Lightweight On-Demand Honeypot Deployment for Cyber Deception

Jaime C. Acosta[1](✉), Anjon Basak[2], Christopher Kiekintveld[2], and Charles Kamhoua[1]

[1] DEVCOM Army Research Laboratory, Adelphi, USA
jaime.c.acosta.civ@army.mil

[2] Department of Computer Science, University of Texas at El Paso, El Paso, USA
cdkiekintveld@utep.edu

Abstract. Honeypots that are capable of deceiving attackers are an effective tool because they not only help protect networks and devices, but also because they collect information that can lead to the understanding of an attacker's strategy and intent. Several trade-offs must be considered when employing honeypots. Systems and services in a honeypot must be relevant and attractive to an adversary and the computing and manpower costs must fit within the function and budget constraints of the system.

It is infeasible to instigate a single, static configuration to accommodate every type of system or target every possible adversary. The work we describe in this paper demonstrates a novel approach, introducing new capabilities to the Cyber Deception Experimentation System (CDES) to realize selective and on-demand honeypot instantiation. This allows honeypot resources to be introduced dynamically in response to detected adversarial actions. These honeypots consist of kernel namespaces and virtual machines that are invoked from an "at-rest" state. We provide a case study and analyze the performance of CDES when placed inline on a network. We also use CDES to start and subsequently redirect traffic to different honeynets dynamically. We show that these mechanisms can be used to swap with no noticeable delay. Additionally, we show that Nmap host-specific scans can be thwarted *during a real scan*, so that probes are sent to a honey node instead of to the legitimate node.

Keywords: Cybersecurity · Network security · Dynamic honeypots · Experimentation · Testbed

1 Introduction

Honeypots for the most part are static and do not change even as unusual or adversarial behavior is detected. The research in this field is growing rapidly as novel technologies allow for more adaptability. These honeypots vary in scope; some focus on breadth while others focus on depth. OWASP Python-Honeypot

[11], KFSensor [6] and many others ([9,14] contains a substantial list) are capable of mimicking several nodes and services simultaneously, using a software backend. Others, such as HADES [7], provide high-fidelity, full-system mirroring.

Still lacking are comprehensive, easily configurable and deployable honeynet infrastructures that are capable of running on the types of small-scale devices that are seeing broad and expanded usage in the commercial and military sectors. This includes the Internet of Things (IoT), Internet of Battlefield Things (IoBT), vehicular systems, and many more. We foresee these devices in the future each hosting and deploying a minimal, yet carefully selected set of honeynets when malicious behavior is suspected. Honeynet inclusion and deployment must be flexible, allowing the use of small-scale, low-to-medium fidelity, as well as large-scale (for which traffic may have to be redirect off-machine) real-system mirroring. We also foresee the use of small-scale, possibly battery-powered, special purpose devices placed on networks, between existing nodes, to monitor and automatically deploy these defense mechanisms on-the-fly; all leveraging technologies such as kernel network namespaces, container technologies, virtualization, and software defined networking. Novel research in computational decision theory, game theory, and machine learning will be used to coordinate and optimize the deployment of specific resources based on observations of the network and the costs and constraints on implementing and deploying specific network modifications.

Before this vision can be used in real operating environments, a substantial amount of analysis studies must demonstrate the feasibility of these approaches as well as limitations and expectations in live networks. Real data is also necessary to estimate the costs and effectiveness of different strategies for use in decision-making modules. Towards this goal we provide the following contributions in this paper:

- A novel implementation and source code for the cybersecurity deception experimentation system (CDES) that uses Open vSwitch for traffic redirection. This novel implementation is scalable and applicable on real networks, unlike its predecessor.
- A case study used to analyze performance in terms of packet round-trip time delay when using CDES inline on a network.
- Empirical evidence for a realistic use case showing that CDES has the ability to dynamically thwart network probes using a standard network scanning tool.

The rest of the paper includes a description of relevant work in this domain, followed by a description of the improvements made to CDES. Next, we define our case study, which focuses on a simple scenario in which traffic is dynamically redirected to honeynets. Finally, we report on the experimental results and discuss directions for future work.

2 Related Work

There are several free and open-source honeypot projects available to the public [9] that mostly provide different capabilities. For example, the growing trend of Web Application Attacks has given rise to associated honeypots. The django-admin-honeypot [3] hosts false login pages to note any malicious attempts, Nodepot [13] is a NodeJS honeypot, and StrutsHoneypot [12] specifically targets attackers looking to exploit the Apache Struts service. Others aim to attract adversaries looking to target physical devices; for example, ADBHoney [4] emulates an Android running an Android Debug Bridge, AMT Honeypot [16] reflects a vulnerable Intel Firmware. Conpot [5] and GasPot [17] look like industrial control systems, which are common in critical infrastructure networks.

Others are more broad in the capabilities they provide. Honeyd [10] was released in 2003 and with much acclaim. Many people have developed additions to the original code base and extended the functionality, which includes being able to mimic various services and nodes. Released more recently, KFSensor [6] is a low-to-medium fidelity honeypot for Windows that provides multiple services running on multiple ports and even on various IP Addresses. SIREN works similarly for Linux systems, including ARM systems, and is available as open source.

HADES [7], developed by Sandia National Laboratories, is a large-scale honeynet platform capable of mimiking large networks and systems, including the ability to mirror entire networks and switchover on-the-fly. When an attacker is detected in the operational network, it is migrated into a deception network where the configurations and monitoring capabilities can be changed dynamically.

Still lacking are honeypot systems that are lightweight, able to run on small-scale systems, and still able to provide multiple levels of fidelity. Additionally, such a system need to be extensible, open source, and easy to configure and deploy. Performance analysis is also critical to real deployment and largely lacking in the literature. This is especially important when incorporating decision algorithms to strategically (using RL or Game theoretic algorithms) deceive attackers [1,2] with minimal latency and system load, especially when deployed on constrained systems. We believe that on-demand instantiation is a solution that will allow intelligent, adaptive use of these systems suitable to resource-constrained environments.

3 Implementation

The Cybersecurity Deception Experimentation System (CDES) [1] is a standalone, emulation-based platform that is aimed at running small-to medium scale network scenarios. It is built using a modular architecture to encourage adaptation and extension, and it is open source. CDES runs in parallel to the Common Open Research Emulator, which provides many of the fundamental emulation functions including network creation and execution mechanics as well as a graphical interface. CDES is based on a three-stage pipeline that comprises the Monitor, Trigger, and Swapper components. These are shown in Fig. 1.

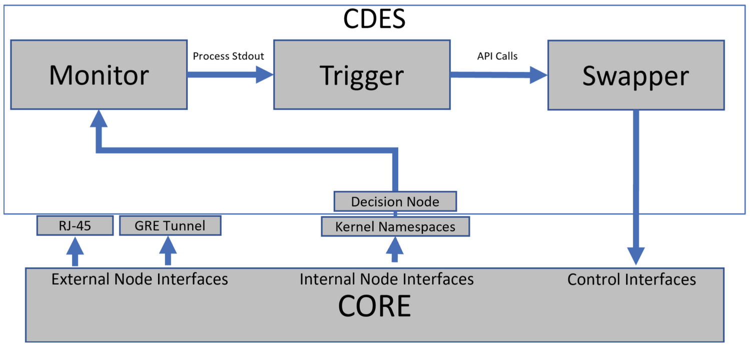

Fig. 1. CDES and the interfaces with CORE

The Monitor observes the output of a particular system process; for example, previous versions of CDES have used the Suricata Intrusion Detection System along with other tools to observe the network. This output is passed to the Trigger, which contains logic (as editable Python code) to determine how to consolidate and process the inputs. The Swapper provides an API that allows a user to execute connection redirection. This component communicates with the CORE backend through various interfaces to achieve this behavior. More information on the internals of CDES along with several samples are available in [1].

The original version of CDES was developed for testing primarily within emulated scenarios, where all nodes reside on a single machine. We have made several modifications to CDES in order to make it more usable on a real network, and to improve the general performance and usability of the system. These changes are backward compatible with the original version, so previous scenarios will work with CORE version 6 and below. Scenarios developed with the CDES updates have been tested on CORE version 6.2.0–7.4 (the latest version as of the writing of this paper). We describe these enhancements below.

3.1 Including Real Networks

CDES previously relied on three different node types to work, two of which we modified. The *decision node* is where the three-stage pipeline is executed, the *conditional nodes* are connected to the decision node, but only if enabled by the Swapper. The third node type, the *conditional connection gateway* was not modified.

Originally, the Swapper would enable or inhibit communication with a node by running an interface command (specifically ifconfig or ip) on the conditional

Table 1. High-level CDES improvements and upgrades

Source	Type	Prior state	Modification
Swapper	Connection swap	Network commands on conditional nodes to enable or disable connections	Connections enabled or disabled with ovs-ofctl commands
	Connection control	Physical interface-based swapping	Any layer-based swapping supported by flow tables
	Performance	Connections are activated individually	Message-based interface for simultaneous activation or deactivation
Trigger	Ad-hoc honey VM instantiation	Written manually in trigger code	Provided by API
Dec. node	Decision node implementation	Layer-2 switch node services	Layer-3 node running Open vSwitch

node. This meant that the conditional node had to exist within the emulation scenario; in other words, this could not be an external node, such as a physical node or a virtual machine. To alleviate this issue, we developed a subclass for the Swapper that instead is based on Open vSwitch that controls the direction of the traffic directly on the decision node. The conditional node can now be a device external to CORE (e.g., incorporated into a scenario using the RJ-45 adapter) (Table 1).

Therefore, the decision nodes now execute ovs processes. Every interface on these nodes is added to a default ovs bridge and flow entries are added and removed depending on which connections are designated as enabled or disabled, through calls to the Swapper. To improve performance, the Swapper now executes all system call operations in a single function call, allowing a user to specify active and inactive connections within a single function parameter. Feedback from users of the system indicated that the previous way of specifying conditional connection nodes (using an integer value) was difficult, because the ordering was not always consistent. To remediate this, we created a messaging system, where different message types use different formats to indicate actions and nodes. This is also how we maintained backward compatibility with previously developed scenarios. Using Open vSwitch also comes with many additional benefits, including the ability to specify swapping at various layers in the network stack, including Ethernet-based, IP Address based, etc.

3.2 Ad-Hoc Virtual Machine Instantiation

To faciliate the task of dynamically instantiating external virtual nodes, we added functions to the backend Trigger API for pausing, saving, resuming, and starting virtual machines. A user can specify whether the virtualization system is local (running VirtualBox on the host) or remote (in which case an SSH session is instantiated and run as a specified user). Instead of VBoxManage, a user may also specify to use pyvbox; which would require that the VirtualBox SDK be installed on the host running VirtualBox.

3.3 Decision Node Configuration

In addition to incorporating Open vSwitch processes on the decision node, the base type had to change. Recent versions of CORE have evolved and added new features; at the same time, some interfaces have also changed. Layer 2 switch nodes are no longer able to be assigned services. To adapt for these changes, the decision node is now a layer 3 node that behaves like a switch (using ovs). A benefit of this change is that now processes can be spawned on decision nodes during a running emulation. This also better fits the logical design and usage of CORE. However, adding and using this type of node in the CORE GUI may cause some confusion, since by default the node is treated as a router node instead of a switch node, e.g., IP addresses are generated and then auto-assigned when connected to another node.

One of the fundamental objectives of CDES is to leverage CORE without requiring any modifications to CORE. We have kept this model, but we also provide users with the option to alleviate some of the confusion presented by the issue mentioned in the previous paragraph. We modified 3 source files (linkcfg.tcl, ip4.tcp and ip6.tcl) that will provide a switch behavior for decision nodes when used in the graphical interface. These are optional and the system will still run correctly without including these changes.

These modifications also made it possible to incorporate CDES into CORE scenarios with mobile wireless nodes. In the first case, we tested this functionality by designating a wireless node as a decision node. The Swapper can switch between the different wireless networks, which in the real world are synonymous with different wireless network interfaces connected to different SSIDs. This also works with what we call base station nodes (with one wired connection and at least one wireless connection).

The code base now include several new samples, including the changes described above as well as several which use SDN components (Open vSwitch, Ryu, etc.) in CORE.

4 Case Study

We envision CDES being used inline on a network, on a limited resource device that will have minimal impact on network throughput. To demonstrate this we developed two separate experimentation setups.

In the first, CDES runs on a recent laptop with considerable memory and a decent processor. In this setup, which we call *In-VM*, CORE is used within a virtual machine. This facilitates deployment, modification, and maintenance.

This setup makes it easy to install CORE, configure the trigger rules, and apply updates to CDES and then transfer the latest version as an importable virtual machine. Other VMs are easily added and their network configurations are preserved. No changes are required to execute the setup across different machines, even those with different host operating systems. For this setup, we used a Laptop with 64 GB RAM and the Intel Xeon E3-1505M v6 3.0 GHz processor. The host operating system is 64-bit Windows 10 and CORE is installed on a virtual machine running Ubuntu 20 64-bit LTS.

The second setup, which we call *Native*, runs CDES installed on an older, less capable laptop. Virtualization is used only to host the honeynet VMs. This laptop had 16 GB of RAM and a 2.6 GHz Intel Core i7-4720HQ. Ubuntu 20 64-bit LTS was the host operating system.

In both cases, the CDES laptop was placed on an isolated network between two communicating nodes (see Fig. 2). On the left side is the scanning machine. This is where traffic originates and where we measure round-trip time for packets. This laptop was the same model as the host used for the In-VM set up; all specs were the same, except that is was running Ubuntu 20 64-bit LTS instead of Windows.

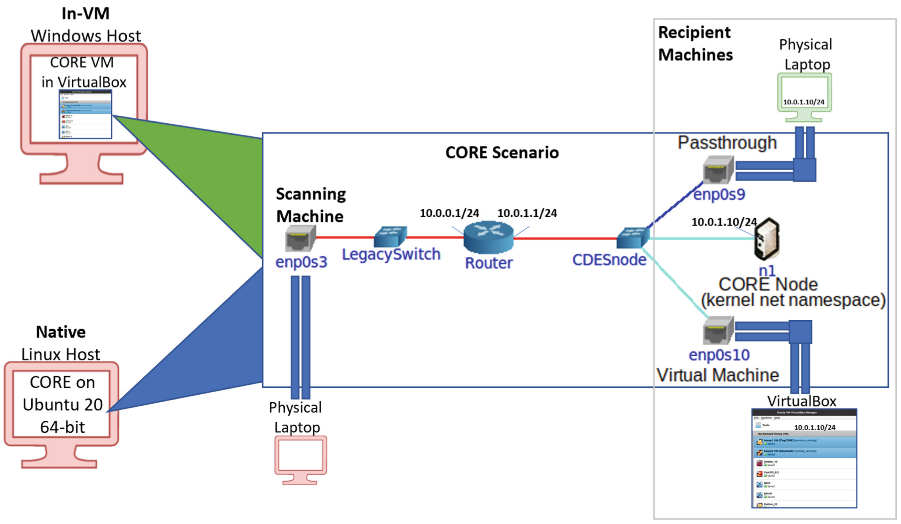

Fig. 2. Case study scenario, developed in CORE

The CDES laptop is connected to the network using two physical interfaces. The left interface (through enp0s3 in Fig. 2) is connected to the scanning machine using a USB 3.0, 1 Gbps dongle, and on the right is the native, hardwired network interface card included with the laptop. It is worth noting that there were no differences in results when we reversed the ordering of these interfaces. When using the In-VM setup, VirtualBox was used to run the CDES node with three virtual network interfaces. The first was bound to the Ethernet dongle in bridged mode. The second was an internal network shared with two *honey* virtual machines, a minimal Linux TinyCORE VM, allocated with 2 processors and 128 MB RAM and an Ubuntu 20 64-bit LTS VM, allocated with 2 processors and 2 GB of RAM. Finally, the third adapter is bound to the laptop's internal 1 Gbps network interface card, which is used as the passthrough.

This right-most interface on the CDES laptop connects to a 1 Gbps Ethernet switch, which is also connected to the passthrough machine. This machine the same laptop model as that used for the In-VM CDES node, running Ubuntu 20 64-bit LTS so that it mirrors one of the honey VMs.

The honey VMs (connected through the virtual interface enp0s10) and the passthrough machine (connected through enp0s9) are all configured to use the same IP Address and the same MAC address. This is so the honey virtual machines closely mimic the passthrough node from a networking perspective. It also eliminates the need for additional address resolution traffic when redirection occurs, which would otherwise cause intermittent delays.

The primary objectives of this case study are as follows:

1. Determine the impact on total throughput that a CDES node will introduce on the network
2. Understand the delays that are introduced with the additional processing required for CDES to work; specifically when switching between a passthrough network, a local kernel network namespace (also known as a native CORE node) and two types of virtual machines connected to a CORE scenario using a CORE RJ-45 node.
3. Demonstrate the capabilities of CDES for thwarting adversarial behavior, specifically network service probes.

We used delays in ICMP echos and requests to measure performance impacts on the network. In each setup, the scanning machine sent requests at a rate of 10 per second. For network probes, the scanning machine used Nmap with different timing templates. This is explained further in the following sections.

5 System-in-the-Middle Overhead Analysis

We measure overhead first with static connections (no CDES or swapping) and then with dynamic connections (using CDES with different instantiation and swapping configurations) in order to characterize delays associated with the different setups.

5.1 Static Connections

To establish a baseline with respect to network delays and the additional load introduced by the CDES node and accompanying software, we measured the ICMP echo-response delays without any intermediate nodes during a 1000-s test. We temporarily removed the CDES node and connected the scanning machine to

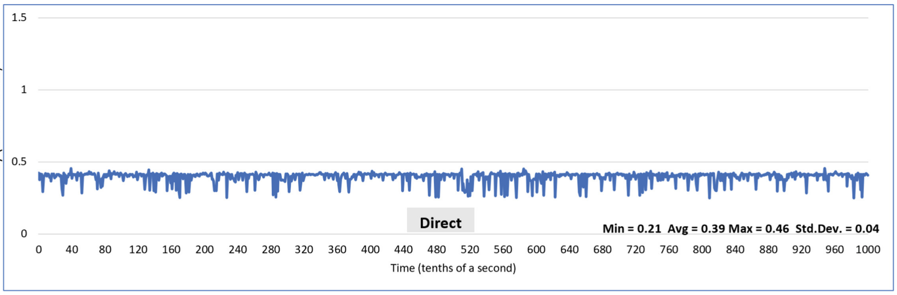

Fig. 3. Ping delays when scanning machine is directly connected to passthrough machine

the passthrough machine through the network switch. The delays are very stable, as expected in such as small and simple networking configuration. Figure 3 shows the delays with statistics exhibited during the duration of the test.

Next, we measured the delays associated with adding an intermediate node. In the case of the native setup, these delays encompass processing done by the hardware and the kernel. In the case of the In-VM setup, results are shown in Fig. 4.

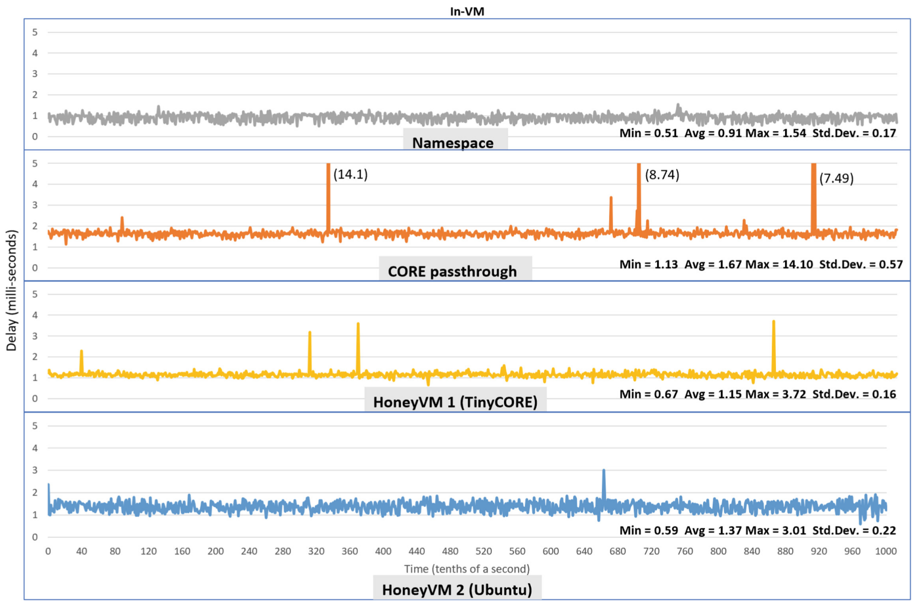

Fig. 4. Ping delays when scanning machine traffic flows through within-VM CORE

Additional delays are caused by the processing required for the VirtualBox bridged adapter and the virtual machine hosting CORE; no other user processes were instantiated. In general, the namespace node responds the quickest. This is because the incoming packets are not redirected to external systems; packets arrive and are processed by a specific kernel network namespace. The CORE passthrough, which is the case when packets are redirected through the second physical network interface is the slowest. In this case, packets are pushed through the network switch to the recipient laptop and processed by that second physical device and its OS kernel.

Additionally, there are several sporadic delays throughout, sometimes rising to 0.14 s with the passthrough. To test whether this behavior was consistent, we repeated the experiment with different combinations of hardware, Operating System (Linux), and virtualization platform (VMWare). The same behavior persisted. This is likely due to the internal switching mechanisms used by the virtualization software. However, it is also worth noting that even though these additional and sporadic delays exist, the delays are still relatively short, on average within 1 ms–2 ms compared to 0.5 ms–1.01 ms for the native setup. The standard deviations are all below 0.6 ms during the 100-s runs.

The results associated with the native setup are shown in Fig. 5. Delays when using the CORE node as the recipient (kernel network namespace) are on average only 0.12 ms higher. Adding the additional node, as seen with CORE passthrough, adds 0.62 ms to the average delay time. The min and max delays in this setup are all within 1.02 ms, demonstrating connection stability even with the additional processing.

When comparing the two setups, it is clear that In-VM introduces more delay across all tests, but they are similar. Depending on the needs of the administrator (e.g., fidelity versus ease of use) both setups are viable, especially in the case where a namespace is used.

5.2 Dynamic Connections Using CDES

We used CDES to swap between the different possible recipient nodes (passthrough, namespace, honey1 VM and honey2 VM). Both of the honey VMs were connected to the CORE scenario using the RJ-45 node and only one was active at any given time. For this reason, even though all of these recipient nodes were configured to use the same addresses there were no conflicts. We tested three different sets of configurations. In the *no_inst* configuration, the honey VM is instantiated before the scenario starts and it is never stopped. The *pause_resume* configuration starts with all VMs in a suspended state, with memory allocated beforehand, but machines are activated only when resumed. The machines are again suspended when they are no longer in use. It is important to note that the ordering of execution has an important role. A VM is instantiated on a thread, and the CDES swapping logic waits until this operation is complete until swapping to use the associated connection. When a VM is toggled for suspension, the swapping does not occur until the machine the operation is complete. This

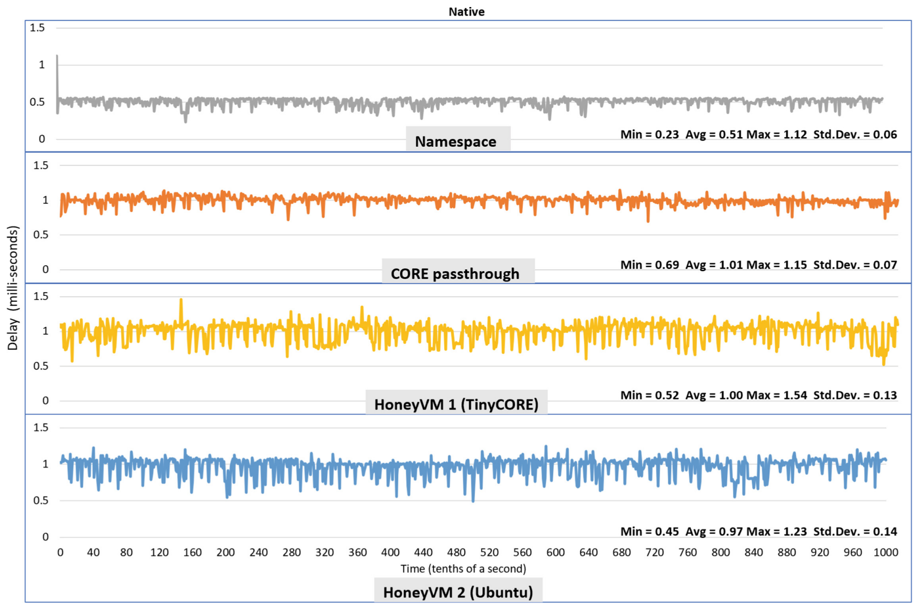

Fig. 5. Ping delays when scanning machine traffic flows through native-installed CORE

model we used in the case study favors possible delays instead of possible duplicate packets. This eliminates the case where, if two connections were to be active at the same time, both recipient nodes could respond to the ICMP echo request. The scanning machine would see these as duplicate packets.

Lastly, in the save_state case, the VMs are offloaded, but their state is saved to disk and restored when they are instantiated. We used the same mechanics and logical execution as the *pause_resume* configuration. In the In-VM setup, the VMs are halted and started as needed using an ssh connection to the host machine and subsequently running the VBoxManage binary to control the VM. In the native setup, we simple called the VBoxManage binary directly.

To measure the delays associated with the various CDES redirections, the redirection would occur every 30 s for a total of 7 times (210 s), and in the following order: passthrough, namespace, passthrough, honey1 VM, passthrough, honey2 VM, passthrough. This behavior was implemented in the CDES Trigger script using the CORE graphical interface, and required only 12 additional lines of Python code.

We started with the no_inst configuration (shown in Fig. 6 and we noticed immediately that in both setups, there was no noticeable delay when the swaps occurred and zero packets were lost. However, as expected from previous results, the delays associated with the namespace were noticeably lower (shown in the upper graphs). We added an intentional delay of 400 ms to the link connecting the CDES node to the namespace node. This decreased the standard deviations from 0.19 ms to 0.09 ms in the Native setup and 0.63 ms to 0.51 ms in the In-VM setup, as shown in the lower graphs.

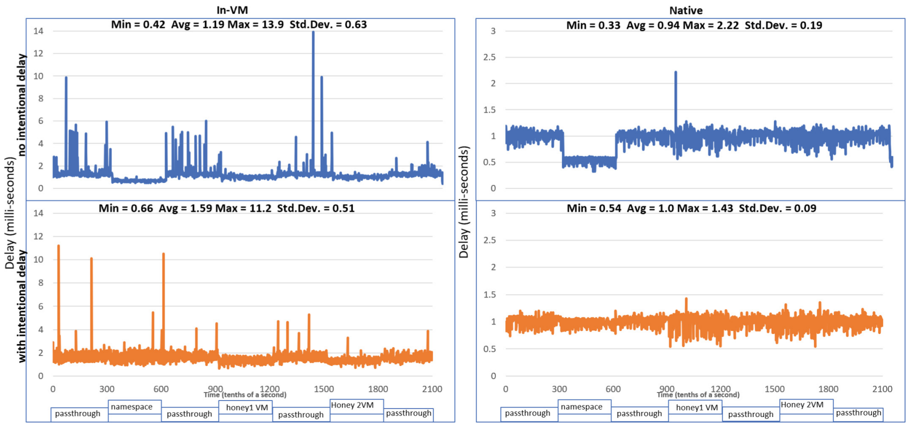

Fig. 6. Ping delays using CDES without (upper graphs) and with (lower graphs) intentional delay in both the In-VM (left graphs) and Native configurations (right graphs).

Also coinciding with the results presented in the previous subsection, the In-VM setup exhibits more sporadic behavior than the Native. However, these results are in the range of milliseconds. In more complex and congested networks, the sporadic behavior may be harder to notice.

The delays associated with the pause_resume and save_state configurations using intentional delay are shown in Fig. 7. In both configurations, there were no dropped packets. In the case of the pause_resume, there is very little noticeable difference in delays when compared to the no_inst configuration. The minimum increase was below 0.08 ms, the average increase was 0.4 ms and the max increase was within 0.15 ms. The standard deviation changed only by 0.01 ms. The resume VM operation completed within 1 s in both the In-VM and Native case. Since the switchover doesn't occur until the VMs are instantiated, this reduces (and in this case virtually eliminates) additional CDES-incurred delay.

The save_state configuration did not perform as well, especifically when instantiating the Honey2 VM (Ubuntu). Originally, we thought that these results would closely mimic those in the pause_resume configuration due to ordering of the the swapping logic, but we noticed that this was not the case; there were additional delays, up to slightly over 2 full seconds. Closer observation revealed that when a VM is started from a saved state, there is a response gap between the time that the VM is fully functional and interactive to when its network devices become active. In the case of the Ubuntu VM this was 2 s. However, it seems that the packets are queued by the virtualization software, since eventually there were responses and no packets were lost. This was the case in both setups, in which the total varied by at most 11 ms. Memory and CPU did not seem to

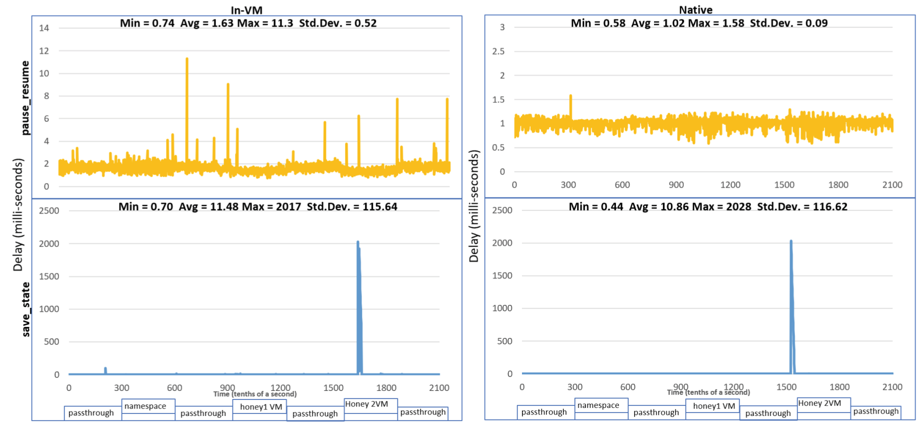

Fig. 7. Ping delays using pause_resume (upper graphs) and save_state (lower graphs) configurations in both the In-VM (left graphs) and Native configurations (right graphs).

play a role in this behavior, as they were both below maximum utilization. We discuss these results in the next section.

6 CDES System Utilization

We analyzed the performance in terms of both CPU and memory utilization in the native setup for the three configurations. CPU usage was recorded using the mpstat tool, which is part of the sysstat package. The load of the system was recorded once per second. The results are shown in Fig. 8.

In both the no_inst and pause_resume configurations, the memory is constant at 25%, which is roughly 4 GB of memory, throughout the scenario. Utilization varies much more during save_state, when the VMs are instantiated and shutdown dynamically. There is only a small increase of roughly 2–3% when the TinyCORE VM (honey1 VM in the chart) is instantiated, but the increase is roughly 10% when instantiating the Ubuntu VM (or honey2 VM). The memory usage is not instantaneous. As we show in the graphs, this is gradual and occurs over 3–5 s.

CPU performance is show in the graphs on the right side of the figure, including statistics for the executions. Since all of the configurations start at 100% utilization (due to CDES instantiation) the maximum statistic is based on the data after 1 s. CPU utilization is much more stable in the configurations where VMs are not instantiated from a saved state, as observed in the average being 2.8%–2.98% and standard deviation between 6.81% and 6.79%. Still, even when using the less capable laptop, the usage remained below 25% throughout.

In summary, the usage of CDES should depend on the availability of resources. When used on a device with limited memory, and when the inherent delays (presented in Sect. 5, are acceptable, the save_state configuration is

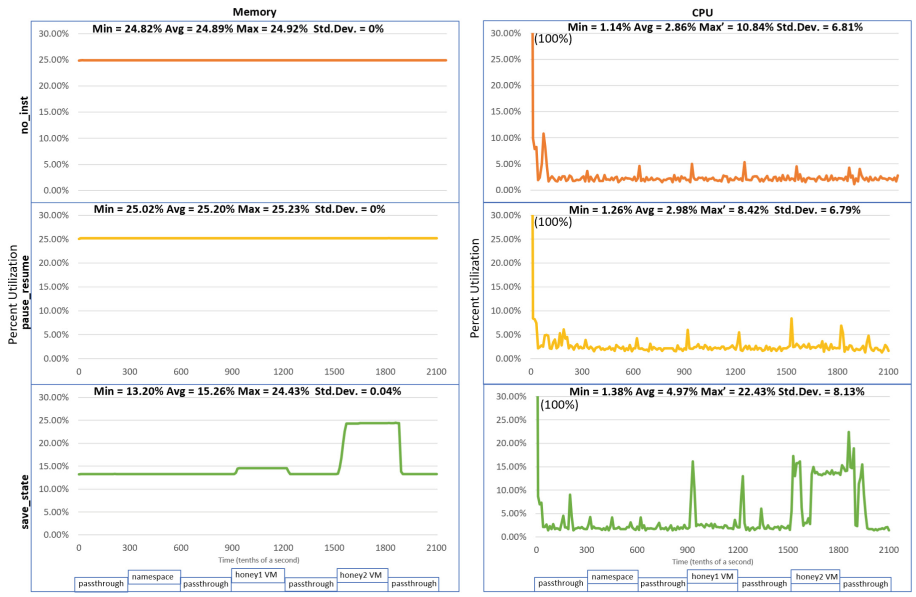

Fig. 8. CPU and memory utilization during the execution of the three configurations.

best. When CPU utilization must remain constant and when delays are a critical factor, then the pause_resume configuration is better suited as it behaves very similar no_inst configuration. The no_inst configuration has no benefits in the case study that we present here. However, there may be situations when honey systems must be configured and tuned dynamically. In this case, a hybrid system would work best. As an example, some honey nodes would be instantiated at th start, but as information becomes available, they could be modified (by starting services or changing IP Addresses/MAC Addresses) through scripting, or stopped all together while another set of honey nodes are instantiated.

7 CDES Performance Against Network Probes

According to [18], reconnaissance is the first step in the Cyber Kill Chain and includes target identification and profiling. We wanted to determine the feasibility of using CDES to directly mitigate this stage in the kill chain by redirecting the traffic of a scanning device during a scan. We ran several tests using the Nmap software on the scanning machine against a recipient node.

Nmap [8] is a well-known tool used to discover nodes on networks as well as the services they are hosting. Nmap accepts a wide range of flags that determine it's behavior, ranging from specifying nodes to specifying network ports, and timing. There are five timing templates included by default in Nmap. These allow a user to indicate general behavior related to how fast and how stealthily a scan should be executed. In general, the highest templates (T4 and T5) favor speed;

T4 is recommended for use on networks with decent broadband or Ethernet connections [8]. T5 is very aggressive and has the potential to present more false negatives. T3 and below are slower and are meant for use on constrained networks. For our study, we used T4 and T5, since our network was small, Ethernet-connected, and also because we wanted to stress test CDES against fast network scans.

We started by documenting the behavior, from a network packet perspective, of Nmap when using the T4 and T5 timing templates. Additionally, we specified the specific IP address of the passthrough node and a specific port (5902 for an open VNC server) on the node; when multiple are specified, they are scanned in a random order. We executed scans from the scanning machine to the passthrough machine, with CDES (running various honey nodes) and a network switch in between (as shown in Fig. 2). This means that Nmap executed a remote network scan, which differs slightly from a internal network scan. The mechanism used to discover a remote node uses ICMP requests and responses, as well as TCP packets, instead of ARP requests and responses. The general set of packets sent with both templates are for the most part the same, with variations in timings due to additional parallelism and other optimizations with T5. Figure 9 shows the behaviors that are most pertinent to our study.

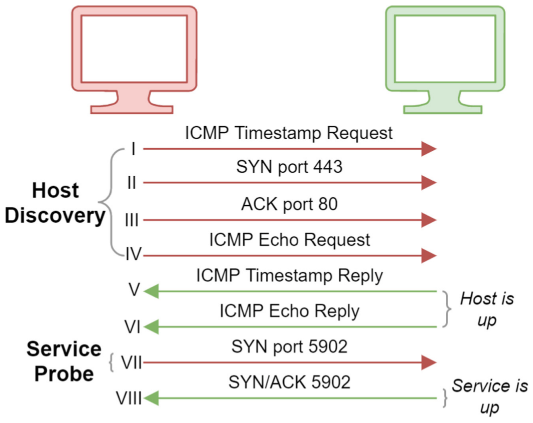

Fig. 9. Network exchanges during Nmap timing templates T4 and T5.

The Nmap behavior occurs in two phases: Host Discovery and Service Probe. Nmap starts by sending a sequence of ICMP and TCP packets (I–IV) to determine if a host is running and responding. The ordering of the packets during the I–VI are not always the same. If the host replies (V and VI), then port probing starts; the scanning machine sends TCP handshake packets (VII) and services that are running will respond (VIII). The TCP packets in the Host Discovery phase do not influence the results of the entirety of the scan. That is, if port 443 was specified as the scan port, an additional TCP SYN packet to port 443 will be sent again during the service probe phase.

We tested whether CDES could swap a connection between the two phases. Specifically, if the swap is fast enough to occur between the time an ICMP Echo Request is observed (IV) to the time that Nmap begins probing for services (VII). This way, even if a real host is discovered, the service probes will be directed to a honey node.

First, we ran Nmap with T4 and T5 five times and recorded timing information. Generally, the total time for the scans, when specifying a port (5902 in our case), were between 0.45 s and 0.54 s for T4; T5 took between 0.44 s and 0.46 s. With both templates, the time between I–IV from Fig. 9 was less then .01 ms. The time from the last packet in Host Discovery (IV) to the first packet in Service probe (VII) occurred within 0.3 s. Therefore, for CDES to achieve our desired effect, it had to respond within 0.3 s.

The CDES Monitor was configured to run a custom network packet sniffer developed using PyPacker [15]. Anytime an ICMP Echo Request packet was encountered, an indicator would print to stdout. This stdout is passed to the Trigger, which then calls the Swapper to switch from the passthrough to a different honey node. Because the timing required to swap to a virtual machine varied depending on the operating system, in addition to the namespace and honey2 VM (TinyCORE) and honey2 VM (Ubuntu 20) used in previous tests, we included 11 additional virtual machines. All VMs are 64-bit operating systems unless otherwise specified.

As discussed in Sect. 5.2, when using the save_state configuration, the time network device activation time gap (roughly 1 s–2 s) is too high to achieve the swap in time, therefore, we only tested using the no_inst and pause_resume configurations. Table 2 shows the results.

CDES was successful in most cases. The most time consuming task is the system calls which are used to control the switch configuration and to VBoxManage, which controls the virtual machines. A note of interest is that when using the T5 template against honey2 VM (TinyCORE), Nmap results indicated a filtered port (as opposed to an open port). This occurred even when scanning it directly, without CDES or any intermediate nodes, and was due to the processing constraints of the VM. As indicated in the Table, with both no_inst and save_state, the swap did occur in time which we validated by analyzing the network traffic on each pathway node.

Looking further into the timings using packet arrival times and recorded timestamps within CDES revealed the following. The time from when a packet is received by the sniffer to when a no_inst swap occurs (when VMs are not instantiated or suspended) is 0.12 s. Within this time, the system call (to the Open vSwitch service) takes 0.10 s to complete. System calls that involve swapping to the honey VMs in the pause_resume case (which includes pausing the previous and resuming the current) varied – we show the total time, from packet arrival through system call in the third column of Table 2. The Ubuntu VMs showed the highest resume times, which are above the 0.3 s threshold before the probes start. One way to alleviate this issue is by using a more capable machine to run the VMs.

Table 2. CDES swap capability against Nmap service probe

Config	Swap from passthrough to	Swap time (seconds)	Win over T4	Win over T5
no_inst	namespace	0.12	Yes	Yes
	Any VM	0.12	Yes	Yes
pause_resume	TinyCORE VM	0.18	Yes	Yes
	Ubuntu 20 VM	0.53	Yes w/defaults	Yes w/defaults
	Alpine 3.11 VM	0.14	Yes	Yes
	CentOS 8.3 VM	0.14	Yes	Yes
	Fedora 3 VM	0.15	Yes	Yes
	FerenOS 2021.01 VM	0.15	Yes	Yes
	Debian 10.7 VM	0.18	Yes	Yes
	Manjaro 21.0 VM	0.19	Yes	Yes
	PopOS 20.04 VM	0.17	Yes	Yes
	Ubuntu 18 VM	0.50	Yes w/defaults	Yes w/defaults
	WinXP 32-bit VM	0.14	Yes	Yes
	Win7 VM	0.16	Yes	Yes
	Win10 VM	0.15	Yes	Yes

When a port is not specified, Nmap will scan the top 1000 most common ports. Using this default behavior, scan completion times ranged from 1.57 s–1.58 s with T4 and 1.56–1.58 with T5. The duration of the Host Discovery phase was the same as when specifying a port (discussed earlier in this section). Chances are higher that CDES will swap before a specific port is probed, resulting in successes when using Ubuntu VMs. This is because ports are not always scanned in the same order.

8 Future Work

We have shown that the open source software, CDES, is a viable solution for employing inline honeynets that can be instantiated and suspended on-the-fly as needed. This solution works well on limited resource devices and with the graphical interface and mechanisms provided by CORE and our adaptations, make this a usable and scalable system. Using this setup, honeynets can range from small-scale (such as using kernel namespaces) to medium-scale (virtual machines) to large-scale external physical machines and networks. Our empirical results show that using CDES has minimal noticeable delay to the connected entities. Finally, we demonstrated that CDES can thwart Nmap probes in real time by redirecting traffic fast enough during an active scan to swap connections before a legitimate node's services are revealed.

The capability demonstrated by this system opens up many possibilities for further improvements, especially in using more sophisticated AI algorithms to decide dynamically which honeynets to activate at any particular time, based on the network monitoring observations. These algorithms can also take into account the available resources and potential impact on load to determine the best course of action. Another direction for research is to test this system using other small-scale systems such as ARM portable devices. We also plan to investigate the feasibility of attaching other software defined networking components such as a controller to synchronize several CDES instances running on multiple devices. Evaluating the performance of the system as well as the effectiveness of different strategies for deploying honeypots in a more complex network setup will also be an important direction for additional experiments.

Acknowledgment. This research was sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not withstanding any copyright notation here on.

References

1. Acosta, J.C., Basak, A., Kiekintveld, C., Leslie, N., Kamhoua, C.: Cybersecurity deception experimentation system. In: 2020 IEEE Secure Development (SecDev), pp. 34–40. IEEE (2020)
2. Basak, A., Kamhoua, C., Venkatesan, S., Gutierrez, M., Anwar, A.H., Kiekintveld, C.: Identifying stealthy attackers in a game theoretic framework using deception. In: Alpcan, T., Vorobeychik, Y., Baras, J.S., Dán, G. (eds.) GameSec 2019. LNCS, vol. 11836, pp. 21–32. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32430-8_2
3. dmpayton: Django Admin Honeypot. https://github.com/dmpayton/django-admin-honeypot. Accessed 20 Mar 2021
4. huuck: ADBHoney. https://github.com/huuck/ADBHoney. Accessed 20 Mar 2021
5. Jicha, A., Patton, M., Chen, H.: SCADA honeypots: an in-depth analysis of Conpot. In: 2016 IEEE Conference on Intelligence and Security Informatics (ISI), pp. 196–198. IEEE (2016)
6. KeyFocus: KFSensor. http://www.keyfocus.net/kfsensor/features/. Accessed 20 Mar 2021
7. Sandia National Laboratories: HADES. https://www.osti.gov/servlets/purl/1525940. Accessed 29 Mar 2021
8. Lyon, G.F.: Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning. Insecure. Com LLC, US (2008)
9. Nazario, J.: Awesome-Honeypots. https://github.com/paralax/awesome-honeypots. Accessed 20 Mar 2021
10. Provos, N.: Honeyd-a virtual honeypot daemon. In: 10th DFN-CERT Workshop, Hamburg, Germany, vol. 2, p. 4 (2003)
11. Razmjoo, A.: OWASP Honeypot. https://github.com/OWASP/Python-Honeypot/wiki. Accessed 20 Mar 2021

12. Cymmetria Research: Struts Honeypot. https://github.com/Cymmetria/StrutsHoneypot. Accessed 20 Mar 2021
13. schmalle: Nodepot. https://github.com/schmalle/Nodepot. Accessed 20 Mar 2021
14. Spitzner, L.: The honeynet project: trapping the hackers. IEEE Secur. Priv. **1**(2), 15–23 (2003)
15. Stahn, M.: pypacker. https://gitlab.com/mike01/pypacker. Accessed 20 Mar 2021
16. travisbgreen: AMT Honeypot. https://github.com/travisbgreen/intel_amt_honeypot. Accessed 20 Mar 2021
17. TrendMicro: GasPot. https://github.com/sjhilt/GasPot. Accessed 20 Mar 2021
18. Yadav, T., Rao, A.M.: Technical aspects of cyber kill chain. In: Abawajy, J.H., Mukherjea, S., Thampi, S.M., Ruiz-Martínez, A. (eds.) SSCC 2015. CCIS, vol. 536, pp. 438–452. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22915-7_40