The Dose Makes the Poison - Leveraging Uncertainty for Effective Malware Detection

Ruimin Sun*, Xiaoyong Yuan†, Andrew Lee¶, Matt Bishop ‖, Donald E. Porter**,
Xiaolin Li§, Andre Gregio†† and Daniela Oliveira‡

University of Florida, USA *gracesrm@ufl.edu, †chbrian@ufl.edu, ‡daniela@ece.ufl.edu, §andyli@ece.ufl.edu
University of North Carolina at Chapel Hill, USA **porter@cs.unc.edu
University of California at Davis, USA ‖mabishop@ucdavis.edu
Federal University of Parana, Brazil ††gregio@inf.ufpr.br
Duke University, USA ¶andrew@cs.duke.edu

***Abstract*—Malware has become sophisticated and organizations don't have a Plan B when standard lines of defense fail. These failures have devastating consequences for organizations, such as sensitive information being exfiltrated.**

A promising avenue for improving the effectiveness of behavioral-based malware detectors is to combine fast (usually not highly accurate) traditional machine learning (ML) detectors with high-accuracy, but time-consuming, deep learning (DL) models. The main idea is to place software receiving borderline classifications by traditional ML methods in an environment where uncertainty is added, while software is analyzed by time-consuming DL models. The goal of uncertainty is to rate-limit actions of potential malware during deep analysis.

In this paper, we describe CHAMELEON, a Linux-based framework that implements this uncertain environment. CHAMELEON offers two environments for its OS processes: standard—for software identified as benign by traditional ML detectors—and uncertain—for software that received borderline classifications analyzed by ML methods. The uncertain environment will bring obstacles to software execution through random perturbations applied probabilistically on selected system calls. We evaluated CHAMELEON with 113 applications from common benchmarks and 100 malware samples for Linux. Our results show that at threshold 10%, intrusive and non-intrusive strategies caused approximately 65% of malware to fail accomplishing their tasks, while approximately 30% of the analyzed benign software to meet with various levels of disruption (crashed or hampered). We also found that I/O-bound software was three times more affected by uncertainty than CPU-bound software.

I. INTRODUCTION

Attacks are continuously evolving and existing protection mechanisms do not cope well with the increased sophistication of attacks, especially advanced persistent threats (APTs), which target organizations. Malware used in APTs attempts to blend in with approved corporate software and traffic, and can act slowly, thus evading detection. As a result, by the time an attack is discovered, sensitive information has already been exfiltrated and many computers have been compromised, making recovery difficult [1], [2].

Real-time malware detection is hard. The industry still relies on antivirus technology for threat detection [3], [4], which is effective for malware with known signatures, but not sustainable given the massive amount of new malware samples released daily. Additionally, since zero-day malware has no known signature, and polymorphic and metamorphic attacks constantly change their patterns, signature scanning operates at a practical detection rate of only 25%–50% [5]. Alternative approaches identify behavioral properties, such as unusual sequences of system calls, and use behavioral patterns to characterize malware. However, research has shown that behavioral-based detectors suffer from a high false-positive rate [6], [7], because of the increasing complexity and diversity of current software. Aggressive heuristics, such as *erring on the side of blocking suspicious software*, can interfere with employee productivity, resulting in employees overriding or circumventing security policies.

Recently, deep learning has achieved state-of-the-art results in a broad spectrum of applications, and has been considered a promising direction for behavioral-based approaches with high detection rates. However, it is unlikely that deep learning methods will be useful in real-time malware detection, because they require considerable computation time for classification when the model needs to be retrained incrementally. Incremental retraining is a common requirement for malware detection, as new variants and samples are regularly discovered and added to the training set. The importance of real-time malware detection is the difference between prevention (discovering malware before some damage is done) and recovery from an attack after the fact.

Thus, there is a frustrating trade-off in malware detection: one can have fast, but less accurate detection using traditional ML methods, or one can use DL for accurate results, but possibly at a much later time. A promising solution to have the best of both approaches is to combine both types of detectors via a spectrum-behavioral operating system (OS). The main idea is as follows. All software in the system starts running in the standard OS environment and is continuously monitored through a behavioral detector. The behavioral detector is based on classical ML algorithms, which provide fast classification and retraining. If a piece of software receives a borderline classification (i.e., reaches a threshold set by the system administrator), it is moved to the uncertain environment. In this environment the software will experience probabilistic and random perturbations, whose severity will depend on whether the software is whitelisted by the organization. The goal of these perturbations is to thwart the actions of potential malware or compromised benign software while deep analysis is underway. If the deep analysis finds the software benign, it is placed back in the standard environment, where it is again continuously monitored.

In this paper, we propose CHAMELEON, a kernel module realizing this spectrum OS behavior for Linux. CHAMELEON has the potential to allow the successful combination of ML detection methods with the power of DL for real-time malware detection to protect computer infrastructures in organizations.

Today, it is standard practice for organizations to restrict or whitelist mission-critical software used by their employees [6]. Employees are supposed to use only approved software tied to their primary task and are allowed to use some personal software. CHAMELEON applies non-intrusive strategies (e.g., delay a system call execution) to whitelisted software, and intrusive strategies (e.g., increase or decrease the bytes in a buffer passed as a parameter to a system call) to non-whitelisted software. The goal of CHAMELEON is to create obstacles to the execution of software running in the uncertain environment, thus buying time for DL-based detectors to provide a definitive and accurate classification of the software.

We evaluated CHAMELEON with 100 samples of Linux malware and 113 common software from several categories. Our results show that at a threshold of 10%, intrusive strategies thwart 62% of malware, while non-intrusive strategies caused a failure rate of 68%. At threshold 50%, the percentage of adversely affected malware increased to 81% and 76% respectively. With a 10% threshold, the perturbations also cause various levels of disruption (crash or hampered execution) to approximately 30% of the analyzed benign software. With a 50% threshold, the percentage of software adversely affected raised to 50%. We also found that I/O-bound software was three times more affected by uncertainty than CPU-bound software.

CHAMELEON has the potential to allow for a combination of traditional ML and DL malware detection, and also provided a "safety net" for failures in standard malware detection solutions. Further CHAMELEON advances systems security, as it can (i) make systems diverse by design because of the unpredictable execution in the uncertain environment, (ii) increase attackers' workload, and (iii) decrease the speed of attacks and their chance of success.

This paper is organized as follows. Section II describes the threat model and assumptions. Section III describes CHAMELEON's design and implementation. Section IV evaluates CHAMELEON. Section V discusses CHAMELEON's results and limitations. Section VI summarizes related work on malware detection, software diversity, and attempts on unpredictability as a security mechanism. Section VII concludes the paper.

II. THREAT MODEL AND ASSUMPTIONS

CHAMELEON's goal is to provide an environment that rate-limits the effects of potential malware, while more time-consuming deep analysis is underway. CHAMELEON's protection is designed for corporations and similar organizations, which adopt a standard practice of controlling software running at the corporate perimeter. These controls commonly apply to mission-critical and task-primary software, as well as allowing some personal software [7]. Organizations face the challenge of enforcing perimeter security, while also causing minimum interference to employees' primary tasks. The combination of fast, preliminary classification by traditional ML methods and deep analysis for borderline cases can help address this challenge.

We also assume that if an organization is a target of a well-motivated attacker, malware will eventually get in. A classic scenario is when a C-level personnel of a targeted organization falls victim to a spear-phishing email attack, thereby causing an APT backdoor to be installed in one of the computers of the victim's company. The malware is zero-day and is not detected by any antivirus. It also behaves in a way that does not raise red flags for a behavior-based detector. Further, a mis-configuration in the administrator's software restriction policies allows the software to run. In a standard OS, this APT would initiate a devastating attack in the organization. With CHAMELEON, this APT might receive a borderline classification at some point by a traditional ML detector and would then be placed in the uncertain environment. In this environment the APT backdoor partially works, while deep analysis makes a more definitive diagnosis.

We assume that a whitelisted software receiving a borderline classification by a traditional ML-based detector can be an indication of software compromise.

It is worth noting that CHAMELEON does not compete with standard lines of defenses, such as antiviruses and traditional behavioral-based detectors. It is complementary, equipping these solutions with a safety net in the event of misdiagnosis.

III. DESIGN AND IMPLEMENTATION

We designed and implemented CHAMELEON for the Linux OS. CHAMELEON offers two environments to its processes: (i) a standard environment, which works predictably as any OS would, and (ii) an uncertain environment, where a subset of the OS system calls undergo unpredictable interferences.

The key insight is that interference in the uncertain environment will hamper the malware's chances of success, as some system calls might return errors in accessing system resources, such as network connections or files. Moreover, random unavailability and some delays will make gaining CPU time difficult for malware.

A. The Interference Set

Our first step was deciding what system calls were good candidates for interference. We relied on Tsai et al.'s study [8], which ranked Linux system calls by their likelihood of use by applications. Based on these insights, we selected 37 system calls for the interference set to represent various OS functionalities relevant for malware (file, network, and process-related). Most of these system calls (summarized in Table I) are I/O-bound, since I/O is essential to most malware, regardless of its sophistication level.

We introduced new versions for all system calls in the interference set. When CHAMELEON's uncertainty module is loaded, it records the pointer to each system call in the interference set as `orig_<syscall_name>` and alters the respective table entry to point to `my_<syscall_name>()`. We also developed two sets of interference strategies, detailed below.

B. Interference Strategies

The **non-intrusive** interference strategies will perturb software execution within the OS specification. They are applied to whitelisted software running in the uncertain environment.

Category	System call
File related	sys_open, sys_openat, sys_creat, sys_read, sys_readv, sys_write, sys_writev, sys_lseek, sys_close, sys_stat, sys_lstat, sys_fstat, sys_stat64, sys_lstat64, sys_fstat64, sys_dup, sys_dup2, sys_dup3, sys_unlink, sys_rename
Network related	sys_bind, sys_listen, sys_connect, sys_accept, sys_accept4, sys_sendto, sys_recvfrom, sys_sendmsg, sys_recvmsg, sys_socketcall
Process related	sys_preadv, sys_pread64, sys_pwritev, sys_pwrite64, sys_fork, sys_clone, sys_nanosleep

TABLE I: System call Interference Set.

System call silencing with error return: The system call immediately returns an error value randomly selected from the range [-255, -1]. This strategy can create difficulties for the execution of the process, especially if it does not handle errors well. Further it can cause transient unavailability to resources, such as files and network connections, creating difficulties for a fork bomb or a network flooder to operate. Note that not all error returns are in the specification; most system calls on Linux have an expected subset, and valid software might fail to check for an unspecified error.

Process delay: Injects a random delay within the range [0,0.1s] during the system call execution with the goal to drag potential malware execution. It can create difficulties in timely malware communication with a C&C for files exfiltration, as well as prevent flooders from sending enough packets in a very short time, rate-limiting DoS in a victim server.

Process priority decrease: Decreases the dynamic process priority to the lowest possible value, delaying its scheduling to one of the system's CPUs.

The **intrusive** interference strategies will cause perturbations that may corrupt the software. They are applied to non-whitelisted software running in the uncertain environment.

System call silencing: The system call immediately returns a value that indicates a successful execution, but without executing the system call.

Buffer bytes change: Decreases the size of the number of bytes in a buffer passed as a parameter to a system call. It can be applied to all system calls with a buffer parameter, such as `sys_read`, `sys_write`, `sys_sendto` and `sys_recvfrom`. This strategy can corrupt the execution of malicious scripts, thus frustrating attempts to exfiltrate sensitive data. Viruses can also be adversely affected by the disruption of the buffer with a malicious payload trying to be injected into a victim's ELF header, and the victim may get corrupted and lose its ability to infect other files.

Connection restriction: Changes the IP address in `sys_bind`, or limits the queue length for established sockets waiting to be accepted in `sys_listen`. The IP address can be randomly changed, which will likely cause an error, or it can be set to the IP address of a honeypot, allowing backdoors to be traced.

File offset change: Changes a file pointer in the `sys_lseek` system call so that subsequent invocations of `sys_write` and `sys_read` will access unpredictable file contents.

C. System Architecture

The uncertain environment adds some fields to the Linux `task_struct` (i.e., thread descriptor):

`process_env`: Informs if the process should run in the standard or uncertain environment.

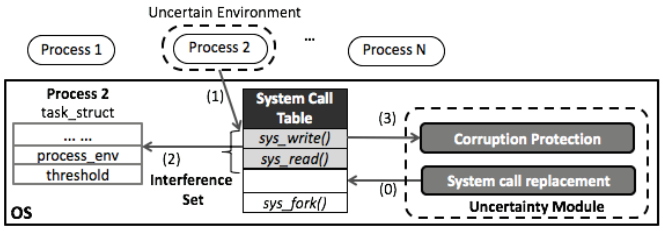

Fig. 1: System architecture. When a process running in the uncertain environment invokes a system call in the interference set (1), the *Uncertainty Module* checks if the process is running in the uncertain environment (2), and depending on the execution of the corruption protection mechanism (3), randomly selects an interference strategy to apply to the system call. The corruption protection mechanism prevents interferences during accesses to critical files, such as libraries.

`fd_list`: Keeps a list of critical file descriptors during runtime execution. Interference on system files, such as library or devices, will likely crash the program execution. Thus, interference is not applied to system calls manipulating those file descriptors (see Section III-D for more details).

`threshold`: Represents the probability that a system call from the interference set invoked by a process in the uncertain environment will undergo interference. The higher the threshold, the higher the probability that an interference strategy will be applied.

Figure 1 illustrates the architecture and operation of the uncertain environment. A key component of the architecture is a loadable kernel module, the *Uncertainty Module*, which monitors the execution of all system calls in the interference set, and applies a randomly-chosen interference strategy to the system call, depending on the process environment and the interference threshold.

For example, consider Process 2 in Figure 1, loaded in the uncertain environment and invoking `sys_write` (Step 1). Because `sys_write` is in the interference set, it can introduce uncertainty in its own execution. First the system call inspects Process 2's environment and finds that it runs in the uncertain environment (Step 2). Next, `sys_write` runs the corruption protection mechanism (Section III-D) to make sure that no interference will occur if the system call is accessing a critical file (Step 3). If `sys_write` is not accessing a critical file, CHAMELEON decides based on the threshold whether or not a strategy should be applied. If a strategy is to be applied, `sys_write` randomly selects one of the strategies that can be applied to its execution.

D. Corruption Protection Mechanism

The uncertainty module employs a corruption protection mechanism to prevent interference while a process in the uncertain environment is accessing critical system files, which might cause early termination of the process. The files are identified through file descriptors, created by `sys_open`, `sys_openat` and `sys_creat`, and deleted by `sys_close`. System calls whose parameters are file descriptors, such as `sys_lseek`, `sys_read` and `sys_write`, are under this protection mechanism. These protected files are determined by an administrator and tracked by setting an extended attribute in the file's inode in the `.security` namespace; a similar strategy is employed by SELinux [9].

When a process running in the uncertain environment opens a file with a pathname beginning with critical directories or containing keywords, the file's descriptor (`fd`) is added to a new per process data structure fd_list. Later, when this process invokes `sys_read` or `sys_write` referring to an fd in fd_list, the protection mechanism will prevent interference strategies from being applied to these system calls.

IV. Evaluation

The goal of our evaluation was to discover the impact of CHAMELEON's uncertain environment in malware and benign softwares behavior. We deployed and evaluated CHAMELEON on a Linux machine running Ubuntu 14.04 with kernel release 3.13, with 16GB RAM, 160GB Hard Disk, x86_64 architecture, and 8 processors.

Our evaluation leveraged a collection of 113 software including common software from GNU projects [10], SPEC CPU2006 [11] and Phoronix-test-suite [12], and 100 Linux malware from THC [13] and VirusShare [14].

The 100 malware samples were randomly selected from different categories (22 flooders, 14 worms, 15 spyware, 24 Trojans and 25 viruses). In total, our evaluation set contained 147 I/O-bound and 66 CPU-bound software samples, containing both common benign software and malware.

For each software or malware sample (and always starting with a clean virtual machine), we configured the system with all files and parameters needed for the evaluation. Then we ran the software (first in the standard environment and later in the uncertain environment) and logged execution-related data, such as the number of invoked system calls, system call parameters, output values, and whether or not the program was adversely affected.

A. General Software

We ran our samples of general software (I/O-bound and CPU-bound) in the uncertain environment and observed their execution outcome. We consider the following cases as *Hampered* executions: (i) a text editor temporarily losing some functionality; (2) a scientific tool producing partial results; (3) a network tool missing packets. The execution outcome was considered *Crashed* if the software hanged longer than twice its standard runtime and needed to be manually killed. A *Succeeded* execution generates outputs that are exactly the same as those produced with the same test case in the standard environment and with a runtime that does not exceed twice that in the standard runtime.

As shown in Table II, at threshold 10% with intrusive strategies, on average 37% of the tasks experienced some form of crash or hamper. With non-intrusive strategies, this percentage was 30%. For a 50% threshold and intrusive strategies, 59% of the software was adversely affected. With non-intrusive strategies, this number was 10% smaller. Programs that wrote into files were the most sensitive to uncertainty as all of the text editors (for 50% threshold) and half of them (for 10% threshold) were adversely affected by uncertainty.

Software that relies on the correctness of the data written into files, such as text editors, were the most sensitive to uncertainty. With intrusive strategies at threshold 50%, none of the 15 text editors running in the uncertain experiment completed their intended tasks. With non-intrusive strategies, only 33% of the software executed without errors. With a threshold of 10%, non-intrusive strategies obtained a Succeed ratio of 73.3%. With intrusive strategies for the same threshold, nearly half of the software crashed or were hampered. These results show that APTs that rely on exfiltrating sensitive data and keyloggers would have a very low success rate in the uncertain environment. This result also shows that text-editors are not suitable to be protected under CHAMELEON.

We analyzed the performance penalty caused by the interference strategies, such as process delay and process priority decrease on all 23 benchmark software whose execution could be scripted. Highly interactive software were tested manually and showed negligible overhead. Figure 2 shows the average runtime overhead for software whose execution could be scripted running in the uncertain environment. For runtimes ranging from 0 to 0.01 seconds, the average penalty is 8%; for runtimes ranging from 0.1 to 1 seconds, the average penalty is 4%; for runtimes longer than 10 seconds, the average penalty is 1.8%. This shows that the longer the runtime, the smaller the overhead is. One hypothesis is that software with longer execution time are usually CPU-bound programs performing time-consuming calculations. Because most of the system calls in the interference set are I/O related, CPU-bound programs are perturbed less and thus smaller overhead are incurred.

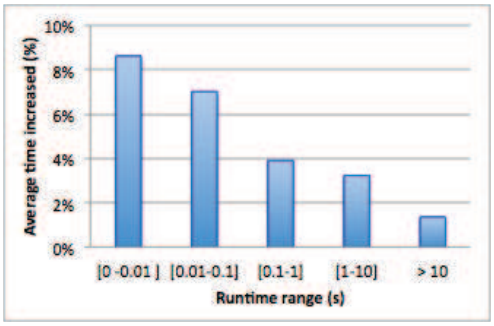

Fig. 2: Performance penalty for 23 benchmark software whose execution time could be scripted. We categorized the software according to their average runtime.

During our analysis we also measured the code coverage by compiling the analyzed software source code with gcov [15], EMMA [16] and Coverage.py [17] based on the software's programming language. On the 71 applications for which we had source code, the average coverage was 69.49%.

We also tested 26 benign applications with different workloads running in the standard and uncertain environment. The workloads contained three levels: *light*, *medium* and *heavy*, which corresponded to *test*, *train*, and *ref* level for SPEC CPU2006, and first, middle-most, and last-level in the Phoronix Test Suite. Our results showed that 2 of the 26 benign software were adversely affected on all three different workloads, indicating that the workload type does not impact the program outcome in the uncertain environment for the two sets of interference strategies we used.

B. Malware

We also analyzed how malware were affected by the uncertain environment, for intrusive and non-intrusive strategies. We considered that malware were adversely affected by the uncertain environment if they crashed or executed in a

hampered fashion. An execution is considered *Crashed* if malware terminates before performing its malicious actions. An execution is considered *Succeeded* if malware accomplished its intended task, such as injecting malicious payload into an executable. The following outcomes are examples of hampered malware execution in the uncertain environment: (1) a virus that injects only part of the malicious code to an executable or source code file; (2) a botnet that loses commands sent to the bot herder; (3) a cracker that retrieves wrong or partial user credentials; (4) a spyware that redirects incomplete stdin, stdout or stderr of the victim; (5) a flooder that sends only a percentage of the total number of packets it attempted.

Our evaluation with 100 Linux malware samples showed that, when intrusive strategies were applied, 81% of the malware samples failed to accomplish their tasks at threshold 50%, and 62% failed at threshold 10%. Non-intrusive strategies yielded similar results for threshold 50% and 10%, with 76% and 68% of malware adversely affected, respectively.

Table II shows the impact of intrusive and non-intrusive strategies in the uncertain environment for different types of malware for the thresholds of 10% and 50%. For each type of malware samples, threshold 50% caused about 20% more Crashed and 13% fewer Succeeded outcomes in the uncertain environment than threshold 10% for both intrusive and non-intrusive strategies. Non-intrusive strategies caused about 9% more Hampered and 5% fewer Crashed execution outcomes than intrusive strategies for different thresholds, showing as expected that non-intrusive strategies are less disturbing to malware than intrusive strategies.

At threshold 50%, intrusive and non-intrusive strategies had a similar ratio of Crashed execution: 40% to 60% for each type of malware. At threshold 10%, non-intrusive strategies had spyware with the lowest ratio (13%) of Crashed execution and flooders with the highest (36%); Intrusive strategies had flooders with the lowest ratio (13%) of Crashed execution and viruses with the highest (44%). In other words, since threshold 50% had a strong influence on malware execution in the uncertain environment, different sets of strategies (intrusive or non-intrusive) caused similar impact on execution outcomes.

The studied viruses appeared not to be sensitive to strategy and threshold, with the ratios of different execution outcomes similar for all cases—Crashed ratio at about 40%, Succeeded ratio at about 25%, and Hampered ratio at about 30%. This can be explained by the homogeneous nature of these viruses' implementation. Usually a virus will inject malicious code to executables or source files in one `write` system call, and once this call is disturbed, the virus will be affected. Therefore, the Crashed ratio for viruses is higher than those of other types of malware.

At threshold 10%, Flooders were more sensitive to non-intrusive strategies than intrusive strategies, with the ratio of Succeeded execution decreased from 41% to 18%, while other types of malware did not change so much. Since the goal of Flooders is to send a great number of packets in a very short time, the advantages of non-intrusive strategies such as delaying and decreasing process priorities affected Flooders more.

Spyware had the lowest ratio of Hampered execution for different thresholds and strategies.

C. Behavior Comparison

Table III compares the execution of malware and benign software at the system call level in the uncertain environment. Modern software invoked more than twice the number of system calls monitored than malware, even with the existence of Flooders, which usually largely increased the average number of system calls invoked. For benign software the number of system calls perturbed or silenced was only half of those for malware, mainly because of the effectiveness of the corruption protection mechanism introduced in Section III-D. Benign software had a larger number of connection attempts and read/write operation monitored than malware. However, a smaller fraction of benign software system calls was perturbed in the uncertain environment.

D. Case Study: Advanced Persistent Threat (APT)

In this section we show the evaluation of the interference strategies with an APT attack. We simulated a watering hole attack similar to *the Black Vine* APT from Symantec [18]. This attack has three main components: a Trojan, a backdoor and a keylogger. First, the attacker sends a spear-phishing e-mail to a user with a link for downloading the Trojan encryption tool. If the user clicks on the link and later uses the Trojan tool to encrypt a file, the tool downloads and executes a backdoor from a C&C server while encrypting the requested file. Then, the backdoor copies the directory structure and the ssh host key from the user's machine into a file and sends it to the C&C server. After the backdoor executes, the attacker deletes any traces of the infection without affecting the Trojan's encryption/decryption functionality. The attacker will also install a keylogger to obtain root privileges. Next, the backdoor runs a script that uploads sensitive data to the C&C server.

The Trojan is written in C using *libgcrypt* for encryption and decryption. It uses the curl library for downloading the backdoor from the Internet. In our simulation we used the *logkeys* keylogger from [19]. The backdoor script uses scp for sending the data to the C&C server.

APT in the Uncertain Environment: in the standard environment, only 85 system calls in the interference set were captured for the simulated Black Vine, in which 52 were connection related calls with 132,254 buffer bytes read or written. Compared to the standard environment, the uncertain environment with a threshold of 10% caused a loss of 4% and 5% of the system calls with intrusive and non-intrusive strategies, respectively. With a 50% threshold, 71% and 76% of the system calls were lost, respectively. This sharp increase on system call loss was caused by early termination of some APT functionalities, e.g., a failure of `sys_open` would prevent the invocation of `sys_read` and `sys_write`. Threshold 50% also caused a great increase on connection loss and byte loss, compared with the numbers we found for threshold 10%. With intrusive strategies at threshold 10%, the number of bytes monitored increased compared with that in the standard environment, and this can be explained by the APT retry-on-fail mechanism. When an incomplete write is detected, the APT would try to write the buffer bytes again. When the threshold increased from 10% to 50%, the retry mechanism failed to write the original number of bytes.

	threshold = 50%						threshold = 10%					
Malware	Intrusive			Non-intrusive			Intrusive			Non-intrusive		
Category	Succ	Crash	Hamper	Succ	Crash	Hamper	Succ	Crash	Hamper	Succ	Crash	Hamper
Spyware	27%	67%	7%	40%	40%	20%	53%	27%	20%	60%	13%	27%
Viruses	24%	48%	28%	24%	40%	36%	24%	44%	32%	28%	36%	36%
Worm	21%	36%	43%	21%	57%	21%	29%	50%	21%	21%	29%	50%
Trojan	17%	63%	21%	29%	50%	21%	46%	29%	25%	38%	25%	38%
Flooders	9%	59%	32%	9%	50%	41%	41%	14%	45%	18%	36%	45%
All	19%	55%	26%	24%	45%	29%	38%	32%	30%	32%	29%	39%
Software	Intrusive			Non-intrusive			Intrusive			Non-intrusive		
Category	Succ	Crash	Hamper	Succ	Crash	Hamper	Succ	Crash	Hamper	Succ	Crash	Hamper
Text Editors	0	73%	27%	33%	33%	33%	53%	20%	27%	73%	20%	7%
Compilers	18%	55%	27%	36%	18%	45%	55%	27%	18%	73%	18%	9%
Network Tools	38%	56%	6%	50%	31%	19%	56%	19%	25%	56%	31%	13%
Scientific Tools	33%	63%	3%	40%	43%	17%	53%	30%	17%	60%	20%	20%
Others	82%	7%	11%	79%	7%	14%	86%	7%	7%	86%	4%	11%
All	41%	47%	12%	51%	27%	22%	63%	20%	17%	70%	17%	13%

TABLE II: Execution for different types of malware and benign software under intrusive and non-intrusive strategies in the uncertain environment. We used *Succ*, *Crash* and *Hamper* for the *ratios* of Succeeded, Crashed and Hampered execution outcomes.

Malware Category	Number of syscalls monitored	Percentage of syscalls perturbed	Percentage of syscalls silenced	Number of connection-related syscalls monitored	Percentage of connection-related syscalls perturbed	Number of buffer-related syscalls monitored	Percentage of buffer-related syscalls perturbed
Flooders	930.50	9.74%	3.39%	626.64	10.13%	91.18	6.58%
Spyware	50.73	2.89%	1.05%	4.67	7.14%	26.13	3.06%
Trojan	523.80	8.09%	2.85%	396.53	9.52%	182.67	7.14%
Viruses	423.44	5.02%	1.66%	211.75	9.56%	15.32	4.96%
Worms	68880.64	0.05%	0.02%	332.56	9.86%	43.79	8.97%
All	9992.49	0.41%	0.14%	266.16	9.87%	77.78	6.83%
Software Category	Number of syscalls monitored	Percentage of syscalls perturbed	Percentage of syscalls silenced	Number of connection-related syscalls monitored	Percentage of connection-related syscalls perturbed	Number of buffer-related syscalls monitored	Percentage of buffer-related syscalls perturbed
Scientific Tools	2071.59	1.13%	0.34%	4.00	0.00%	1878.28	0.46%
Compilers	167303.36	0.04%	0.01%	4.00	0.00%	164418.73	0.00%
Network Tools	515.50	2.85%	1.01%	106.43	10.99%	206.38	1.54%
Others	566.31	0.54%	0.19%	2.00	0.00%	394.77	0.19%
Text Editors	6693.20	0.42%	0.14%	3404.00	0.04%	4377.93	0.40%
All	20863.74	0.10%	0.03%	1283.64	0.40%	20023.82	0.03%

TABLE III: Comparison between malware and benchmark software on system call perturbation under the uncertain environment (with Non-intrusive strategies at threshold 10%).

Environment	Number of syscalls monitored	Percentage of syscalls lost	Number of connection-related syscalls monitored	Percentage of connection-related syscalls lost	Number of buffer-related bytes monitored	Percentage of buffer-related bytes lost
None	85	0%	52	0%	132254	0%
Intrusive (10%)	82	4%	49	6%	145200	9%
Intrusive (50%)	25	71%	16	69%	5905	96%
Non-intrusive (10%)	81	5%	48	8%	120366	9%
Non-intrusive (50%)	20	76%	11	79%	5963	95%

TABLE IV: Execution details of the Black Vine APT in the standard and uncertain environment. Under the uncertain environment, the experiment was carried out with intrusive and non-intrusive strategies for the 10% and 50% thresholds.

V. Discussion

As we discussed in Section II, a resourceful and motivated adversary can bypass any protection mechanism. Even though the uncertain environment is designed to rate-limit stealthy malware, it can still be eluded by attacks. For example, highly fault-tolerant malware will be resilient to the uncertain environment.

There are some trade-offs in selecting an interference strategy. Intrusive strategies are more aggressive, and will affect software running in the uncertain environment more. For an organization with high security demands and less tolerance for non-approved software, intrusive strategies will offer more protection. However, our approach is not suitable for organizations that do not control software running in their perimeter.

Strategy *Process Delay* is different from just suspending software execution. A suspended execution stops suspicious software from running and will not generate data for DL analysis—this does not address the challenge of false positive. *Process Delay*, on the other hand, slows down software execution, thus potentially buying time for deep analysis and allowing for the accurate classification of borderline cases. Also, suspension of execution can be detected by malware just by checking wall clock time. If malware comes to this realization, it can infer it is being monitored and avoid behaving maliciously for some time to avoid detection.

The worst case scenario for software in Chameleon would be to keep getting a borderline classification from ML detectors, and end up running in the uncertain environment all the time. One possibility to address such corner cases is for the ML detector raise the borderline threshold or the system administrator change the uncertainty level.

Although this approach was implemented for Linux (to allow the release of the open source code), it can also be

implemented in other operating systems, such as Windows, which is a popular target of malware attacks.

Finally, we are aware that the degree of uncertainty is not a one-size-fits-all solution—we expect an administrator to dial in the level of uncertainty to the needs of the organization and applications.

VI. Related Work

Our work intersects the areas of malware detection, software diversity, and deception. This section summarizes how they have been used in software design and highlights understudied areas.

Malware Detection: There are extensive literature dating to the 1990s on detection of intrusions and malware. Malware detection techniques can be signature-based [3], [4] or behavior-based [20]–[22].

Signature-based approaches match bytes and instructions from known malware to the unknown program under analysis. These techniques are accurate, but they can be evaded when attackers use polymorphism and metamorphism to create malware variants; these variants have the same behavior but have different byte signatures. Further, these approaches cannot detect zero-day malware and have a practical detection rate ranging from 25% to 50% [5].

Behavior-based techniques, which can be static or dynamic, analyze program behavior and attempt to detect events, instructions or resource access that are indicative of malware. Behavioral solutions based on static analysis [21] analyze the source code of malware and benign applications in an attempt to extract their unique behavior in high level specifications. Most of the work on dynamic behavior-based malware detection [20], [22] are based on seminal work by Forrest et al. [20]. System call-based malware detectors suffer, however, from high positive rates due to the diverse nature of system calls invoked by applications. This challenge has worsened as programs are becoming increasingly diverse [22].

Some approaches analyze the data flow of a program to extract malware behavior. Panorama [23], for example, performs system-level taint-tracking to discover how malware leaks sensitive data. Martignoni et al. [24] leveraged hierarchical behavioral graphs to infer high-level behavior of low-level events. The approach traces the execution of a program, performing data-flow analysis to discover relevant actions such as proxying, data leaking and key stroke logging. Ether [25] improved on tracing granularity on single instructions and system calls via hardware virtualization extensions. Ye et al. [26] proposed a semi-parametric classification model for combining file content and file relation information to improve the performance of file sample classification. More recently, Bromium [5] proposed the use of virtualization on a per-process basis to isolate every process from the system and from each other. While this certainly advances the level of granularity offered by traditional sandboxes, it has some inconveniences for the user (e.g., it creates obstacles to inter-process communication) and cannot guarantee complete perimeter protection (e.g., a keylogger still can record credentials).

CHAMELEON's goal is to provide an environment where possible malware can be rate-limited, while time-consuming deep analysis is underway.

Diversity: The ability to diversify behavior within a system is an essential building block for unpredictability. Diversifying components within the software stack can improve overall robustness. Researchers have studied building diverse computer systems. Forrest et al. [27] proposed guidelines and advocated the use of randomized compilation techniques, which motivated later work in this area [28]. She and her colleagues [29] also showed that code exhibits evolutionary characteristics similar to those seen in the biological world. A program, like a biological organism, has the potential to mutate, but can still function normally [29].

Several projects mitigate buffer overflows and other memory errors by randomizing system call mappings, global library entry points, stack placement, stack direction, and heap placement—often in conjunction with running multiple versions in parallel to detect divergence [30].

Deception: To a limited extent, deception has been an implicit technique for cyber warfare and defense, but is understudied as a fundamental abstraction for secure systems. Honeypots and honeynets [31] are systems designed to look like production systems in order to deceive intruders into attacking the systems or networks so that the defenders can learn new techniques. Several technologies for providing deception have been studied. Software decoys are agents that protect objects from unauthorized access [32]. The goal is to create a belief in the attacker's mind that the defended systems are not worth attacking or that the attack was successful. Red-teaming experiments at Sandia tested the effectiveness of network deception on attackers working in groups [33]. The deception mechanisms at the network level successfully delayed attackers for a few hours. Almeshekah and Spafford [34] further investigated the adversaries' biases and proposed a model to integrate deception-based mechanisms in computer systems. In all these cases, the fictional systems are predictable to some degree; they act as real systems given the attacker's inputs. True unpredictability requires randomness at a level that would cause the attacker to collect inconsistent results. This observation leads to the notion of *inconsistent deception* [35], a model of deception that challenges the cornerstone of projecting false reality with internal consistency. Sun et al. [36], [37] also argued for the value of unpredictability and deception as OS features.

In this paper we explored non-corruptive unpredictable interferences to create an uncertain environment for software being deep analyzed after an initial borderline classification.

VII. Conclusions

In this work we presented CHAMELEON, a novel Linux framework that introduced uncertainty as an OS built-in feature to rate-limit the execution of possible malware that received a borderline classification by traditional ML detectors, while a second performance expensive deep-learning detector is operating. CHAMELEON's protection target are organizations, where it is a common practice to whitelist software to run in the organization perimeter. CHAMELEON offers two environments for software running in the system: (i) standard, which works according to the OS specification and (ii) uncertain, for any software that receives a borderline classification by traditional ML-based detectors. In the uncertain environment software experiences a set of perturbations, which create obstacles for their execution, while deep-learning analysis is underway.

We evaluated CHAMELEON with 113 common applications and a set of 100 malware samples for Linux from various categories. Our results showed that a threshold of 10% caused various levels of disruption to 30% of the analyzed software. Malware was affected more with intrusive strategies caused 62% and non-intrusive strategies caused 68% of the malware to fail to accomplish their tasks.

Besides effectively supporting the combination of the best of traditional ML and emerging DL methods and providing a "safety net" for failures of standard intrusion detection systems, CHAMELEON improves system security through (i) making systems diverse by design, (ii) increasing attackers' work factor, and (iii) decreasing the success probability and speed of attacks. The idea of making systems less predictable is audacious, nonetheless, our results indicate that an uncertain system can be feasible for raising an effective barrier against sophisticated and stealthy malware. The degree of uncertainty is not a one-size-fits-all solution—we expect an administrator to dial in the level of uncertainty to the needs of the organization and applications. Finally, we define success of software execution in the uncertain environment as benign software tolerating uncertainty and users obtaining useful results from benign software in the system.

VIII. ACKNOWLEDGMENT

We thank the DSC reviewers for their insightful comments. This research is supported by NSF grant CNS-1464801, CNS-1228839, CNS-1161541, DGE-1303211, ACI-1229576, CNS-1624782, VMWare and Florida Cyber Security grants.

REFERENCES

[1] T. Wrightson, *Advanced Persistent Threat Hacking: The Art and Science of Hacking Any Organization*, 1st ed. McGraw-Hill Education, 2014.

[2] "Email Attacks: This Time It's Personal." [Online]. Available: http://itknowledgeexchange.techtarget.com/security-detail/cisco-report-email-attacks-this-time-its-personal/

[3] S. Kumar and E. H. Spafford, "An application of pattern matching in intrusion detection," 1994.

[4] G. Vigna and R. A. Kemmerer, "Netstat: A network-based intrusion detection approach," ser. ACSAC '98, 1998.

[5] "Bromium end point protection." [Online]. Available: https://www.bromium.com/

[6] "Modern malware exposed." [Online]. Available: http://www.nle.com/literature/FireEye_modern_malware_exposed.pdf

[7] "The modern malware review." [Online]. Available: http://media.paloaltonetworks.com/documents/The-Modern-Malware-Review-March-2013.pdf

[8] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter, "A study of modern linux api usage and compatibility: What to support when you're supporting," in *Eurosys*, 2016.

[9] "Security-enhanced linux." [Online]. Available: http://www.nsa.gov/research/selinux/

[10] "Gnu project." [Online]. Available: http://www.gnu.org/software/software.html

[11] "SPEC CPU 2006 https://www.spec.org/cpu2006/."

[12] "The phoronix test suite http://www.phoronix-test-suite.com/."

[13] "THC: the hacker's choice." [Online]. Available: https://www.thc.org/

[14] "Virusshare." [Online]. Available: https://virusshare.com/

[15] "Gcov." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[16] "EMMA: a free java code coverage tool." [Online]. Available: http://emma.sourceforge.net/

[17] "Coverage.py https://github.com/msabramo/coverage.py."

[18] "The black vine cyberespionage group." [Online]. Available: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-black-vine-cyberespionage-group.pdf

[19] "Logkeys ubuntu." [Online]. Available: http://packages.ubuntu.com/precise/admin/logkeys

[20] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff., "A sense of self for Unix processes," in *Proceedings of the IEEE Symposium on Security and Privacy*, 1996, pp. 120–128.

[21] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, ser. SP '05, 2005.

[22] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: Using system-centric models for malware protection," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10, 2010, pp. 399–412.

[23] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," *ACM CCS 07*, pp. 116–127, November 2007.

[24] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A layered architecture for detecting malicious behaviors," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID '08, 2008, pp. 78–97.

[25] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 51–62. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455779

[26] Y. Ye, T. Li, S. Zhu, W. Zhuang, E. Tas, U. Gupta, and M. Abdulhayoglu, "Combining file content and file relations for cloud based malware detection," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '11. New York, NY, USA: ACM, 2011, pp. 222–230. [Online]. Available: http://doi.acm.org/10.1145/2020408.2020448

[27] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[28] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *IEEE Security and Privacy Symposium*, 2014, pp. 276–291.

[29] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *Genetic Programmable and Evolvable Machines*, vol. 15, no. 3, 2014.

[30] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," UC, Berkeley, Tech. Rep., 2002.

[31] L. Spitzner, *Honeypots: Tracking Hackers*. Addison Wesley Reading.

[32] N. R. J. Michael, M. Auguston, D. Drusinsky, H. Rothstein, and T. Wingfield, "Phase II Report on Intelligent Software Decoys: Counterintelligence and Security Countermeasures ," *Technical Report, Naval Postgraduate School, Monterey, CA*, 2004.

[33] F. Cohen, I. Marin, J. Sappington, C. Stewart, and E. Thomas, " Red Teaming Experiments with Deception Technologies ," *IA Newsletter*, 2001. [Online]. Available: \url{http://all.net/journal/deception/experiments/experiments.html}

[34] M. H. Almeshekah and E. H. Spafford, "Planning and integrating deception into computer security defenses," in *New Security Paradigms Workshop (NSPW)*, 2014.

[35] V. Neagoe and M. Bishop, "Inconsistency in deception for defense," in *New Security Paradigms Workshop (NSPW)*, 2007, pp. 31–38.

[36] R. Sun, D. E. Porter, D. Oliveira, and M. Bishop, "The case for less preditable operating system behavior," in *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[37] R. Sun, A. Lee, A. Chen, D. E. Porter, M. Bishop, and D. Oliveira, "Bear: A framework for understanding application sensitivity to os (mis) behavior," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE, 2016, pp. 388–399.