15(b), although, at first sight, we can also notice an increase with the decrease in the mutation window size, it has less to no impact. On the one hand, the average OSM's RAM usage when orchestrating and managing our CNF without MTD is approximately 2.81 ± 0.01 GBs. Conversely, for the 10 s mutation window, the RAM consumption is roughly 2.92 ± 0.03 GBs, an increase of only 0.11 GBs. Therefore, we can conclude that our MTD PoC has practically no impact on the RAM usage of OSM.

Finally, regarding management network usage, we measured the number of packets and MBs received and transmitted by the Kube API Server on the Kubernetes cluster where the CNF was instantiated. These results are showcased in the plots of Fig. 16. We did this analysis because OSM communicates with this Application Programming Interface (API) to manage any instantiated CNF. This communication happens when it requests the instantiation or removal of a CNF and, in our PoC, for scaling the application's versions.

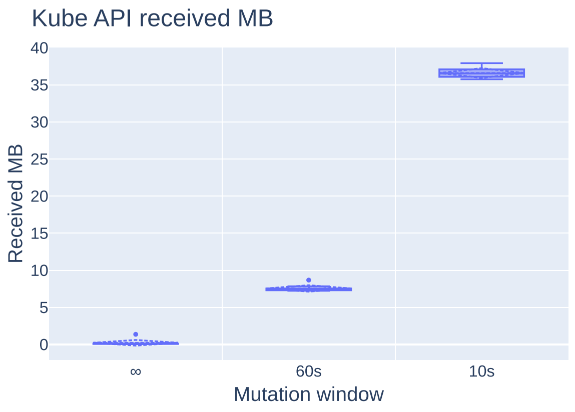
Regarding the received and transmitted packets, the results met our expectations. In broad terms, the number of packets that OSM sends and receives to and from the Kube API Server grows with the decrease in the mutation window size used, as observed in the graphs of Figs. 16(a) and 16(b). This behavior is easy to explain: our MTD PoC will make OSM trigger more scaling operations when we have smaller window sizes. These operations then translate into OSM requesting the Kubernetes cluster to launch a specific



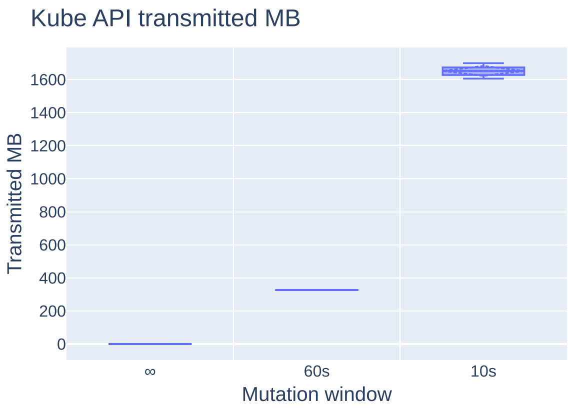
(**a**) Received packets.



(**b**) Transmitted packets.



(**c**) Received MBs.



(**d**) Transmitted MBs.

Fig. 16. Comparative network performance metrics between using and not using MTD.

version of our application and remove the previous one. With more of these requests, more packets travel over the management network. Furthermore, as expected, the number of packets sent and received are almost the same among windows of the same length.

However, the results about the number of MBs transferred were surprisingly different from our predictions. For the MBs received by the Kube API Server during our experiment of 10 min, we observed an increasing pattern similar to the number of received packets. Although there are many MBs received for a mutation window size of 10 s, they only totalize about 36.58 ± 0.66 MBs during 10 min, compared to the 0.38 ± 0.38 MBs received when we tested the non-protected CNF. The overwhelming results were related to the transmitted MBs by the Kube API Server. Before realizing this evaluation, we anticipated that the transmitted MBs would roughly be similar to the received quantity. However, as depicted in the plot of Fig. 16(d), we obtained an average of more than 1.5 GB transmitted by the Kube API to OSM over 10 min for the 10-s mutation window. This result is especially concerning in an edge scenario, where our MANO is probably in a cloud, managing distant edge clusters.

Therefore, we searched for the problem that might be causing this extreme bandwidth consumption. To do so, we carefully examined all the OSM code related to a scaling operation, tracking all accesses to the Kube API Server. Then, we did all these calls separately while tracing the network usage using tcpdump[8] at the same time. With this, we discovered the source of the problem. In any scaling operation, OSM uses Helm to list[9] all Helm releases installed on the analogous Kubernetes cluster without any filter. OSM does this since it needs to detect if the CNF is still installed in the cluster and to get the namespace where it is running. We observed that the Kube API Server sent about 7 MBs of data to OSM on each call. However, this solely was not the complete cause of our problem. For each scaling operation, OSM runs this command twice in a row. First, when it needs to find how many instances of the CNF are running, and the other, to achieve the scaling itself. Therefore, the Kube API Server will send about 14 MBs of data to OSM for each scale. Since when each mutation window starts, our MTD PoC requests the scaling-out of a new version and the scaling-in of the previous one, each new mutation window is related to roughly 28 MBs of data transferred to OSM. With this in mind, we can understand why so many MBs were transferred to OSM when we used a 10 s mutation window for 10 min.

[8] Tcpdump project page: https://www.tcpdump.org/.
[9] Helm list documentation page: https://helm.sh/docs/helm/helm_list/.

As demonstrated, OSM is not yet correctly fit and, therefore, optimized for edge scenarios. This fact is understandable, as the discussion and practical achievement of MEC using NFV is still somewhat recent. Nevertheless, in our vision, it could easily be fixed not to have this problem. First, instead of making a Helm list of all installed releases, OSM could use this command with a filter for the CNF in question. Furthermore, it could be optimized not to run this command twice a row since the output is the same. Moreover, there also are much less intrusive ways of getting the information if a CNF is still launched in Kubernetes and the namespace where it is. For example, OSM could use Kubernetes Labels[10] to query for any installed application efficiently, and the amount data returned by the query would be much smaller than the current method.

5.3. User plane

Regarding the user plane performance evaluation, we simulated 100 users making continued requests to our CNF over 10 min. From the obtained results, we created the box plots represented in Fig. 17 regarding the percentage of failures, the number of requests per second, and the average response time. In each graph, we present those metrics for the multiple mutation window sizes we tested within the security evaluation in Section 4 and for the CNF without any protection to have a baseline to compare the results. Furthermore, we also obtained results for one and two simultaneous versions. Considering the results obtained in Section 4.3.1, we only evaluated the user plane performance against mutation window sizes between 10 and 60 s for one simultaneous version and 20 and 60 for two simultaneous versions. The first is showcased on the left side plots of Fig. 17, while the other is on the right side.

Beginning with the failures percentage, the results shown in Fig. 17(a), for one simultaneous version, and Fig. 17(b), for two, demonstrate that it grows with the decrease in mutation window size. However, the growth rate is smaller for two simultaneous versions. Although counterintuitive at first sight, the explanation for this result is that with two simultaneous versions, when a mutation window changes, there is a short period where more versions are active than with one. With two simultaneous versions, four versions are being executed after the scaling-out of the two new versions. Therefore, Kubernetes will balance new connections among all these four. In contrast, when the MTD PoC is set for one simultaneous version, there will be only two versions in this time interval. Consequently, more clients will be connected to each version when only two versions are instantiated. That means that, in most cases, when OSM starts the scaling-in process of the older versions, more clients' connections will be dropped when MTD is set for one simultaneous version rather than two. Nevertheless, these failures have more to do with our MTD system implementation still being a PoC. As previously explained in Section 3.4, the system shall use a proxy in front of the application to prevent connection loss between mutation window changes. With a proxy, any client connecting with the application will do so with the proxy and not with each instantiated version. However, as demonstrated in Appendix A, our implementation still does not have one. Still, the percentage of failures we obtained is somewhat low, only about $0.24 \pm 0.03\%$ for the 10 s window of one simultaneous version and $0.12 \pm 0.01\%$ for the 20 s window of two simultaneous versions.

Concerning the number of requests per second, although it decreases with the mutation window size, this decrease is not very pronounced, as demonstrated on the box plots of Fig. 17(c), for one simultaneous version, and Fig. 17(d), for two. In the worst case, the number of requests dropped by about 4.12% for one simultaneous version, from the non-protected CNF to the CNF protected by our MTD implementation with a 10 s mutation window size. Likewise, it dropped approximately 3.10% using the MTD system with a 20 s mutation window for two simultaneous versions. Similarly, the average response time also worsens with the decrease in the mutation window size, but it never reaches concerning levels. For the 10 s window and one simultaneous version, we observed an increase of roughly 7.48% in relation to the non-protected application. As for the 20 s window and two simultaneous versions, the inflation was around 5.79%.

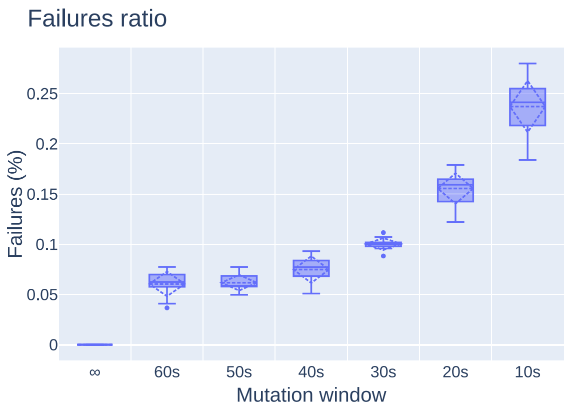
With these results in mind, we can say that our solution does not have much impact on the performance from a client's point of view. Although we observed failures in some connections, they only happened within mutation window changes. As detailed in our theoretical solution, we believe this issue can be solved in future works by using a proxy.

6. Conclusion

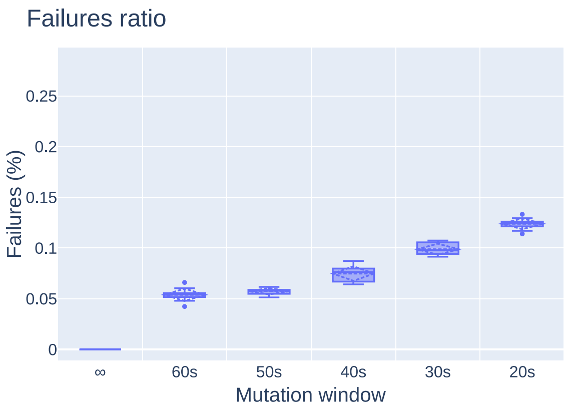
The main novelty of this work was the creation of an MTDaaS methodology to protect CNFs in NFV-based ecosystems, which include MEC in NFV and potential 5G-IoT. With such a system, Network and Mobile Network Operators can offer the protection of MTD as a security product, which any CNF provider can choose to use. Therefore, this protection system is agnostic to the application it will protect, and we successfully demonstrated its efficacy in defending a CNF against zero-day-based attacks.

To achieve these results, we defined an MTD method based on the key design concepts delineated by Cai et al. [28] by answering the following questions. On what-to-move, we established that the target of movement would be an application's versions, where each version had the same functionalities. Regarding when-to-move, we followed a proactive MTD approach, where the system moves from time to time, regardless of being under attack or not. Finally, we used all three techniques described in [28] for the how-to-move decision. Shuffling was applied by continually changing the running version of an application, diversity by requiring applications to have more than one version, and redundancy by allowing more than one version to be executed simultaneously. In addition to these characteristics, we modeled the proposed MTD system to tolerate being set up as more advantageous for each client as possible. To achieve that, we defined our method as parametrizable, allowing an application's provider to select the mutation window size and the number of simultaneous versions to be considered while protecting that application.

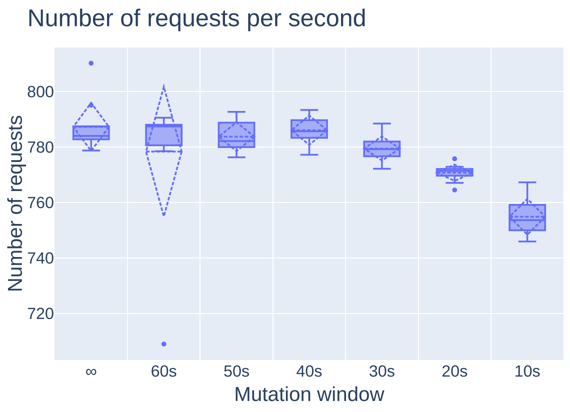
[10] Kubernetes Labels and Selectors documentation page: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/.



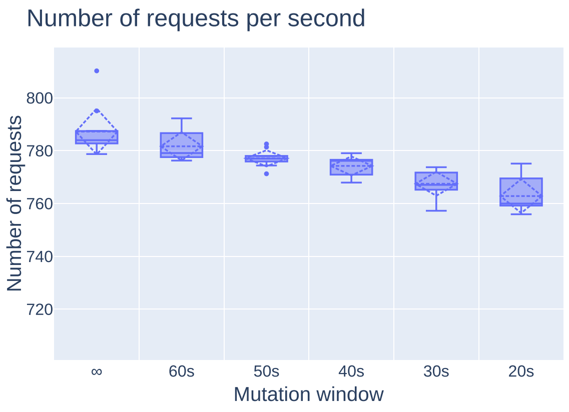
(a) Percentage of failures for one simultaneous version.



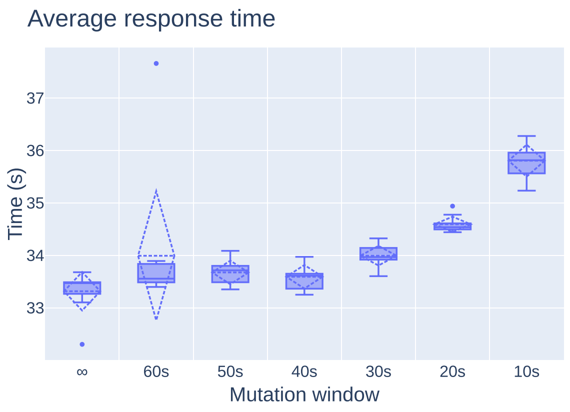
(b) Percentage of failures for two simultaneous versions.



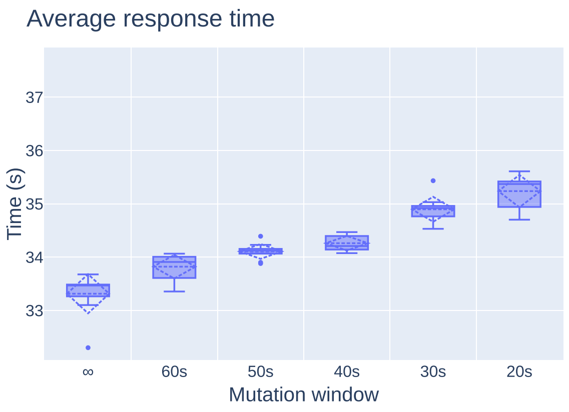
(c) Requests per second for one simultaneous version.



(d) Requests per second for two simultaneous versions.



(e) Average response time for one simultaneous version.



(f) Average response time for two simultaneous versions.

Fig. 17. Communication with the protected CNF-related metrics.

In the end, we created a PoC based on OSM as the NFV-MANO framework to demonstrate the practical significance of this system. With it, we were able to conduct a security evaluation. In this respect, we also innovated by performing a thorough analysis based on the Intrusion Kill Chain phases. With this evaluation, we concluded that the system could trick a potential attacker when inspecting a protected application during the Reconnaissance phase, rendering the functionality of some well-known tools, such as Nmap, questionable when this method is used to protect them. In this phase, if the intruder cannot grasp the meaning of such "strange behavior", it may be discouraged to proceed with the attack. However, if it continues, the results demonstrated that the attacker would have further difficulty in the Exploitation, Installation, and C2 steps. As the mutation window size decreases, the attacker has less success in achieving the first two phases. Furthermore, it cannot proceed to the C2 phase if it cannot complete them. However, even if a malicious actor successfully installs a backdoor on the target system, our results show that the C2 window will depend on the mutation window size being used. This size will, therefore, constrain the attacker only to have a short period to achieve its objectives with the intrusion because when the mutation window ends, the installed backdoor is also removed.

Moreover, we also studied the performance impact of the proposed system. However, this evaluation was limited by the MANO framework we utilized and our still being a PoC, lacking certain critical aspects of our theoretical methodology. Regarding the management plane performance, we observed a considerable degradation of the OSM's processing performance when using smaller mutation windows, requiring much more CPU usage than when not protecting the same application. Another problem we discovered was the amount of data transmitted by the Kube API Server to OSM in each scaling operation, a concerning issue for edge scenarios. Nevertheless, we pinpointed the cause of these issues in OSM and provided solutions that can be resolved in future works. In terms of the RAM used by OSM and the data received by the Kube API Server, although we found a degradation, it is negligible. As for the user plane performance, the biggest issue was the connection failures between clients and the tested CNF. However, as described in our theoretical solution, this problem would be fixable if our PoC contemplated a proxy in front of the protected applications.

There are, however, some future directions this work may take. For example, the developed methodology can offer better protection if combined with other MTD techniques. These techniques may not only pass by combining with other MTD types, such as Dynamic Networks methods but also with other movement techniques. For instance, our system could combine proactive and reactive MTD to offer a more effective defense. Moreover, we designed our solution with stateless applications in mind, as they are the most common in cloud settings. However, it is also essential to protect stateful applications. Therefore, another possible future work might be to analyze how our MTDaaS methodology could be adapted to protect them and its potential caveats. An additional potential future work is the extension of our analysis, primarily the performance-related. For the management plane, we need to assess further the impact particular components of OSM have on our system and prove that the pointed issues are the ones causing most of the performance losses. As for the user plane, we could study specific user behavior's impact on the overall system by tinkering with the Locust setup parameters.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the European Regional Development Fund (FEDER), through the Regional Operational Programme of Lisbon (POR LISBOA 2020) and the Competitiveness and Internationalization Operational Programme (COMPETE 2020) of the Portugal 2020 framework [Project NETEDGE with Nr. 069977 (POCI-01-0247-FEDER-069977)].

Appendix A. Building the proof of concept

With all the considerations taken in Section 3, we created a PoC to assess the proposed MTDaaS solution. Since the present study focused on the enhanced NFV architecture where containers are used to realize VNFs, we have chosen OSM as the MANO framework and Kubernetes as the container manager and orchestrator. However, although OSM is an open-source project hosted by ETSI, it strictly does not follow the ETSI's NFV references. This unfortunate characteristic, which is not only specific to OSM but probably all MANO implementations, impacted our PoC implementation.

Therefore, in Appendix A.1, we start by presenting the OSM-induced limitations, considering the theoretical blueprint we have provided in Section 3. Then, in Appendix A.2, regarding the outlined considerations, we analyze the integration of our PoC with OSM. In the end, in Appendix A.3, we finally present the overall implementation process and peculiarities of our MTD PoC.

A.1. Limitations caused by the used MANO solution

As mentioned above, OSM is not entirely aligned with the ETSI NFV-MANO references. Therefore, we needed to modify the original MTD proposition to function correctly within the specified scenario. First, although the ETSI references imply that a VNFM is a "plug and play" module, OSM conducts most lifecycle management operations through its Lifecycle Management (LCM) component. Therefore, providing a proprietary VNFM to perform all its expected functionalities when using OSM is impossible. We can think of the OSM's LCM module as being a generic VNFM, considering the ETSI's report on the NFV architectural options [25]. The OSM's solution for VNF-specific VNFMs was to allow users to create Execution Environments (EEs) whose primary function is to deploy and update all the VNFs or CNFs' software. This function, however, is only a tiny portion of all the VNFM ones defined by ETSI [22].

With that said, there was a problem: how can we command the scaling of a CNF using a VNFM? In other words, the proposed MTD Controller VNFM, created and provided by the Operator, was idealized as the entity which would command the NFVO to scale a CNF's versions to realize the MTD functionalities. However, in OSM, this is impossible, i.e., there is no direct way of creating an EE to command some component of OSM to scale a CNF. However, an EE can also be employed to monitor NFs, creating custom metrics. Although the metrics generation behavior is related to the NF's monitoring, OSM does not enforce it, meaning an EE can produce metrics "as pleased".

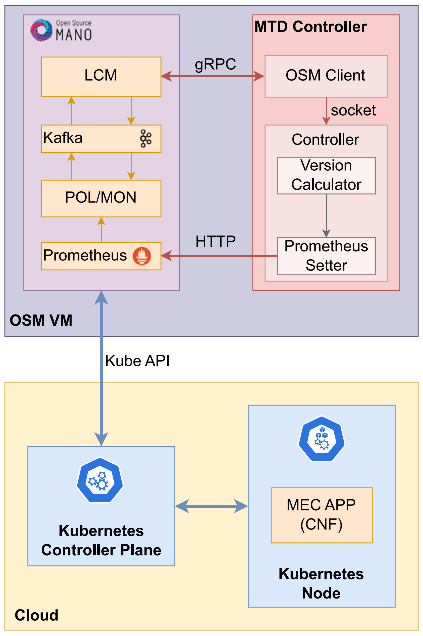


Fig. 18. High-level architecture of the integration of the MTD PoC with OSM in a MEC scenario.

In OSM, it is also possible to define the auto-scaling of VNFs, although in a different way from what ETSI conceptualized. ETSI initially intended for the VNFM to trigger the auto-scaling process by analyzing infrastructure events generated by the VIM or the VNF and its Element Management (EM). The first type of event is covered by the OSM's LCM module. However, the second type of event is not directly monitored by the LCM module but rather by a potential EE. As described before, an EE in OSM can access NF instances and generate metrics from their events. However, this EE, unable to directly trigger the scaling of a CNF, needs to publish those metrics to the OSM metrics collector, which happens to be a Prometheus instance. Then, OSM has two modules, one called Monitoring (MON) and the other called Policy (POL), the first responsible for gathering those metrics from Prometheus and the second for sending an alarm to the LCM when those metrics correspond to the scaling of a VNF. These alarms are created considering the thresholds defined by the CNF descriptors when the provider who launched them requires auto-scaling.

Considering all this information, we can have a nearly equivalent third-party VNFM in OSM as an EE that controls a CNF scaling. To achieve that, the EE needs to post some metric data to the OSM Prometheus. Then OSM will automatically scale the corresponding CNF instances according to the rules given within its descriptor. Furthermore, when we started working on this proposal, OSM only allowed the auto-scaling of VNFs, which led to the proposal and development[11] of that feature for CNFs, to complete this MTD PoC.

A.2. Integration with OSM

Considering the limitations imposed by OSM, we created an MTD Controller that followed them as closely as possible. In Fig. 18, we provided a high-level architecture of its interactions with the OSM components. As observed, we divided the MTD Controller into two distinct pieces: the OSM Client, responsible for the interactions with OSM through Remote Procedure Call (RPC), and the MTD Controller itself. The Client functions as a proxy between OSM and the MTD Controller, working as a driver for OSM. Our idea is that, in the future, if needed, we can create another driver for other orchestrators or MANOs, and the MTD Controller will work just fine. In any case, when OSM needs to execute a primitive towards the MTD Controller EE, the LCM module makes an RPC to the corresponding procedure made available by that OSM Client module. When this happens, the OSM Client process communicates with the Controller module through a Linux socket, sending all the primitive details needed for the Controller's internal logic.

Furthermore, the referred primitives OSM will execute towards the MTD Controller are defined within the CNF descriptor. There, we established them as scaling primitives. Therefore, when a CNF in these conditions is scaled, OSM executes the *scaled* primitive towards the MTD Controller EE's Client module, as represented in Fig. 18. Through this primitive execution, the LCM module is able to send to the OSM Client information related to the scaling operation that was or will be performed, information that the Client next sends to the Controller. The scaling operation-related data that OSM sends only contains the NS and CNF IDs, the scaling operation, and the scaling moment. The scaling operation notifies if the scaling operation is inwards or outwards. In contrast, the scaling moment informs of when the scaling primitive takes place, i.e., if OSM will execute it before or after the scaling operation.

[11] Auto-scaling of CNFs feature proposal and design: https://osm.etsi.org/gitlab/osm/features/issues/10938.

When the Controller decides it is time for scaling out or in some version of the application running through the CNF, it will expose the corresponding metric value through a HTTP server. This server is made available to the OSM Prometheus while instantiating the corresponding CNF so Prometheus can scrape it occasionally. Then, the LCM module will create an alarm in POL, which will generate an alarm in MON as defined by the CNF descriptor for each metric associated with a scaling operation. Therefore, as represented in Fig. 18, the MON module will notice when Prometheus provides a new CNF metric, hence notifying POL, which will request LCM to scale the corresponding CNF version. From this scaling request, LCM will use the Kube API Server to request the Kubernetes Control Plane, i.e., the CISM, to perform this scaling operation. When this happens, LCM will execute the *scaled* primitive towards the MTD Controller before or after the scaling operation, depending on the CNF descriptor's definition.

A.3. Controller logic and functioning

Since the created MTD Controller is an OSM EE, it will only be executed when a CNF it protects is instantiated. Therefore, the instantiation moment is somewhat crucial for the overall process. However, some essential steps must be considered before the instantiation is triggered.

First, for the described auto-scaling to work, it is necessary to define the corresponding thresholds for each CNF version within its descriptor. Therefore, in this work, we created a standardized way of expressing these parameters under each protected CNF descriptor, as demonstrated in Listings 1, 2, and 3. First, it is necessary to start with the decision of what are the CNF versions to take into account, as demonstrated in Listing 1. Each resource name represented in the listing is related to the corresponding application's version Kubernetes Deployment. For each defined version, it is also required to determine the minimum number of instances as zero and the maximum as one. Hereupon, it is necessary to decide on the usage of the MTD Controller EE, as represented in Listing 2. In it, we defined the Helm Chart corresponding to the created MTD Controller and the monitoring parameter that EE will make available to the OSM Prometheus. Finally, the last defined component is the definition of the scaling-related aspects, as presented in Listing 3, where it is demonstrated the scaling aspect for one of the versions. In it, it is necessary to define three main aspects: the primitive that OSM needs to execute towards the MTD Controller, as well as the related scaling moments and actions, i.e., if the primitive is to be triggered before or after the scaling operation, and if it will be related to the scale-in or -out process; the scaling policy, where it is defined the threshold of the metric value associated with when OSM needs to scale out or in the corresponding version, and; the delta related to that scaling aspect, where it is selected the number of instances to scale.

Listing 1: Definition of the CNF versions to be instantiated.

```
kdu-resource-profile:
- id: app-scale
  kdu-name: mec_app
  resource-name: mp1
  min-number-of-instances: 0
  max-number-of-instances: 1
- id: app-scale-python
  kdu-name: mec_app
  resource-name: mp1_from_python
  min-number-of-instances: 0
  max-number-of-instances: 1
- id: app-scale-ubuntu
  kdu-name: mec_app
  resource-name: mp1_from_ubuntu
  min-number-of-instances: 0
  max-number-of-instances: 1
```

Listing 2: Definition of the MTD Controller EE.

```
lcm-operations-configuration:
  operate-vnf-op-config:
    day1-2:
    - id: mec_app
      execution-environment-list:
      - id: mtd-ee
        external-connection-point-ref: mgmt-net
        helm-chart: mtd-chart
        helm-version: v3
        metric-service: mtd
monitoring-parameter:
- id: mtd
  name: mtd
  performance-metric: my_metric
```

Listing 3: Definition of one of the CNF's versions' scaling aspect.

```
scaling-aspect:
- id: scale-kdu
  name: scale-kdu
  max-scale-level: 10
  scaling-config-action:
  - id: mec_app
    config-primitive-list:
    - trigger: pre-scale-in
      vnf-config-primitive-name-ref : scaled
    - trigger: pre-scale-out
      vnf-config-primitive-name-ref : scaled
    - trigger: post-scale-out
```

```
      vnf-config-primitive-name-ref : scaled
  scaling-policy:
  - cooldown-time: 5
    name: scale-policy-app
    scaling-criteria:
    - name: app-scale-in
      scale-in-relational-operation: EQ
      scale-in-threshold: -1
      vnf-monitoring-param-ref: mtd
    - name: app-scale-out
      scale-out-relational-operation: EQ
      scale-out-threshold: 1
      vnf-monitoring-param-ref: mtd
    scale-in-operation-type: OR
    scale-out-operation-type: OR
    scaling-type: automatic
  aspect-delta-details:
    deltas:
    - id: kdu-delta
      kdu-resource-delta:
      - id: app-scale
        number-of-instances: 1
```

Considering these listings, when a client wants to protect its CNF, it can create a descriptor by adapting the above rules. In the case of the descriptor's section corresponding to Listing 1, it only has to create a resource profile for each of its application's versions. On the other hand, Listing 2 can be maintained for any CNF descriptor. As for Listing 3, it shall be repeated for each version. For instance, if the CNF has three versions, this listing shall be replicated in the corresponding descriptor three times. The provider only needs to modify two details. First, it needs to change the value of the *scaling-aspect:aspect-delta-details:deltas:kdu-resource-delta:id* key to the corresponding version that replica represents. The other modification is the values corresponding to the scaling thresholds, which in Listing 3 have the value -1 and 1 for the scale-in and scale-out operations, respectively.

Another important decision made while creating this PoC was the scaling threshold values. This decision was essential because the MTD Controller needs a consistent mechanism to create the metric value used by OSM to know when to scale each application's version. Therefore, for each version, the envisioned method was to have a number that represents it, which begins in one, and that for each new version added to the CNF descriptor, is incremented by one. For example, if we have three versions defined within the CNF descriptor, they will be represented by numbers one, two, and three. Then, for each version, its associated number is used for the scaling thresholds: for the scale-out threshold, we used the actual number, while for the scale-in threshold, we used its negative form. For instance, if we are defining the scaling aspect of the version represented by the number two, its scale-out threshold will be two, while its scale-in threshold will be minus two. With this definition, the MTD Controller can easily infer the thresholds corresponding to each version by having the information on how many versions there are.

A further important aspect considered while designing this MTD PoC was when the LCM module needed to run a primitive towards the MTD Controller concerning the scaling operation and its moment. As mentioned earlier, in OSM, it is possible to define the execution of an EE primitive before or after a scaling operation. As depicted in Listing 3, we specified that the LCM module should trigger the *scaled* primitive before any scaling-in and -out operations and after any scaling-out operation. We followed this approach because we needed to find a way for OSM not to try scaling a specific version continually. In other words, if the MTD Controller, at some moment, specified the scaling-out of version one, it would set the metric value to one. Then, after some time, the MON and POL modules would perceive that metric value, sending an alarm to LCM to scale out that version. However, if, for example, the mutation window defined was sufficiently large, and the metric value was not updated, holding the value one for the whole window, MON and POL would repeatedly send an alarm to LCM to scale that version. Each time this happened, the LCM module would verify that version's maximum number of instances was one, as defined within the CNF descriptor, and the scaling operation would fail. Therefore, this behavior would overwhelm OSM, especially the LCM module's performance. As such, we defined the execution of the *scaled* primitive before any scaling operation. Each time the MTD Controller executes that primitive on such occasion, it receives the information regarding the scaling action and moment. The execution of this primitive also gives the indirect information that OSM already acknowledged that it is supposed to scale the selected version. With this information, the MTD Controller sets the metric to zero, leading OSM to do nothing regarding scaling operations, avoiding the described performance issues. Moreover, as demonstrated in Listing 3, the *scaled* primitive is executed after a scaling-out process. The reason is to let the MTD Controller acknowledge that the scaling-out operation already took place and that the MTD Controller shall remove the previous running version, therefore setting the metric value to the negative form of the earlier version number.

Furthermore, these triggers have also another finality. We also used them for locking the MTD Controller from setting a new metric value before receiving the confirmation that each scaling operation was indeed received with success from OSM. This characteristic is essential since, without it, the MTD Controller may post new metric values faster than OSM can acknowledge them, especially when using smaller mutation windows. Therefore, that lock is released whenever the *scaled* primitive is executed after a scaling-out operation and before a scaling-in operation. It would make more sense to release the lock before the scaling-out process occurs, not after, since that marks when OSM acknowledges a new metric value. However, this would lead to a performance issue. To understand it, we must remember that OSM only scales one instance of a CNF or VNF at a time to avoid Time-of-check Time-of-use (TOCTOU) situations. Now, consider the MTD Controller released the lock just before a scaling-out operation occurred. Since the scaling-out procedure takes some time, a new metric value would stay in Prometheus for that time. This delay is because OSM takes a while to complete the current scaling-out operation and then read and perform the necessary actions related to the new metric value. Since the metric value would not be changed for a while, the MON and POL OSM modules would continually signal LCM to begin the corresponding scale operation. These signals would be queued as LCM would still be scaling out a previous

CNF version. When the LCM module finally completed that scale-out, it would be overwhelmed with all those signals, affecting its performance. Therefore, we found that releasing the lock just after the scaling-out operation takes a smaller burden on the OSM's overall performance.

To summarize, the sequence diagram of Fig. 19 represents the high-level functioning of the presented MTD Controller PoC and its interactions with OSM. To this extent, the MTD Controller starts by acknowledging the client's parameters. In this example, the number of versions is three, and the number of simultaneous versions is one. Then, after initializing, the MTD Controller starts the endless loop, as described in Section 3.3. In this case, after some iterations, a mutation window where version two was instantiated finishes, and the MTD Controller needs to find a new version to replace it. To achieve that, the Version Calculator component starts by obtaining all non-running versions, which in this example, are versions one and three. Then, it chooses one of them randomly, ending up with version one and selecting it to replace the previous version. With the newly chosen version, the Version Calculator instructs the Prometheus Setter module to set the metric value to one and starts to sleep for a period corresponding to the established *time_window*.

With this instruction, OSM, through MON and POL modules, will gather this new value from Prometheus and instruct LCM to scale out version one of the analogous CNF. However, before scaling out the new version, OSM will command the execution of the *scaled* primitive towards the MTD Controller. Only then will it trigger the Kubernetes Control Plane to scale out this new version. After the acknowledgment that the scaling operation will occur, the MTD Controller will set the metric to the value zero, which, as explained, instructs OSM not to perform any scaling operation.

After the Kubernetes Control Plane completes the actions corresponding to the scaling-out operation, it returns the finalization acknowledgment to OSM. From this point, OSM triggers once again the execution of the *scaled* primitive within the MTD Controller. By receiving the information that version one was scaled out successfully, the MTD Controller sends a command to scale in the previous version to OSM, i.e., it sets the metric value to minus two, instructing OSM to scale in version two. When OSM reads this new value from Prometheus, it sends an acknowledgment to the MTD Controller, continuing its execution by instructing the Kubernetes Control Plane with the necessary commands to scale in version two of the corresponding CNF. Finally, the MTD Controller, after receiving the order to execute the *scaled* primitive related to the pre-scaling in operation, will set the metric value to zero to avoid misguiding OSM.

Appendix B. Attacking the vulnerable version without protection

This section will present the steps an attacker could take on each Intrusion Kill Chain phase to attack the web application's vulnerable version. To provide these results, we instantiated the presented vulnerable version as a CNF using OSM without the MTD Controller PoC's protection.

B.1. Reconnaissance

This phase is characterized by identifying the desirable targets and gathering the most information possible about them. The attacker later uses this information to plan the actual attack on the system.

One of the first steps most attackers take when presented with some server is to discover all the open listening ports and, from there, the related services and applications. Therefore, considering that our sample CNF would be deployed in a server whose address would not be confidential, as legitimate end clients need to connect to its applications, the attacker would also know that address. For test purposes, considering that the targeted server address is *10.0.13.243*, the malevolent actor could, for instance, execute the *Nmap*[12] tool to scan that server and discover all the open ports, as demonstrated in Listing 4.

Listing 4: Nmap scan of the server where we installed the vulnerable CNF.

```
$ nmap -sV -Pn 10.0.13.243
Starting Nmap 7.92 ( https://nmap.org ) at 2022-10-05 15:38 WEST
Nmap scan report for 10.0.13.243
Host is up (0.034s latency).
Not shown: 996 closed tcp ports (conn-refused)
PORT      STATE SERVICE     VERSION
22/tcp    open  ssh         OpenSSH 8.2p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
8000/tcp  open  http        Apache Tomcat 8.5.82-jre11
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 100.36 seconds
```

From this scan, we found that ports *22* and *8000* were open. However, in a real environment, it would be more likely that more ports would be opened since, most likely, there will not exist any use case where each server has only one CNF deployed. In this test, we also discarded port 22, as it does not belong to any CNF, being used just for management purposes. As for port 8000, Nmap specifies that it is a TCP port that makes available an HTTP service, hence being a web server. It also tells the web server is based

[12] Nmap project page: https://nmap.org/.

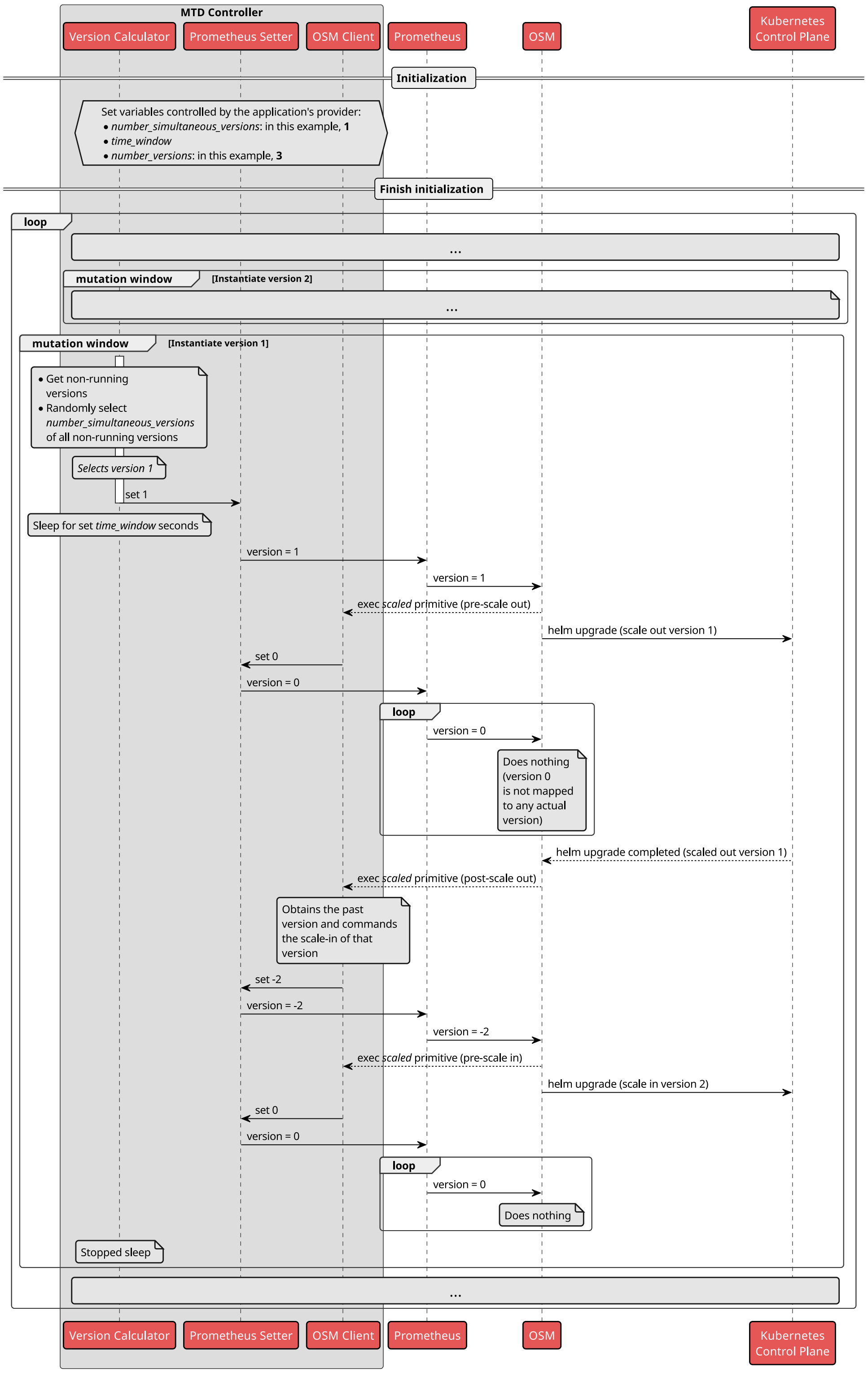
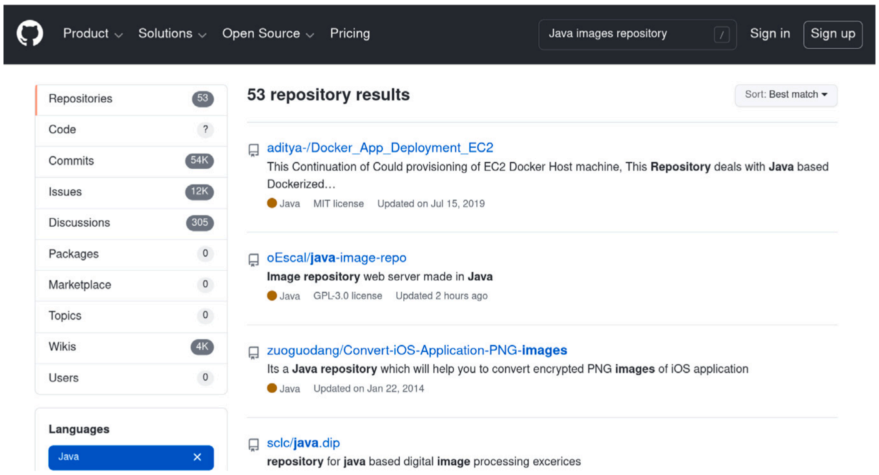

MTD Controller
Version Calculator
Prometheus Setter
OSM Client
Prometheus
OSM
Kubernetes Control Plane
Initialization
Set variables controlled by the application's provider:
• number_simultaneous_versions: in this example, 1
• time_window
• number_versions: in this example, 3
Finish initialization
loop
...
mutation window [Instantiate version 2]
...
mutation window [Instantiate version 1]
• Get non-running versions
• Randomly select number_simultaneous_versions of all non-running versions
Selects version 1
set 1
Sleep for set time_window seconds
version = 1
version = 1
exec scaled primitive (pre-scale out)
helm upgrade (scale out version 1)
set 0
version = 0
loop
version = 0
Does nothing (version 0 is not mapped to any actual version)
helm upgrade completed (scaled out version 1)
exec scaled primitive (post-scale out)
Obtains the past version and commands the scale-in of that version
set -2
version = -2
version = -2
exec scaled primitive (pre-scale in)
helm upgrade (scale in version 2)
set 0
version = 0
loop
version = 0
Does nothing
Stopped sleep
...

Fig. 19. Sequence diagram of the functioning of the proposed MTD Controller PoC and its interactions with OSM.

on Apache Tomcat with version 8.5.82, and the corresponding application is being executed in a Java Runtime Environment (JRE) with version 11.



Java images repository

Repositories	53
Code	?
Commits	54K
Issues	12K
Discussions	305
Packages	0
Marketplace	0
Topics	0
Wikis	4K
Users	0

Languages: Java

53 repository results

Sort: Best match

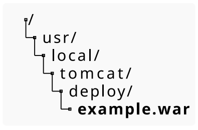
aditya-/Docker_App_Deployment_EC2
This Continuation of Could provisioning of EC2 Docker Host machine, This **Repository** deals with **Java** based Dockerized…
Java MIT license Updated on Jul 15, 2019

oEscal/**java**-image-repo
Image repository web server made in **Java**
Java GPL-3.0 license Updated 2 hours ago

zuoguodang/Convert-iOS-Application-PNG-**images**
Its a **Java repository** which will help you to convert encrypted PNG **images** of iOS application
Java Updated on Jan 22, 2014

sclc/**java**.dip
repository for **java** based digital **image** processing excerices

Fig. 20. GitHub search results for the "Java image repository" keywords, where the repository of the vulnerable application is presented.



```
/
└ usr/
  └ local/
    └ tomcat/
      └ deploy/
        └ example.war
```

Fig. 21. Tree diagram of the vulnerable version path within its container.

From this point, after discovering a web service available on port 8000, the attacker can try to access the corresponding page using a browser. From a simple interaction with it, the necessary details for an aggressor are that it can upload files to that web server, precisely image and zip files. This information is vital because it reports a potential entry point into the targeted system. Another detail on the page that may be significant is the developer's name. Usually, web pages have this information regarding the team that created the application or its provider.

Finally, the attacker may also try to find more specific information regarding the web application's functioning. Usually, this is done by searching for other open endpoints that give access to confidential files or using Open Source Intelligence (OSINT) methods to find more information about the application's provider or source code. In this scenario, we made the code publicly available on a GitHub repository. So, we idealized an attacker that would search on GitHub, for instance, the keywords "Java image repository", a combination of the web server's programming language, discovered using Nmap and the page's name and filtered the search by that language. In this scenario, it would be presented with 53 distinct repositories, at least when we did this experiment.[13] On the first page of the search results, depicted in Fig. 20, GitHub presented a repository that would seem suspicious, as it had the developer's name and all the searched keywords.

By inspecting this repository,[14] the attacker would be able to notice that the application might be vulnerable to a Zip Slip due to the usage of version 1.12 of the ZeroTurnaround ZIP library. Another bit of information that will be important during the Exploitation and Installation phases are the container's name and directory where the compiled application is made accessible to Tomcat and the method used to package the Java classes, which in this case, was within a Web Application Resource (WAR) file. We displayed this information on the tree diagram of Fig. 21, representing the path to the said file.

Furthermore, a knowledgeable attacker might also notice that the Tomcat attribute's *autoDeploy* value was not modified. According to the Tomcat Configuration Reference,[15] that flag indicates if Tomcat shall periodically check for an updated web application. This detail means that if its value is true, it will dynamically deploy any new version. Since, by default, its value is true, if the attacker replaces the web application WAR file with a malicious one, it can change the running application and potentially access the container.

[13] We obtained these search results on 05/10/22.
[14] Vulnerable web application's GitHub repository: https://github.com/oEscal/java-image-repo.
[15] Apache Tomcat 8 Configuration Reference page: https://tomcat.apache.org/tomcat-8.0-doc/config/host.html.

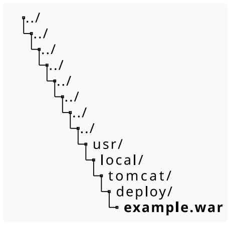


Fig. 22. Tree diagram of the path where the exploit was packaged inside its analogous zip file.

B.2. Weaponization

From the information obtained in the previous phase, the attacker could craft an exploit that would target the Zip Slip vulnerability. In our evaluation scenario, we decided to use the open-source remote code execution tool *Apache Tomcat webshell application,*[16] which provides a WAR with a web application that allows executing shell code using HTTP requests.

Then, with the information of where the original web application WAR file is located, and its name, a possible way of weaponizing this web shell is to include it within a purposely crafted zip file. When the vulnerable web application extracts this WAR file, it will replace the original with this one. As mentioned previously, the Zip Slip vulnerability is exploitable by delivering a particular zip file which, when "unzipped" by the vulnerable application, causes a path traversal. This behavior means that the application will store that zip's file contents within a directory where they were initially not meant to be. This path traversal is achieved by creating directories inside the zip file corresponding to the relative path where the attacker needs its files to be saved. For instance, in this scenario, since the objective is to replace the original web application WAR file with the web shell one, the created directories inside the weaponized zip file are presented in the tree diagram of Fig. 22, where *example.war* corresponds to the web shell. When comparing this with the chart shown earlier in Fig. 21, the last part of the new one is the same. The only difference is the beginning, where the multiple ".." directories inside each other will cause the web application to go from parent to parent until it achieves the root directory. From there, the original path can access the actual web application file, hence the last part of the diagram being the same.

B.3. Delivery

This phase transmits the "weapon" created in the previous step to the targeted system. In this case, from the information gathered in the Reconnaissance, the attacker knows that it can deliver the crafted zip file by uploading it using the application's web page, initially intended for uploading multiple images.

B.4. Exploitation

If the intruder appropriately crafted the said "weapon" and delivered it to the target web page, the application would launch it. When this happens, due to the flag *autoDeploy*'s value being true, Tomcat executes that new file instead of the original web application's, it can be said that the exploitation phase was completed.

B.5. Installation

This phase is characterized by establishing a backdoor within the targeted system. Therefore, a point can be made that an attacker would already have reached this stage using the presented exploit since it is a web shell. However, we must remember that this web shell replaces the original application. Realistically, a sophisticated intruder would probably use this web shell to establish another backdoor within the system, not to raise any alarms from the administrators or the application provider. Then, with that backdoor, it could revert the execution of the web shell to the original application. A simple way would be uploading the application's WAR file through that backdoor, replacing the web shell WAR. To obtain a WAR similar to the original application, the attacker could compile the code from the GitHub repository found in the Reconnaissance phase.

[16] Apache Tomcat web shell application repository: https://github.com/p0dalirius/Tomcat-webshell-application.

B.6. Command and control

C2 is usually associated with the access done by the attacker after the said backdoor was installed on the targeted system. As previously argued, in some sense, this phase would already be reached within the Exploitation and Installation phases. If the attacker accesses the installed web shell, it can execute commands on the targeted system. However, when the intruder installs a backdoor and removes its traces by setting up the original version, the attacker could use that backdoor to access the exploited system. Furthermore, since, in this study, the MTD system does not protect the system, the vulnerable version is always being executed. Therefore, theoretically, the attacker will always be able to access the established backdoor in the future unless the application is updated or its container restarted for some reason. This reason can be as simple as a scale operation that might take place according to the necessities of that application. Another potential explanation would be that the Operator or the application's provider noticed the application was infected, hence patching and updating it.

B.7. Actions on objectives

Finally, the attacker can achieve its goals using the referred backdoor and the C2 channel established in the previous steps. For example, its purpose may be to disrupt the application, which, in this case, can be achieved by removing all the images other users have stored within the repository. Furthermore, this intrusion may be one of many in an attempt to access other system components, where the attacker's goals are moving laterally to access and compromise the Operator's infrastructure. However, the attackers' objectives are out of this manuscript's scope and shall be studied in further works.

Appendix C. Theoretical modeling of the MTD system

Although we empirically analyzed each Intrusion Kill Chain phase in Section 4, we can also make some theoretical assumptions about how some phases could be achieved in the presence of the proposed MTD methodology. However, it is necessary to denote that the success of an attacker intruding on any system will heavily depend on its strategies and not only the mechanisms protecting that system. Therefore, we do not focus on such models, although many works based on techniques such as Game Theory try to model an attacker's behavior. Moreover, in these calculations, we do not consider the impact of time, i.e., the size of the chosen mutation windows. This specificity should be, therefore, studied in future works.

This section is subdivided as follows. We start by calculating the probability of a client communicating with a specific version at any given moment, i.e., if a client mindlessly chooses to contact the application, what is the probability of hitting a particular version? Then, we demonstrate the likelihood of reaching a specific version if it is installed in a particular mutation window, i.e., knowing that the targeted version is installed at a specific mutation window, what is the probability of communicating with it. With these probabilities, we can have a better theoretical look at how the MTD system should behave empirically.

C.1. Probability of hitting a specific version

In this section, we will give a theoretical analysis of the probability of hitting a specific version with the presence of the proposed MTD methodology. Its importance stems from it being related to the success of the Reconnaissance and Delivery phases if an attacker tries to reach the vulnerable version at any given moment. On the Reconnaissance, the intruder has to contact the vulnerable version to obtain the correct information about that version to proceed with the intrusion. On the other hand, on the Delivery, it has to be able to deliver the crafted exploit to the correct version, as the other ones might not possess the said vulnerability. Furthermore, the Delivery's success will be connected to the Exploitation's success, as if "the correct exploit" is delivered to the correct version, it will be executed (unless any other mechanism is protecting the application). If we consider event A as *hitting a specific version*, we can create a theoretical model of the probability $P(A)$ of a client communicating with a particular application version.

For the intruder to communicate with a specific version, two conditions need to hold for the proposed MTD system: that version needs to be installed at that moment, and the load balancer needs to redirect the request to that version from all the executing ones. Therefore, starting with the first affirmation, if we have a pool with n versions to choose from and we need to select s number of simultaneous versions to be executed at a given moment, the number of options we have is given by Eq. (1).

$$\binom{n}{s} \tag{1}$$

And, from those options, the ones corresponding to one of the chosen versions being the targeted one are given by Eq. (2).

$$\binom{n-1}{s-1} \tag{2}$$

With this in mind, the probability of a specific version being executed at any given moment can be expressed by Eq. (3).

$$\frac{\binom{n-1}{s-1}}{\binom{n}{s}} \tag{3}$$

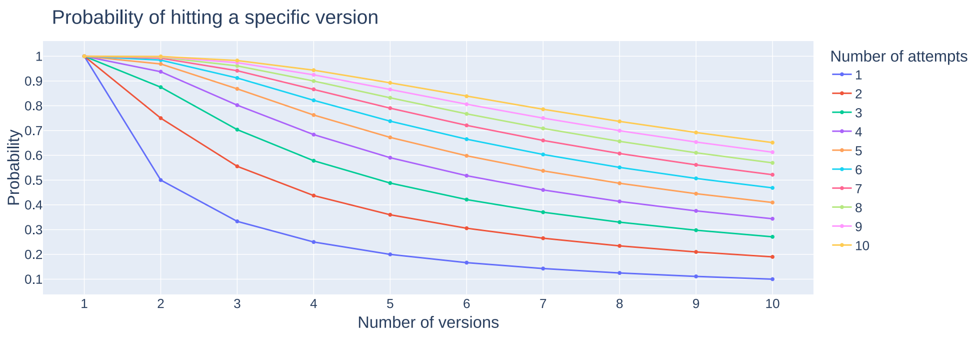


Fig. 23. Probabilities of communication with a specific version for multiple attempts concerning the number of application versions.

However, this probability does not account for the second condition we highlighted previously: for the attacker to communicate with the targeted version, the load balancer must redirect the request to that version from all the executing. If the specific version is installed, the probability of the load balancer directing the request to it can be expressed by Eq. (4).

$$\frac{1}{s} \tag{4}$$

Finally, considering the two outlined probabilities, we can express $P(A)$ with Eq. (5) formula.

$$P(A) = \frac{1}{s} * \frac{\binom{n-1}{s-1}}{\binom{n}{s}} \tag{5}$$

However, we can simplify this formula by expanding each binominal coefficient, as showcased in Eq. (6).

$$\frac{1}{s} * \frac{\binom{n-1}{s-1}}{\binom{n}{s}} = \frac{1}{s} * \frac{\frac{(n-1)!}{(s-1)!*(n-1-s+1)!}}{\frac{n!}{s!*(n-s)!}} = \frac{1}{s} * \frac{(n-1)! * s! * (n-s)!}{(s-1)! * (n-s)! * n!} = \frac{1}{s} * \frac{(n-1)! * s * (s-1)!}{(s-1)! * n * (n-1)!} = \frac{1}{s} * \frac{s}{n} = \frac{1}{n} \tag{6}$$

We then conclude that $P(A)$ can be expressed by Eq. (8) formula. Therefore, the probability of hitting a specific version depends not on the number of concurrent versions but on the number of application versions.

$$P(A) = \frac{1}{n} \tag{7}$$

Now, if an attacker continues to try contacting said version after each failure, the probability $P(A)$ for a specific number of attempts can be expressed by Eq. (8).

$$P(A) = \sum_{t=1}^{t} \frac{1}{n} * (1 - \frac{1}{n})^{t-1} \tag{8}$$

Finally, we can plot this final probability concerning different numbers of application versions and various numbers of attempts. We drew such a graph in Fig. 23. As expected, if we increase the number of application versions, the attack difficulty increases, as the attacker will have more trouble reaching the vulnerable version. Even if it makes multiple followed attempts, it is more difficult to have success as the number of versions increases. Moreover, we need to denote that, since this is an MTD system, windows are maintained for a specific period and only then change. This characteristic impacts that if the targeted version is not installed at one particular trial, the attacker will have to wait until the next window for another chance.

C.2. Probability of hitting a specific version knowing it is installed

Apart from knowing the probability of hitting a specific version at any given moment, it is also essential to calculate the chance of hitting a version if it is installed. Only with it can we guess the added protection per mutation window. Moreover, this probability is related to the Delivery, Exploitation, and C2 phases. In this case, the first two can be considered interconnected. If we think of any application with a vulnerability, if an attacker crafted “the correct exploit” for it, if it successfully delivers the exploit, the Exploitation will be achieved (if no other mechanism is protecting the application). Therefore, with the presence of the MTD system, the probability of hitting a version if it is installed in a given window will give the likelihood of achieving Delivery with success in that window and, therefore, the probability of attaining the Exploitation with success. As for the C2 phase, the attacker needs to reach the backdoor after the exploit is executed and a backdoor is installed. However, it can only achieve this if it can contact the vulnerable version where the backdoor was installed. Considering this information, we can consider event B as *the particular version is already installed*. Having, once again, event A as *hitting a specific version*, we will calculate, in this section, $P(A|B)$.

If we consider an attacker only makes one attempt when the vulnerable version is installed, the probability of it contacting the vulnerable version is given by Eq. (9). The explanation for this equation is that when a client is trying to communication with a

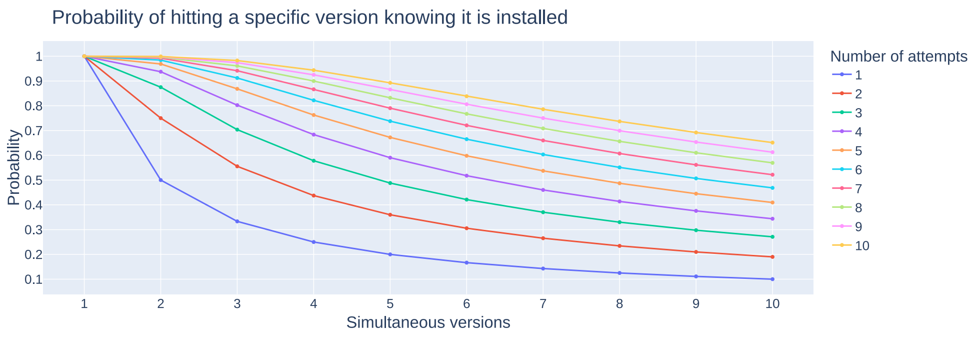


Fig. 24. Probabilities of communication with a specific version for multiple attempts concerning the number of application versions, knowing that version is installed.

specific version the version is being executed, the load balancer will randomly choose an installed version to process the request. Since the number of versions being executed at each moment is equal to the number of concurrent versions (we are not considering the periods between mutation windows), and the request may reach any executing version with the same likelihood, the probability of hitting the targeted version is given by the referred formula.

$$\frac{1}{s} \tag{9}$$

However, suppose an attacker realizes it did not reach the Exploitation or C2 phases (for example, by verifying that the application's behavior is maintained). In that case, it will probably keep sending the same request to the application. Therefore, we must also consider the probability of not hitting the targeted version before the new attempt. This probability is given by Eq. (10).

$$1 - \frac{1}{s} \tag{10}$$

And, therefore, the probability of a new attempt being successful, considering that the previous one was not, is given by Eq. (11).

$$(1 - \frac{1}{s}) * \frac{1}{s} \tag{11}$$

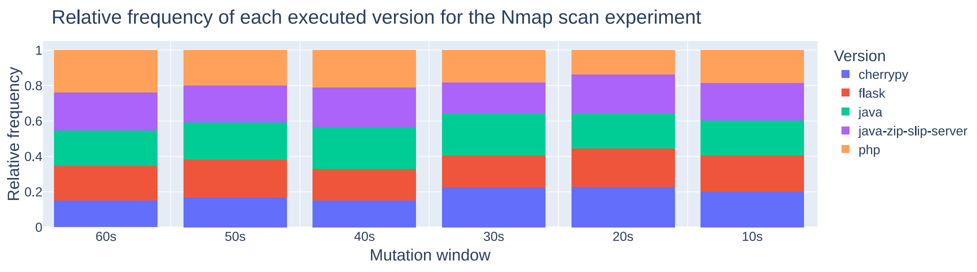
Considering multiple consequent attempts, we can formulate $P(A|B)$ as given by Eq. (12) and simplified as Eq. (13). Therefore, $P(A|B)$ only depends on the number of simultaneous versions.

$$P(A|B) = \frac{1}{s} + (1 - \frac{1}{s}) * \frac{1}{s} + (1 - \frac{1}{s})^2 * \frac{1}{s} + (1 - \frac{1}{s})^3 * \frac{1}{s} + \cdots \tag{12}$$

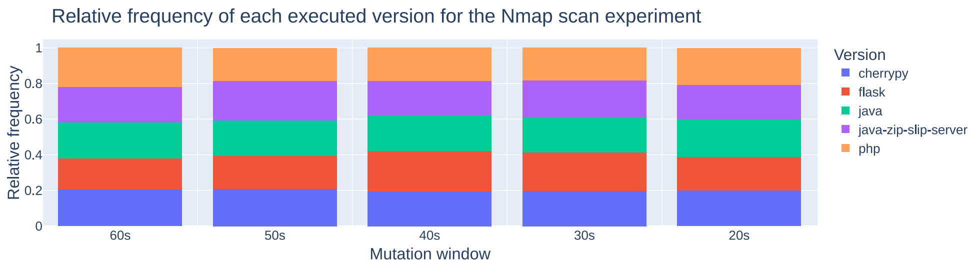
$$P(A|B) = \sum_{t=1}^{t} \frac{1}{s} * (1 - \frac{1}{s})^{t-1} \tag{13}$$

Moreover, comparing Eq. (13) with Eq. (8), we can conclude that both formulas are the same but have distinct variables. While the probability of hitting a specific version depends on the number of application versions, this depends on the number of concurrent versions in a mutation window with the installed vulnerable version. From these observations, we can conclude that both MTD variables, i.e., the number of application versions and simultaneous versions, are crucial for different intrusion moments. The number of application versions will confer protection within the Reconnaissance and Delivery phase if the intruder is mindlessly trying to reach the vulnerable versions or deliver the exploit at any moment. However, if the vulnerable version is installed at a specific mutation window, the number of simultaneous versions will give the application more protection.

We can plot $P(A|B)$ from this formula concerning the number of simultaneous versions for multiple attempt values. Fig. 24 plots this probability for various concurrent versions and numerous attempts. Comparing the probability considering different numbers of attempts, we can conclude that the probability of an attacker achieving a specific version if it is installed increases with the increase in the number of tries. However, we also need to remember that in a real scenario, an attacker will have a limited number of trials for two reasons. First, inherently to the proposed MTD methodology, the vulnerable version will only be installed for a limited time, after which it will be removed and replaced by a new version. The other reason is that if an attacker tries too many times to make the same dubious request, an IDS might detect such behavior as concerning and block it. Therefore, if the vulnerable version is installed, the threat actor will have limited time to deliver the exploit or to contact the installed backdoor, depending on the phase. If this mutation window ends, it must wait for another window with the vulnerable version to deliver the exploit. In the case of the C2 phase, it will have to start from the delivery phase all over again since the backdoor is removed with the removal of the vulnerable version.

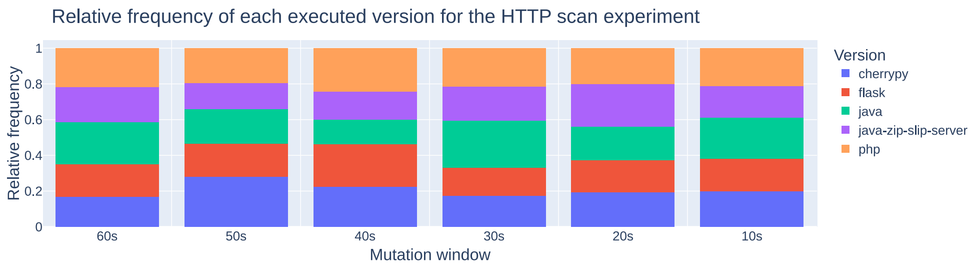


(**a**) For one simultaneous version.

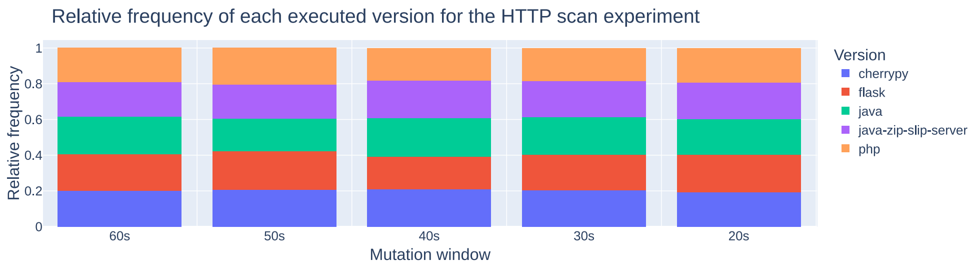


(**b**) For two simultaneous versions.

Fig. 25. Relative frequency of each executed version, obtained throughout the Nmap scan experiment of one hour, for one simultaneous version.



(**a**) For one simultaneous version.



(**b**) For two simultaneous versions.

Fig. 26. Relative frequency of each executed version, obtained throughout the HTTP scan experiment of one hour, for one simultaneous version.

Appendix D. Frequency of executed versions during the reconnaissance phase

During the experiments done in Section 4.3.2 for the Reconnaissance phase, we also obtained the version executed at each moment. We did this by querying the Kubernetes API, using kubectl, at each second, for the Kubernetes Pods installed in the analogous namespace. With this information, we could filter the corresponding version of the Pod. With all the versions installed at each moment, we created the graphs of Figs. 25 and 26, corresponding to the relative frequency of the installed versions throughout the one-hour experiment. The first graph corresponds to the Nmap scan analysis, while the second corresponds to the HTTP scan one.

From these histograms, we can conclude that, as theoretically expected, each created version is instantiated roughly one-fifth of the time. However, as expected from a practical experiment, which is inherently constrained by its duration, there were some mutation window sizes where some versions were instantiated more or less time than others. However, this difference is less noticeable when we use redundancy of two, as demonstrated in the histograms of Figs. 25(b) and 26(b). This characteristic is because more versions are installed within the same interval, diluting the error between the expected values and the obtained results when we have more simultaneous versions.

References

[1] H. Chen, W. Qin, L. Wang, Task partitioning and offloading in IoT cloud-edge collaborative computing framework: a survey, J. Cloud Comput. 11 (1) (2022) 86, http://dx.doi.org/10.1186/s13677-022-00365-8.
[2] S. Li, L.D. Xu, S. Zhao, 5G Internet of Things: A survey, J. Ind. Inf. Integr. 10 (2018) 1–9, http://dx.doi.org/10.1016/j.jii.2018.01.005.
[3] 5G-PPP, 5G Vision - The 5G Infrastructure Public Private Partnership: The Next Generation of Communication Networks and Services, Technical Report, 5G-PPP, 2015.
[4] B. Blanco, J.O. Fajardo, I. Giannoulakis, E. Kafetzakis, S. Peng, J. Pérez-Romero, I. Trajkovska, P.S. Khodashenas, L. Goratti, M. Paolino, E. Sfakianakis, F. Liberal, G. Xilouris, Technology pillars in the architecture of future 5G mobile networks: NFV, MEC and SDN, Comput. Stand. Interfaces 54 (2017) 216–228, http://dx.doi.org/10.1016/j.csi.2016.12.007.
[5] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, R. Boutaba, Network function virtualization: State-of-the-art and research challenges, IEEE Commun. Surv. Tutor. 18 (1) (2016) 236–262, http://dx.doi.org/10.1109/COMST.2015.2477041.
[6] ETSI, Network Functions Virtualisation, Technical Report, 2012.
[7] ETSI, ETSI GS NFV 002 V1.2.1, Technical Report, 2014.
[8] ETSI, ETSI GS NFV 001 V1.2.1, Technical Report, 2017.
[9] A. Filali, A. Abouaomar, S. Cherkaoui, A. Kobbane, M. Guizani, Multi-access edge computing: A survey, IEEE Access 8 (2020) 197017–197046, http://dx.doi.org/10.1109/ACCESS.2020.3034136.
[10] P. Cruz, N. Achir, A.C. Viana, On the edge of the deployment: A survey on multi-access edge computing, ACM Comput. Surv. 55 (5) (2023) 1–34, http://dx.doi.org/10.1145/3529758.
[11] ENISA, ENISA Threat Landscape for 5G Networks, Technical Report, ENISA, 2020, http://dx.doi.org/10.2824/802229.
[12] M. Pattaranantakul, R. He, Q. Song, Z. Zhang, A. Meddahi, NFV security survey: From use case driven threat analysis to state-of-the-art countermeasures, IEEE Commun. Surv. Tutor. 20 (4) (2018) 3330–3368, http://dx.doi.org/10.1109/COMST.2018.2859449.
[13] P. Ranaweera, A.D. Jurcut, M. Liyanage, Survey on multi-access edge computing security and privacy, IEEE Commun. Surv. Tutor. 23 (2) (2021) 1078–1124, http://dx.doi.org/10.1109/COMST.2021.3062546.
[14] ETSI, ETSI GR NFV-SEC 003 V1.2.1, Technical Report, 2016.
[15] ETSI, MEC Security: Status of Standards Support and Future Evolutions, Technical Report, 2022.
[16] I. Farris, T. Taleb, Y. Khettab, J. Song, A survey on emerging SDN and NFV security mechanisms for IoT systems, IEEE Commun. Surv. Tutor. 21 (1) (2019) 812–837, http://dx.doi.org/10.1109/COMST.2018.2862350.
[17] W.-S. Choi, S.-Y. Lee, S.-G. Choi, Implementation and design of a zero-day intrusion detection and response system for responding to network security blind spots, in: K.-H. Yeh (Ed.), Mob. Inf. Syst. 2022 (2022) 1–13, http://dx.doi.org/10.1155/2022/6743070.
[18] H. Hindy, R. Atkinson, C. Tachtatzis, J.-N. Colin, E. Bayne, X. Bellekens, Utilising deep learning techniques for effective zero-day attack detection, Electronics 9 (10) (2020) 1684, http://dx.doi.org/10.3390/electronics9101684.
[19] W. Attaoui, E. Sabir, H. Elbiaze, M. Guizani, VNF and CNF placement in 5G: Recent advances and future trends, IEEE Trans. Netw. Serv. Manag. (2023) 1, http://dx.doi.org/10.1109/TNSM.2023.3264005.
[20] P. Escaleira, M. Mota, D. Gomes, J.P. Barraca, R.L. Aguiar, Multi-access edge computing as a service, in: 2022 18th International Conference on Network and Service Management, CNSM, IEEE, 2022, pp. 177–183, http://dx.doi.org/10.23919/CNSM55787.2022.9964650.
[21] W. Wang, S. Yongchareon, A survey on security as a service, in: Web Information Systems Engineering – WISE 2017, Springer International Publishing, 2017, pp. 303–310, http://dx.doi.org/10.1007/978-3-319-68786-5_24.
[22] ETSI, ETSI GR NFV-MAN 001 V1.2.1, Technical Report, 2021.
[23] ETSI, ETSI GS MEC 003 V3.1.1, Technical Report, 2022.
[24] ETSI, ETSI GS NFV 006 V4.4.1, Technical Report, 2022.
[25] ETSI, ETSI GS NFV-IFA 009 V1.1.1, Technical Report, 2016.
[26] V. Sciancalepore, F. Giust, K. Samdanis, Z. Yousaf, A double-tier MEC-NFV architecture: Design and optimisation, in: 2016 IEEE Conference on Standards for Communications and Networking, CSCN, IEEE, 2016, pp. 1–6, http://dx.doi.org/10.1109/CSCN.2016.7785157.
[27] NITRD, National Cyber Leap Year Summit 2009 Participants' Ideas Report, Technical Report, NITRD, 2009.
[28] G.-l. Cai, B.-s. Wang, W. Hu, T.-z. Wang, Moving target defense: state of the art and characteristics, Front. Inf. Technol. Electron. Eng. 17 (11) (2016) 1122–1153, http://dx.doi.org/10.1631/FITEE.1601321.
[29] J.-H. Cho, D.P. Sharma, H. Alavizadeh, S. Yoon, N. Ben-Asher, T.J. Moore, D.S. Kim, H. Lim, F.F. Nelson, Toward proactive, adaptive defense: A survey on moving target defense, IEEE Commun. Surv. Tutor. 22 (1) (2020) 709–745, http://dx.doi.org/10.1109/COMST.2019.2963791.
[30] V.A. Cunha, D. Corujo, J.P. Barraca, R.L. Aguiar, TOTP Moving Target Defense for sensitive network services, Pervasive Mob. Comput. 74 (2021) 101412, http://dx.doi.org/10.1016/j.pmcj.2021.101412.
[31] K. Sattar, K. Salah, M. Sqalli, R. Rafiq, M. Rizwan, A delay-based countermeasure against the discovery of default rules in firewalls, Arab. J. Sci. Eng. 42 (2) (2017) 833–844, http://dx.doi.org/10.1007/s13369-016-2359-0.
[32] P. Sharma, L. Chaufournier, P. Shenoy, Y.C. Tay, Containers and virtual machines at scale, in: Proceedings of the 17th International Middleware Conference, ACM, New York, NY, USA, 2016, pp. 1–13, http://dx.doi.org/10.1145/2988336.2988337.
[33] X. Merino Aguilera, C. Otero, M. Ridley, D. Elliott, Managed containers: A framework for resilient containerized mission critical systems, in: 2018 IEEE 11th International Conference on Cloud Computing, CLOUD, IEEE, 2018, pp. 946–949, http://dx.doi.org/10.1109/CLOUD.2018.00142.
[34] M. Azab, B. Mokhtar, A.S. Abed, M. Eltoweissy, Toward smart moving target defense for linux container resiliency, in: 2016 IEEE 41st Conference on Local Computer Networks, LCN, IEEE, 2016, pp. 619–622, http://dx.doi.org/10.1109/LCN.2016.106.
[35] H. Okhravi, A. Comella, E. Robinson, J. Haines, Creating a cyber moving target for critical infrastructure applications using platform diversity, Int. J. Crit. Infrastruct. Prot. 5 (1) (2012) 30–39, http://dx.doi.org/10.1016/j.ijcip.2012.01.002.
[36] N. Ahmed, B. Bhargava, Mayflies: A moving target defense framework for distributed systems, in: MTD 2016 - Proceedings of the 2016 ACM Workshop on Moving Target Defense, Co-Located with CCS 2016, ACM, New York, NY, USA, 2016, pp. 59–64, http://dx.doi.org/10.1145/2995272.2995283.
[37] M. Villarreal-Vasquez, B. Bhargava, P. Angin, N. Ahmed, D. Goodwin, K. Brin, J. Kobes, An MTD-based self-adaptive resilience approach for cloud systems, in: 2017 IEEE 10th International Conference on Cloud Computing, CLOUD, IEEE, 2017, pp. 723–726, http://dx.doi.org/10.1109/CLOUD.2017.101.
[38] M. Thompson, N. Evans, V. Kisekka, Multiple OS rotational environment an implemented Moving Target Defense, in: 2014 7th International Symposium on Resilient Control Systems, ISRCS, IEEE, 2014, pp. 1–6, http://dx.doi.org/10.1109/ISRCS.2014.6900086.
[39] M. Thompson, M. Mendolla, M. Muggler, M. Ike, Dynamic application rotation environment for moving target defense, in: 2016 Resilience Week, RWS, IEEE, 2016, pp. 17–26, http://dx.doi.org/10.1109/RWEEK.2016.7573301.
[40] ETSI, ETSI GR NFV-IFA 007 V4.2.1, Technical Report, 2021.
[41] E.M. Hutchins, M.J. Cloppert, R.M. Amin, Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains, Technical Report, 2011.