A First Look: Using Linux Containers for Deceptive Honeypots

Alexander Kedrowitsch
Department of Electrical Engineering and Computer Science, United States Military Academy
West Point, New York, USA
alexander.kedrowitsch@usma.edu

Danfeng Yao, Gang Wang, Kirk Cameron
Department of Computer Science, Virginia Tech
Blacksburg, Virginia, USA
{danfeng,gangwang,cameron}@cs.vt.edu

ABSTRACT

The ever-increasing sophistication of malware has made malicious binary collection and analysis an absolute necessity for proactive defenses. Meanwhile, malware authors seek to harden their binaries against analysis by incorporating environment detection techniques, in order to identify if the binary is executing within a virtual environment or in the presence of monitoring tools. For security researchers, it is still an open question regarding how to remove the artifacts from virtual machines to effectively build deceptive "honeypots" for malware collection and analysis.

In this paper, we explore a completely different and yet promising approach by using Linux containers. Linux containers, in theory, have minimal virtualization artifacts and are easily deployable on low-power devices. Our work performs the first controlled experiments to compare Linux containers with bare metal and 5 major types of virtual machines. We seek to measure the deception capabilities offered by Linux containers to defeat mainstream virtual environment detection techniques. In addition, we empirically explore the potential weaknesses in Linux containers to help defenders to make more informed design decisions.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; *Software security engineering*;

KEYWORDS

Deception; Honeypots; Virtual Machine; Linux Containers

1 INTRODUCTION

As malware and botnets grow in sophistication, many malware authors attempt to harden their malicious binaries against security analysis and reverse-engineering by conducting environment checks [17]. When malware detects a virtual environment or monitoring tools, it may modify its behavior during execution to evade detection. Researchers have suggested that up to 40% of the malware in the wild alter their malicious behavior when executing in a virtual environment or a debugger [3] and the defeat of their detection methods remains an open problem within the security community today.

The evasive behavior of malware presents a significant challenge for security researchers who implement high-interaction honeypots to capture and analyze malware. This is particularly true for organizational networks, government agencies and large data centers, which have recently become the primary targets of malware infections [9]. If the virtualized nature of their honeypots is detected, it will cause major false negatives, leading to significant delays for discovering and reporting malware infections for the rest of the networks and organizations.

In recent years, the ubiquity of Linux systems, including the rapid growth of IoT devices running Linux, calls for new Linux-based honeypots that can defeat virtual environment detection techniques. Indeed, early in 2017, security researchers have discovered what is believe to be the first example of VM-aware malware targeting Linux-based IoT systems with the identification of the "Amnesia" botnet that exploits the DVR component of specific CCTV cameras [27].

Under this trend, it is high time to explore new methodologies for developing Linux-based honeypots that can defeat adversarial environment tests from evasive malware. In this paper, we explore the possibility of using Linux containers as a substitute for virtual machines (VM) in high-interaction honeypots. A Linux container is an OS level virtualization method for running multiple isolated Linux systems (*i.e.*, containers) using a single Linux kernel. A Linux container offers a similar environment as a VM without the need for running a separate kernel or simulating the hardware. This design has the potential to remove numerous artifacts from the VM as an out-of-the-box defense against virtual environment detection techniques. In addition, the low overhead of running Linux containers also allows for much greater scalability of honeypot deployment. Although Linux containers are promising *in theory*, its effectiveness against VM detection techniques has not been sufficiently evaluated empirically so far.

In this work, we provide the first systematic evaluation on Linux containers using controlled experiments to answer three key questions. First, how well can Linux containers defeat common virtual environment detection methods (§4)? Second, how well can Linux containers defeat monitoring tool detection methods (§5)? Third, what are the new artifacts (if any) that Linux containers introduce that can be exploited by future malware (§6)? By thoroughly discussing the advantages and potential new problems of Linux containers, we hope our results can help future honeypot designers to make more informed decisions.

We answer these questions by setting up a realistic testbed to run a series of VM detection tests on Linux containers in a variety of hardware settings (*e.g.*, 3 types of CPU chipsets are used). The

SafeConfig'17, November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5203-1/17/11…$15.00
https://doi.org/10.1145/3140368.3140371

implemented tests focus on hardware-based detection methods which are historically the most difficult artifacts to mask in virtual environments. We compare the performance of Linux containers with "bare mental" and 5 mainstream virtual machine software.

Our study has 3 key findings (or contributions):

- Our experiments confirm that mainstream VMs can be (easily) identified by hardware-level environment detection techniques such as CPU clock sampling, reported CPU information and instruction execution time. A Linux container, for its lack of virtualization and direct interface with the host kernel, has returned a similar profile to "bare metal", defeating these detection methods natively.
- An initial investigation shows Linux containers are very promising to defeat environment detection methods that examine in-host monitoring tools. This is largely attributed to Linux containers' employment of kernel namespaces, which grants in-host monitoring with high semantic details and helps to overcome detection methods that check popular monitoring and debugging tools.
- We also find that Linux containers are vulnerable to new methods of identification that exploit the implementation tools of containers, such as namespaces and permissions. Some of the new methods are more difficult to run for attackers (*e.g.*, requiring root), but not impossible. This suggests honeypot designers need to carefully handle these artifacts when deploying Linux containers for deceptive malware analysis.

Overall, the use of Linux containers for deception-based security provides an additional and beneficial tool for researchers if implemented with a strong set of assumptions on an attacker's environment detection methods. Our investigation comes at a critical time when IoT-targeting malware is increasing in frequency and threat [1]. Leveraging Linux containers on low-power devices grants highly scalable honeynets that are capable of defeating many virtual environment detection methods. Our result also shows the possibility of new adversarial tests against Linux containers. Future research will look into possible countermeasures against container-based environment tests.

2 BACKGROUND AND RELATED WORK

2.1 Honeypot

As botnets pursue ever-stealthier means of communication, it is simply insufficient for security researchers to monitor network traffic in order to study the proliferation and activity of malicious software [28]; monitoring malware behavior on a host becomes increasingly essential. Discussed frequently throughout literature is the importance of the role honeypots play in the ongoing struggle to understand and defeat botnets and other malware [4, 11, 18, 20, 23, 24, 26]. A honeypot is typically defined as a system (or group of systems) that is designed to pose as a legitimate server waiting to receive incoming connections, but has no production value; instead, it monitors and logs all interactions it has with outside entities [20].

Honeypots are widely developed within the open-source community with nearly 1,000 honeypot related repositories on GitHub possessing functionalities that span a wide range of capabilities. High-interaction honeypots are fully functional servers and systems running in virtual environments in order to provide containment and isolation from production systems as well as aid monitoring, where low/medium-interaction honeypots are *emulations* of given systems and services consisting of rapid approximations of how the advertised services would respond to network requests [24].

Combining the low-cost versatility afforded by low-power single-board computers with the emerging use of honeypot sensors in a *honeynet*, security researchers and administrators are given the ability to deploy cheap sensors throughout one or more networks to collect data on network penetration attempts and new exploits as they are used in the wild [5]. The only limitation in such a *honeynet* is the reduced deception afforded by low/medium-interaction honeypots as the CPU resource overhead imposed by virtual machines for high-interaction honeypots on single-board computers is prohibitive.

2.2 Virtual Environment

Virtual machines provide an ideal environment for use in high-interaction honeypots for containment of malware and fast image restoration [24] [11] [4]. After a successful penetration by malware, actions can be safely contained within the VM and, upon attack completion, the environment can be quickly restored.

2.3 Linux Containers

Linux containers are the product of tying together two Linux technologies, namespaces and cgroups, that provide isolation and containment for one or more processes from the rest of the host, effectively abstracting applications away from the operating system. The initial development of containers is called Linux Containers (otherwise referred to as LXC), which allows multiple Linux systems to be run on the same host, sharing only the system kernel. Each of these containers 'feel' like its own entire Linux system and are isolated from both each other and the host system (as opposed to container implementations such as Docker).

A significant component to the reduced overhead of Linux containers is that they execute on the host kernel, thus do not duplicate any kernel functions. As an isolated example, during experimentation a particular code sample added approximately 6% execution time when compared to bare metal execution on the same system; the same code sample added approximately 41% when being executed in KVM.

Implementing high-interaction honeypots with containers on low-power devices provides a new intersection between honeypot interaction level and system resource expense (as depicted in 1). The full amount of interaction afforded by virtual environments in high-interaction honeypots can now be utilized on low-power devices through the use of Linux containers.

It needs to be noted that honeypots executing within Linux containers carry different security concerns than honeypots within a virtual environment. This is due to code within the container being run on the same kernel as the host machine. Mechanisms are in place that attempt to minimize the impact a container may have on the performance of the host machine, but risk consideration must be taken into account.

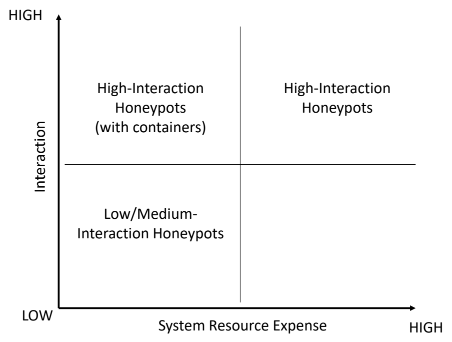

HIGH
Interaction
High-Interaction Honeypots (with containers)
High-Interaction Honeypots
Low/Medium-Interaction Honeypots
LOW
System Resource Expense
HIGH

Figure 1: Chart depicting traditional honeypot interaction capabilities contrasted with system resource expense.

2.4 Related Work

Much of the attention regarding VM-aware malware has focused on malicious binaries already loaded on to a system that, during execution, perform some environment checks in order to determine if it is operating in a virtual environment [6] [12] [21]. These environment checks can range from simply checking the file system for tell-tale indicators such as standard VMWare network adapter drivers, to testing for the presence of model-specific CPU errors that aren't replicated in a simulated system [12].

One of the most well-known detection mechanism is related to timing, where specific operations are performed and the elapsed time is compared to an expected reference; if the operation took longer than expected or frequent executions of the operation have a wide variance in execution time, a program can assume it is operating in a virtual environment [21]. The reason for the delay is inherent to the virtualization technology, where, at minimum, several CPU instructions are either emulated in software or captured and translated to other instructions, thus creating an additional time overhead [7].

Bare Metal Implementation. A mechanism to implement several of the key features of using virtual machines for binary analysis on bare metal systems was explored in [12] where a highly modified OS allowed a state of the bare metal execution environment to be captured and later restored after analysis of the malware execution was complete. Unfortunately, the framework is complicated and requires a modification of the running operating system that will be dependent upon the hardware utilized by the bare metal system.

Transparent Virtual Environments. Attempts have been made to develop fully transparent virtual environments that are invisible to malware; Cobra and Ether are two often cited examples of this attempt. However, both virtual environments had ultimately fallen short of their goal, revealing their presence either through inaccurate CPU semantics or through timing tests with verification from outside sources [21] [19]. It has even been argued by Garfinkel, et al. in [8], that a fully transparent virtual environment is impossible to achieve due to the necessary deviations virtual environment developers must make that are different from the hardware they are emulating.

VM-Aware Malware Detection. In [2], Balzarotti, et al. demonstrated a system that compares the behavior of malware executing on highly controlled bare-metal and a virtual system. The authors found their technique to be reliable and efficient at detecting what they referred to as a 'split personality', where malware behaves differently depending on the environment it is being run in.

The three techniques outlined have yet to provide researchers with a currently usable and easy to implement tool that allows deceptive environments to avoid virtual environment detection. The capabilities of Linux containers grant easy replacement of virtualization in deceptive environments and is capable of defeating many well-known virtual environment detection methods out of the box.

3 SECURITY MODELS

This paper focuses on the threats to Linux systems from sophisticated malicious binaries attempting proliferation through network connections. The malware seeks to evade the sandbox analysis by proactively detecting the virtual environment and monitoring tools and altering behavior accordingly. Our paper explores the possibility of using Linux containers to defeat common tactics used by malware authors to detect virtual environments. Below, we briefly discuss the common methods by which a binary can identify the malware sandbox by detecting a virtual environment or monitoring and debugging tools.

Detecting the VM Environment. Virtual environment detection can be broadly divided into the categories of operating system artifacts and hardware artifacts [2, 3, 6, 13, 21]. Operating system artifacts are tell-tale signs that a binary is operating within a virtual environment through the names of drivers and processes, the presence of specific files, or the configuration of the operating system. Hardware artifacts include indicators from instruction execution, such as increased execution time or increased variability in execution time, or hardware configurations found only in virtual environments, such as abnormal CPU information reporting or hardware identifiers specific to virtual environments. As an example, the detection method utilized in the previously mentioned 'Amnesia' botnet detects hardware artifacts listed in /SYS/CLASS/DMI/ID/ PRODUCT_NAME and /SYS/CLASS/DMI/ID/SYS_VENDOR by searching for the text strings 'VirtualBox' and 'QEMU' [27].

In order to conduct a preliminary investigation into Linux container's abilities to defeat virtual environment detection methods, select tests were generated based on discussions in literature, as well as online security forums, and tested against an array of system configurations, both to validate the ability of the test to detect a virtual environment as well as test its detection of a Linux Container. In order to be functional virtual environment detection methods, the tests must be capable of identifying when it is executing in a bare metal environment and when it is executing in a virtual one.

As most operating system artifacts, such as tell-tale network interface card MAC addresses, can be eliminated through careful configuration [17], the focus of this investigation is on detection methods that identify abnormalities in the hardware environment and performance. The selected detection methods to test are:

- Variability and execution time in CPU clock sampling
- Reported CPU information

- Instruction execution time

Detecting Debugging and Monitoring Tools. In addition to detecting virtual environments, malware authors may also try to recognize the malware analysis sandbox by detecting the presence of monitoring and debugging tools. Evidence shows that malware has indeed tried to probe the debugging software in an attempt to thwart monitoring and analysis of their attacks [2, 3]. Our experiment will test related tactics on Linux containers.

Roadmaps. In section 4, we conduct three experiments in order to assess the feasibility of Linux Containers to defeat traditionally difficult virtual environment detection methods. In addition to defeating environment detection tests, we investigate the ability of Linux containers to defeat two different types of monitoring tool detection methods in Section 5. Additional deception techniques enabled by Linux Containers, specifically when implemented on low-power devices, are also explored in Section 5. In Section 6, detection methods of Linux Containers are investigated to determine if containers are susceptible to their own unique types of detection.

4 VM DETECTION EXPERIMENTS

4.1 Experiment Setup

In order to avoid a bias against particular hardware architecture goals and virtual environments, a variety of CPU chipsets and virtual environment software were included in this study. The hardware chipsets included in this study are:

- 1 x Minnowboard Turbot Intel Atom x86 64-bit Single-Board Computer
- 1 x "Desktop Class" Intel Core i5-2400 system
- 1 x "Server Class" Intel Xeon E5320 system

Additional low-power devices using the ARM architecture were sought for this study as they are a frequently used low-power device chipset. Unfortunately, due to driver incompatibilities, a common testing environment across all hardware platforms with the chosen operating system could not be established that would allow direct comparison across all systems. Additional ARM hardware support included in recent Linux mainline kernel versions is expected to resolve this dilemma in the near future.

Each system was installed with Ubuntu 16.04 as the host operating system. Ubuntu was selected due to its prominence as a Linux desktop environment and in commercial servers. Canonical is also the developer for LXC and is expected to provide the greatest support for Ubuntu environments. Ubuntu 16.04 was the latest Long-Term Support version available at the start of this study.

LXD version 2.12 was installed on all systems. LXD is the management daemon for LXC developed by Canonical, which provides easier management of LXC containers.

Virtual environments included in this study were sought to identify popular types that span multiple implementations. The virtual environment software included in this study consists of:

- VMWare Workstation, Ver 12.5.2
- QEMU, Ver 2.5.0
- KVM, Ver 2.5.0
- Xen Paravirtualized (PV), Ver 4.6.0
- Xen Hardware Assisted (HVM), Ver 4.6.0

Software	Type	Implementation
QEMU	Type II	Full software emulation
KVM	Type II	Hardware assisted
VMWare	Type II	Full virtualization / Hardware assisted
Xen PV	Type I	Paravirtualized
Xen HVM	Type I	Hardware assisted

Table 1: Table of tested virtual environments and their properties.

Table 1 provides a brief listing of the implemented virtual environments and their properties; the breakdown of virtual environment installation to physical host is included in Table 2.

4.2 Experiment Tests

In order to identify Linux container's ability to defeat VM detection three traditionally challenging detection methods to beat were chosen for testing:

- Variability and execution time in CPU clock sampling
- Reported CPU information
- Instruction execution time

CPU clock variability seeks to sample the various system clocks that are available on modern CPUs a high number of times, then determine the amount of variability in the clock samples by calculating a mean and standard deviation. This is a test devised from an online discussion regarding 'red pill' virtual environment detection [22]. Due to the sharing of system resources and the need to handle hardware and software interrupts on both the guest and host system, it is expected that virtual machines will demonstrate a variability in their clock timing sources that is not present in bare metal operating environments.

CPU information testing merely queries the system for information on the CPU and attempts to determine if there are any abnormalities such as a non-standard CPU name or an unexpected number of CPU cores. This test has been identified in [6] as an effective means for detecting certain virtual environments due to how the hypervisor, or the configuration of the hypervisor, presents the physical attributes of the bare metal system to the virtual environment. An additional test, demonstrated in code from [15], shows an effective means of detecting hypervisors that advertise themselves by the CPUID instruction on x86 processors. When executing CPUID, the 31st bit of the ECX register can (optionally) return a value of '1' if a hypervisor is present [25].

Execution time of instructions was outlined in several papers [3] [8] [19] [21] as a candidate for virtual environment detection. The overall concept is that by adding an additional layer of abstraction in virtual environments, instruction execution will show a characteristic increase. While simply tracking the amount of time taken to execute an instruction may indicate the presence of a virtual environment, the detection scheme would require a database of baseline performance values for comparison. As recommended by Garfinkel, et al., in [8], the instruction execution timing test compares the execution time between two different instructions as a means of generating a stand-alone comparison metric that does not rely on referencing baseline performance values.

4.3 Experiment Results

The virtual environment detection methods were generally successful in identifying the presence of a VM with the notable exceptions of QEMU and Xen HVM. Due to QEMU being a hardware emulator, it lacks a hypervisor, thus behaves similarly to bare-metal. However, QEMU's extremely slow performance can be easily detected when tested for. Xen HVM distinguished itself as a highly efficient hypervisor and was the most resilient against detection, revealing itself only when being selectively targeted during the Instruction Execution Time test.

CPU clock variation was measured by sampling the timestamp counter preceded by a CPUID instruction call, which was identified as providing the greatest variability in execution. The timestamp counter was sampled 255 times with a coefficient of variation being calculated using the mean and standard deviation of time elapsed between samples. Figures 2(a), 2(b), and 2(c) show a coefficient of variation between 0.01 and 0.1 provided a good measure of detecting non-virtual environments; coefficient of variation values outside of those ranges were exclusively virtual. False negatives were generated for QEMU and Xen HVM, which demonstrated bare metal coefficients of variation. VMWare was omitted from figure 2(b) due to its significantly higher coefficient of variation value of 1.3.

When polling the test systems' CPU information, most of the virtual systems returned non-standard CPU results as illustrated in Table 3. Both VMWare and Xen PV returned the name of the bare metal CPU the virtual machine was executing on, however the number of cores was not consistent with manufacturer specifications. Intel's core i5-2400 processors support at minimum 2 cores, while Intel's Xeon E5320 have 4 cores. Both of these non-standard results can indicate to malware that it is not executing in a bare metal environment so long as the malware has a reference for the proper number of cores for each CPU model.

QEMU, KVM, and VMWare hypervisors advertise their presence within the CPUID registers, which quickly identifies those environments as virtual; neither Xen PV nor Xen HVM advertise their presence in CPUID. Only Xen HVM remained undetected by this test by both correctly relaying all appropriate CPU information as well as not advertising its presence in CPUID.

As Linux containers poll the system information directly from the kernel, they returned results matching those of the bare metal system.

As stated earlier, the instruction execution timing test was generated by comparing the execution time between two different operations. A simple arithmetic operation was selected for a timing baseline with the CPUID instruction selected as the second operation. CPUID was selected as the target function due to being an unprivileged instruction call that interacts directly with the CPU, anticipating a required interaction with the virtual environment hypervisor to is expected to impact execution time. Dividing the target function execution time by the base function execution time generated an execution time ratio that was observed for various environments.

Figure 3, displaying the results of 255 trials, illustrated that any ratio value greater than 1 indicated the presence of a VM with false negatives for QEMU and Xen. Replacing the CPUID instruction call with executing a memory mapping operation did successfully identify both QEMU and Xen environments, however the result is not as generic as the original CPUID test who's indication ratio remained consistent across all platforms tested; memory mapping performance varies greatly across platforms and a single, platform-agnostic ratio could not be identified for this instruction.

As expected, due to the difference in isolation techniques employed by Linux containers, as opposed to virtual environments, the experiments identified that Linux containers are highly effective in defeating the examined virtual environment detection methods. A more detailed explanation of the tests implemented and the results found was omitted for brevity due to the experiments confirming intuitive expectations.

System	**Installed Virtual Environments**
Atom	KVM
Core i5	QEMU, KVM, VMWare
Xeon	QEMU, KVM, VMWare, Xen PV, Xen HVM

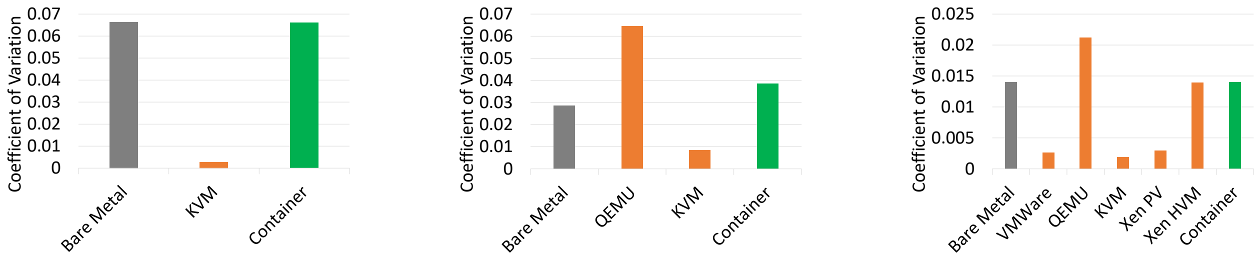
Table 2: Table of systems and installed virtual environments.

	System	**CPU Information**	**CPUID Flag**
Atom	**Bare Metal**	**Atom E3826 x 2**	**No**
	KVM	QEMU Virtual CPU	Yes
	Container	**Atom E3826 x 2**	**No**
i5	**Bare Metal**	**i5-2400 x 4**	**No**
	VMWare	i5-2400 x 1	Yes
	QEMU	QEMU Virtual CPU	Yes
	KVM	QEMU Virtual CPU	Yes
	Container	**i5-2400 x 4**	**No**
Xeon	**Bare Metal**	**Xeon E5320 x 8**	**No**
	VMWare	Xeon E5320 x 1	Yes
	QEMU	QEMU Virtual CPU	Yes
	KVM	QEMU Virtual CPU	Yes
	Xen PV	Xeon E5320 x 2	No
	Xen HVM	Xeon E5320 x 8	No
	Container	**Xeon E5320 x 8**	**No**

Table 3: Results of CPU Information Test.

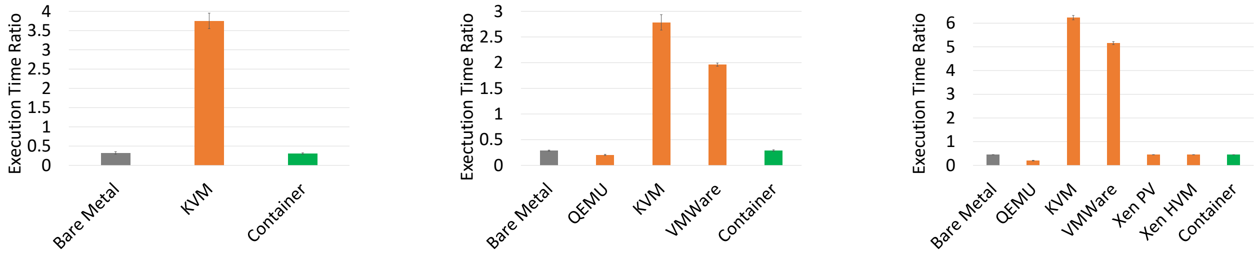
5 MONITORING TOOL DETECTION

Rather than exclusively detecting virtual environments, malware authors have also attempted to detect the presence of monitoring and debugging tools as a means to avoid analysis. As discussed in [2] and [3], malware has been identified in the wild that attempts to detect debugging software either in conjunction with, or in replacement of, virtual environment detection tests in an attempt to thwart monitoring and analysis of their attacks. This is a logical approach for malware authors who don't wish to exclusively target end-user platforms, as the use of virtual environments has grown tremendously in professional applications; to avoid attacking virtual environments would be to limit an attacker from going after the potentially most profitable targets.



(a) Intel Atom (b) Intel Core i5 (c) Intel Xeon

Figure 2: Coefficient of variation values of CPUID+RDTSC clock sampling for various environments on multiple host platforms.



(a) Intel Atom (b) Intel Core i5 (c) Intel Xeon

Figure 3: Ratio results comparing arithmetic operation performance to CPUID performance for various environments on multiple host platforms.

Fortunately, Linux containers, by use of their unique features, may present a much less technologically challenging solution than what was presented by Jian, et al., in [10] where the hypervisor reconstructs semantic details of attack behavior while remaining off the host to avoid detection, but with similar results for certain types of monitoring tool detection methods.

Containers Defeating Recursive strace Detection.
A useful and popular monitoring tool used to record the behavior of malware is STRACE, which allows the kernel to output the system calls a process and its children make, providing a great amount of behavior detail.

Unfortunately, a very simple test is available for malware to determine whether it is being monitored by STRACE - attempting to use STRACE on itself or one of its child processes. Due to an implementation restriction in the tool, STRACE is not able to be called recursively and fails, alerting the malware to the fact it is being monitored.

Linux containers, through the means of process isolation, are able to circumvent the recursive STRACE detection method. From the host, STRACE may be called on any process executing within the container (or on the container itself) in order to monitor the system calls made by that process. Due to container isolation, if the process being monitored attempts to call STRACE on itself or one of its children, STRACE will execute normally and falsely indicate to the malware that it is not under analysis.

Containers Defeating Debugger Detection in /proc/self/status.
Another debug detection mechanism that is available to malware is reading information present in the Linux /proc/self/status file in the process file system. By searching that file for the text string "TracerPid:", its presence will indicate to a malicious binary that a debugging tool, such as GDB or STRACE is currently monitoring its execution. A debugger detection test was written using a slightly modified code example provided by Sam Liao in [14], testing for for both GDB and STRACE and correctly identified the presence of a monitoring tool. However, when using STRACE from the host system to monitor the process within a Linux container, the tool did not detect the presence of the process tracer.

These preliminary tests show great promise in using Linux containers to defeat debugger and monitoring tool detection methods, which are an identified and present threat to security analysis. Using the terminology of Jiang, et al., this unanticipated benefit from the use of Linux containers grants researchers "in-host" semantic detail of malware behavior with "out-of-host" detection prevention.

	Atom	i5	Xeon
Max	122.33	8.95	8.30
Median	113.97	8.58	7.66
Min	10.16	7.58	7.52

Table 4: Max, min, and median CPU cache latency values (ns) for Intel Atom, Core i5, and Xeon CPUs.

Additional Deception Capacities for Low-Power Devices. There are additional deception capabilities offered by Linux containers when employed for the purposes of honeypots and malware analyzation beyond just masking its presence from traditional virtual environment detection methods, but they are not without their limitations.

As containers are abstracted from many aspects of the host OS, each container can have its particular software load-out highly configured, to include running entirely different Linux distributions, so long as they are compatible with the host kernel. This grants great flexibility for security researchers who need to satisfy specific software and version installation requirements for malware that is targeting a very specific exploit.

6 CONTAINER DETECTION

Adopting Linux container usage as a means to defeat virtual environment detection tests leads to a follow-on question: Can Linux containers be detected by malware just as virtual environments are? As virtual environment detection tests generally focus on discovering discrepancies between what a bare metal machine will look and behave like versus what a virtual machine looks and behaves like, Linux container detections tests would seek to identify traits or characteristics that would be different when executed inside a container versus bare metal.

As Linux containers are managed by the operating system kernel and execute directly on the hardware, many of the virtual environment detection discussed earlier are not effective against Linux containers due to the different attack surface presented; attackers will need to focus their attention on the possible gaps in isolation methods employed by Linux containers in order to identify the virtual environment.

6.1 Hardware Classification from within a Container using Cache Latency

When considering the deception capabilities of Linux containers when used as honeypots, a natural question may arise whether the host hardware can be masked and presented as something different, as may be done in several virtual environments. Unfortunately, due to the container executing directly on the kernel of the host system, the OS kernel and hardware layout must match that of the host system. This limitation is unfortunate as it would be beneficial for a lower-power, low-cost system to masquerade as a high-end system. Even considering the situation where a kernel may be configured to present a false hardware profile to system calls, the actual performance of the system hardware can reveal its true nature.

As an example, an attacker may attempt to identify a low-power device by its cache latency; low-power CPUs often sacrifice performance for power efficiency and can be identified by their significantly reduced performance. Using the program LMBENCH, cache latency was measured for arrays ranging in 0.35 to 1.5 MB in size, being read using Fread with a stride size of 512kb on multiple bare metal hosts. The results, shown in Table 4, clearly identify which host has the low-power CPU.

6.2 Container Detection using Namespaces

An example of container detection through namespace isolation was identified in late 2016, being employed by the VLANY LD_PRELOAD rootkit malware source code that was published online [16]. The VLANY rootkit is designed to contain a large number of stealth and anti-detection features, one of which is a technique to identify if the code is running within an LXC container by means of comparing the number of processes listed by the PS command and the number of processes provided by SYSINFO.

This test identified a lapse in coverage by Linux kernel namespaces in that PS lists the processes owned by the container, yet the SYSINFO command still enumerates the total number of processes running on the entire host. If PS provides a significantly smaller process count than SYSINFO (VLANY rootkit tests for a discrepancy greater than 200), then it is assumed that PS is restricted by user namespaces and hiding the true number of total processes. Results for the Namespace test are listed in 5, illustrating accuracy at correctly identifying container environments with the exception of the Atom system, which operates with a non-GUI installation of Ubuntu and has a significantly lower process count than the other Ubuntu installations.

It is anticipated that SYSINFO would need to be updated to account for kernel namespaces in order to prevent this type of container discovery method; however, other lapses in namespace isolation may also be present.

6.3 Container Detection using Permissions

Another area in which Linux containers may be revealed is in user permissions. Through the use of containers, there arises a unique condition in which a user may be in possession of root privileges in a container, but the container itself is unprivileged. Within the container namespace, such a user will be operating with a UID of 0, however, on the host, the user's actual UID will be that of an unprivileged user. This generates a scenario in which a user with root privileges in a container may attempt to perform an operation on the system that they expect to be successful but are denied.

As an example, the Linux command DMIDECODE requires root privilege in order to access the Desktop Management Interface table. When attempting to perform the Linux command DMIDECODE as a root user in an unprivileged container, the kernel reveals the lack of requisite permissions. Results are presented in Table 5 and demonstrate container detection with perfect accuracy.

Defeating permissions-related container detection tests remains an open problem. A possible solution would be for privileged users within an unprivileged container to be granted read-only permissions in order to access system resources. Unfortunately this will not be an effective counter-measure for permissions checks that

	System	Namespace Test	Permission Test
Atom	**Bare Metal**	**No**	**No**
	KVM	No	No
	Container	**No**	**Yes**
i5	**Bare Metal**	**No**	**No**
	VMWare	No	No
	QEMU	No	No
	KVM	No	No
	Container	**Yes**	**Yes**
Xeon	**Bare Metal**	**No**	**No**
	VMWare	No	No
	QEMU	No	No
	KVM	No	No
	Xen PV	No	No
	Xen HVM	No	No
	Container	**Yes**	**Yes**

Table 5: Results of Container Detection Tests.

seek to modify system aspects, such as the CPU governor. As such, at this time there is no effective means to defeat this Linux container environment test for an attacker with root access.

7 CONCLUSION

This paper explored the use of Linux containers as a means to defeat several types of virtual environment and monitoring tool detection methods when used as an alternative to virtualization. Additionally, this paper explored the deception capabilities currently offered by Linux containers, as well as their suitability for deployment on low-power devices due to the minimal resource overhead.

During the investigation it was identified that there are limitations to the deception such systems are capable of, such as the understanding that the hardware and kernel presented within the container must match that of the host system; security researchers and administrators need to be aware of such limitations in order to make informed decisions on what deception tactics are appropriate for low-powered devices.

Of serious concern is the apparent readily available container detection methods investigated. While successfully hiding their presence to VM detection methods, containers appear to be susceptible to tests seeking to identify their presence. Namespace isolation tests may potentially be correctable in future versions; however, permission discrepancies, particularly when containers are employed for honeypots that are intended to grant a root interface to an attacker, are easily identifiable and lack a straightforward solution.

As such, containers should be employed to defeat environment detection methods only when container detection is not anticipated to be employed. Without a clear road ahead to resolving the ease of detectability, the use of Linux containers as honeypots, while ideally suited for deployment on low-powered devices, may have a rocky and short-lived future.

8 ACKNOWLEDGMENTS

This work has been supported in part by NSF grants CNS-1565314 and CNS-1717028.

REFERENCES

[1] Srini Avirneni. 2017. The Rise of Open-Source Malware and IoT Security. https://www.forbes.com/sites/forbestechcouncil/2017/04/05/the-rise-of-open-source-malware-and-iot-security/. (2017).

[2] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2010. Efficient Detection of Split Personalities in Malware.. In *Proc. of NDSS.*

[3] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proc. of DSN.*

[4] WY Chin, Evangelos P Markatos, Spiros Antonatos, and Sotiris Ioannidis. 2009. HoneyLab: large-scale honeypot deployment and resource sharing. In *Proc. of NSS.*

[5] Andreas Christoforou, Harald Gjermundrød, and Ioanna Dionysiou. 2015. HoneyCY: A configurable unified management framework for open-source honeypot services. In *Proc. of the Panhellenic Conference on Informatics.*

[6] Peter Ferrie. 2007. Attacks on more virtual machine emulators. *Symantec Technology Exchange* 55 (2007).

[7] Jason Franklin, Mark Luk, Jonathan M McCune, Arvind Seshadri, Adrian Perrig, and Leendert Van Doorn. 2008. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating Systems Review* 42, 3 (2008), 83–92.

[8] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility Is Not Transparency: VMM Detection Myths and Realities.. In *Proc. of HotOS.*

[9] FireEye Inc. 2017. M-Trends: Trends from the Year's Breaches and Cyber Attacks. https://www.fireeye.com/current-threats/annual-threat-report/mtrends.html. (2017).

[10] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 2007. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proc. of CCS.*

[11] John P John, Fang Yu, Yinglian Xie, Arvind Krishnamurthy, and Martín Abadi. 2011. Heat-seeking honeypots: design and experience. In *Proc. of ICWWW.*

[12] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2011. BareBox: efficient malware analysis on bare-metal. In *Proc. of ACSAC.*

[13] Boris Lau and Vanja Svajcer. 2010. Measuring virtual machine detection in malware using DSD tracer. *Journal in Computer Virology* 6, 3 (2010), 181–195.

[14] Sam Liao. 2014. How to detect if the current process is being run by gdb. http://stackoverflow.com/questions/3596781/how-to-detect-if-the-current-process-is-being-run-by-gdb. (2014).

[15] LordNoteworthy. 2016. Al-Khaser. https://github.com/LordNoteworthy/al-khaser/tree/ff8d53891709b407cbf43a323abc302730504fae. (2016).

[16] mempodippy. 2016. VLANY. https://raw.githubusercontent.com/mempodippy/vlany/master/misc/detect_lxc.c. (2016).

[17] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. 2017. Spotless Sandboxes: Evading Malware Analysis Systems using Wear-and-Tear Artifacts. In *Proc. of IEEE S&P.*

[18] Hamid Mohammadzadeh, Masood Mansoori, and Ian Welch. 2013. Evaluation of fingerprinting techniques and a windows-based dynamic honeypot. In *Proc. of Australasian Information Security Conference.*

[19] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. 2011. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In *Proc. of European Workshop on System Security.*

[20] Niels Provos et al. 2004. A Virtual Honeypot Framework.. In *Proc. of USENIX Security.*

[21] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. 2007. Detecting system emulators. In *Proc. of ICISC.*

[22] SecuriTeam. 2004. Red Pill... Or How to Detect VMM Using (Almost) One CPU Instruction. http://www.securiteam.com/securityreviews/6Z00H20BQS.html. (2004).

[23] Govind Singh Tanwar and Vishal Goar. 2014. Tools, Techniques & Analysis of Botnet. In *Proc. of ICTCS.*

[24] Ritu Tiwari and Abhishek Jain. 2012. Improving network security and design using honeypots. In *Proc. of CUBE International Information Technology Conference.*

[25] Linux Torvalds. 2011. Linux Kernel Source Tree, cpufeature.h. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/include/asm/cpufeature.h. (2011).

[26] Amit Kumar Tyagi and G Aghila. 2011. A wide scale survey on botnet. *International Journal of Computer Applications* 34, 9 (2011), 10–23.

[27] Claud Xiao, Cong Zheng, and Yanhui Jia. 2017. New IoT/Linux Malware Targets DVRs, Forms Botnet. http://researchcenter.paloaltonetworks.com/2017/04/unit42-new-iotlinux-malware-targets-dvrs-forms-botnet/. (2017).

[28] Kui Xu, Patrick Butler, Sudip Saha, and Danfeng Yao. 2013. DNS for massive-scale command and control. *IEEE TDSC* 10, 3 (2013), 143–153.