Leveraging Uncertainty for Effective Malware Mitigation and Software Resilience Improvement

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.

With this paper, we aim to leverage uncertainties to rate-limit actions of potential malware during deep analysis. We improve CHAMELEON, an existing Linux-based framework that implements this uncertain environment. Specifically, we design and implement a dynamically changed interference threshold to perturb malware execution, evaluate the results comparing with a static threshold, and analyze on benign software adversely affected by CHAMELEON. Results showed that CHAMELEON with a dynamic threshold caused 92% of the malware to fail to accomplish their tasks, and 10% of the benign software to meet with various levels of disruptions. Compared with previous work with a static threshold [1], CHAMELEON made an improvement with 20% more benign software succeeded and 24% more malware crashed or hampered in the uncertain environment. CHAMELEON was also capable of reproducing 7 bugs in vim, tar, Mozilla Firefox and Thunderbird), from the crashes of these software undergoing non-intrusive interference strategies.

1. 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 [2, 3].

Real-time malware detection is hard. The industry still relies on antivirus technology for threat detection [4, 5], 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% [6]. 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 [7, 8], 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 em-
ployees 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.

The goal of CHAMELEON is to create obstacles to the execution of software running in the uncertain environment, thus saving time for DL-based detectors to provide a definitive and accurate classification of the software. 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. 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.

In this paper, we improve our work described in CHAMELEON [1], and present the following contributions.

- We design and implement a dynamically changed perturbation threshold based on the behavior of software execution.

- We show that a dynamic threshold will better protect the system by bringing more adverse effects to malware execution and less adverse effect to benign software execution, compared with a static threshold.

- We dig into the reasons causing benign software in the uncertain environment to crash, and found 7 bugs from vim, tar, Mozilla Firefox and Thunderbird reported before. We argue that resilient benign software will be less affected by the uncertain environment.

2. Threat

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 [8]. 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 antivirus and traditional behavioral-based detectors. It is complementary, equipping these solutions with a safety net in the event of misdiagnosis.

### 3. 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.

#### 3.1. The Interference Set

Our first step was deciding what system calls were good candidates for interference. We relied on Tsai et al.’s study [9], 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 1) 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 based on above behaviors. The first set, non-intrusive, perturbs software execution within the OS specification. The second set, intrusive, causes corruptive perturbations.

#### 3.2. Interference Strategies

We considered the following non-intrusive interference strategies, and they are applied to whitelisted software running in the uncertain environment. **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

<table>
<thead>
<tr>
<th>Category</th>
<th>System call</th>
</tr>
</thead>
<tbody>
<tr>
<td>File related</td>
<td><code>sys_open</code>, <code>sys_openat</code>, <code>sys_creat</code>, <code>sys_read</code>, <code>sys_readv</code>, <code>sys_write</code>, <code>sys_writev</code>, <code>sys_lseek</code>, <code>sys_close</code>, <code>sys_stat</code>, <code>sys_lstat</code>, <code>sys_fstat</code>, <code>sys_stat64</code>, <code>sys_lstat64</code>, <code>sys_fstat64</code>, <code>sys_dup</code>, <code>sys_dup2</code>, <code>sys_dup3</code>, <code>sys_unlink</code>, <code>sys_rename</code></td>
</tr>
<tr>
<td>Network related</td>
<td><code>sys_bind</code>, <code>sys_listen</code>, <code>sys_connect</code>, <code>sys_accept</code>, <code>sys_accept4</code>, <code>sys_sendto</code>, <code>sys_recvfrom</code>, <code>sys_sendmsg</code>, <code>sys_recvmmsg</code>, <code>sys_socketcall</code></td>
</tr>
<tr>
<td>Process related</td>
<td><code>sys_preadv</code>, <code>sys_pread64</code>, <code>sys_pwritev</code>, <code>sys_pwrite64</code>, <code>sys_fork</code>, <code>sys_clone</code>, <code>sys_nanosleep</code></td>
</tr>
</tbody>
</table>

Table 1: System call Interference Set.
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. This strategy can hamper malware execution, buying time for a definitive detection.

The intrusive interference strategies 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.

### 3.3. System Architecture

The uncertain environment adds some fields to the Linux `task_struct`.

![Figure 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.](image)

- **process_env**: Informs if the process should run in the standard or uncertain environment.
- **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 3.5 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, \texttt{sys write} runs the corruption protection mechanism (Section 3.5) to make sure that no interference will occur if the system call is accessing a critical file (Step 3). If \texttt{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, \texttt{sys write} randomly selects one of the strategies that can be applied to its execution.

3.4. Per-System Call Interference Threshold

In the uncertain environment, before the OS executes a system call from the interference set, it will compute the threshold that will determine the probability that this system call will undergo interference.

The goal of dynamic per-system call interference threshold is to effectively interfere with behaviors that are more relevant to malware than to benign software. To find these behaviors, we manually measured 183 Linux malware and 100 common benign software in a virtual machine and found the following patterns.

1. Invocation of one type of system call or a pattern of several system calls very frequently, such as \texttt{sys open()}, \texttt{sys fork()} and \texttt{sys sendto()};
2. Writing to ELF executable headers;
3. Redirection of the system standard input, output or error;
4. Renaming or unlinking of system binaries.

Behavior (1) generalizes how many types of malware operate. This behavior can represent a flooder sending millions of packets to block a server, a botnet trying to scan victim IPs and report back to the C&C server, a password cracker attempting to brute-force a ssh session key, or a fork bomb trying to use up system resources. Behavior (2) is common for viruses trying to inject themselves into other benign executables or source code files. Behavior (3) is common for malware opening a backdoor or a reverse shell, a crucial step for the operations of C&C servers. Behavior (4) is common for malware replacing system files with Trojans. All in all, malware lifeblood are I/O operations and they eventually depend on one or a combination of the behaviors described above to perform their primary malicious tasks.

Benign software, unless under debugging and configuration modes, are less likely to show such behaviors, especially clusters of them. Therefore, interference on system calls relevant to these types of behaviors should disrupt malware more than benign software.

The algorithm below summarized how we change the threshold based on these behaviors. All processes loaded in the uncertain environment start with a default threshold \( t_d \). During a process execution, we dynamically adjust the threshold upon system call invoking behavior. For Behavior (1), given that different programs invoke system calls at different speed and quantity, we compute “frequent invocation” from the ratio of a system call pattern occupying the total number of system calls.

\[
\text{ratio} = \frac{\text{Count syscall}}{\text{Count totalsyscall}}
\]

If \( \text{ratio} \) is larger than \( r \) (a customized number), the system call pattern is considered “frequent invocation”. To alleviate the problem of too frequent invocation, the threshold and the ratio should be in a proportional relationship \[10\]—the more frequent the invocation is, the higher threshold the interference should be. We use \( P \times \text{ratio} \) to denote the threshold, in which \( P \) represents how fast the threshold changes. In any case, the threshold will be smaller than \( t_{\text{max}} \).

If a process exhibits any of the behaviors from 2-4, the threshold will be adjusted to \( t_{\text{max}} \). Examples of a process’ actions that can be classified as behaviors 2-4 are: \texttt{sys write("\\177ELF")} (Behavior 2), \texttt{sys dup(0)}, \texttt{sys dup(1)} (Behavior 3), and \texttt{sys unlink("bin")} or \texttt{sys rename("bin")} (Behavior 4). In our study we considered \( r = 70\% \), \( t_d = 10\% \), and \( t_{\text{max}} = 95\% \). \( t_{\text{max}} \) was not chosen as 100\%, because we would not want processes falsely transferred to the uncertain environment to stop running.

\[1\] developers performing debugging tasks or system administrators could have a different customization for the uncertain environment.
$P$ is a value in $[0.2, 1.2]$, so that $P \times \text{ratio}$ can lie in $[t_d, t_{\text{max}}]$.

In the beginning of a process execution, it is very likely that \text{ratio} is greater than $r$. Thus, we only start adjusting the threshold when the total number of system calls invoked by the process is greater than 100.

3.5. 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 \text{sys.open}, \text{sys.openat} and \text{sys.create}, and deleted by \text{sys.close}. System calls whose parameters are file descriptors, such as \text{sys.lseek}, \text{sys.read} and \text{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 [11].

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 \text{sys.read} or \text{sys.write} referring to an $fd$ in $fd_list$, the protection mechanism will prevent interference strategies from being applied to these system calls.

Algorithm 1 shows how the OS applies the interference strategies on \text{sys.write}. First, the following conditions are checked: (i) the process is running in the standard environment ($\text{process.env} == 0$), and (ii) the targeted file descriptor is a critical system file (see Section 3.5). If any of these conditions are true the system call runs normally. Otherwise, the system call updates its execution counters in the current process (i.e. the total number of system calls invoked $\text{current.total.syscall}$ and the total number of \text{sys.write} invoked $\text{current.counter}[\text{WRITE}]$) and computes a threshold. Then, the algorithm generates a random number in the range $[0,1]$, and if the number is smaller than the threshold, the system call undergoes interference.

The algorithm will randomly selects one of the interference strategies: System call silencing with error return, Process delay, or Process priority decrease. In this case, only non-intrusive strategies are considered. If \text{sys.write} is silenced, a random error code is returned, so that the process knows that an error occurred. If Process delay is chosen, the algorithm randomly selects a delay for the system call execution in the range $[0, 0.18]$. If Process priority decrease is selected, the algorithm decreases the process priority to the minimum.

Algorithm 1: Applying interferences to \text{sys.write()}

Function long my.sys.write(fd, buf, size)

if $\text{process.env} == 0$ or \text{corruption.protection(sys.write, pid)} then

| return orig.sys.write(fd, buf, size); |

else

| current.total.syscall++; |
| current.counter[WRITE]++; |
| updateThreshold(); |
| if $(\text{random}(0,1.0) > \text{threshold})$ then |
| return orig.sys.write(fd, buf, size); |

strategy $= \text{random}(1,3);$ 

if strategy $= 1$ then

| /* Fail system call */ |
| return random(-255, -1); |

else if strategy $= 2$ then

| /* Delay process */ |
| delay(random(0, MAX_DELAY)); |
| return orig.sys.write(fd, buf, size); |

else

| /* Process priority reduction */ |
| decrease_current_priority(); |
| return orig.sys.write(fd, buf, size); |

end

end

4. Evaluation

The goal of our evaluation was to discover the impact of CHAMELEON’s uncertain environment in malware and benign software behavior. We considered security, performance, and software behavior to answer the following research questions: (i) are systems employing a dynamic threshold better protected than a static threshold? (ii) how different strategies impact the malware in the uncertain environment? and
threshold = \begin{cases} 
  t_{max}, & P \times \text{ratio} \geq t_{max} \& \text{Count}_{\text{syscall}} > 100 \\
  P \times \text{ratio}, & \text{ratio} > r \& P \times \text{ratio} > t_{max} \& \text{Count}_{\text{syscall}} > 100 \\
  t_d, & \text{Behavior}(2-4) \\
  \text{otherwise} 
\end{cases}

(iii) how benign software can be more resilient in the uncertain environment?

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 [12], SPEC CPU2006 [13] and Phoronix-test-suite [14], and 100 Linux malware from THC [15] and VirusShare [16].

The 100 malware samples were randomly selected from different categories (22 flooders, 14 worms, 15 spyware, 24 Trojans and 25 viruses) detailed in the Appendix. 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, we configured the system with all files and parameters needed for the evaluation, in a clean virtual machine. Then we ran the software first in the standard environment, then 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.

4.1. Security

The goal of our security evaluation was to analyze how the unfavorable environment with a dynamic threshold protected the system against several types of malware. We compared the results with those in previous work [11] using a static interference threshold. We considered that malware were adversely affected by the uncertain environment if they crashed or executed in a hampered fashion. 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.

We evaluated the dynamic threshold ($P = 1.2$) and found that 92 out of the 100 studied Linux malware failed to accomplish their tasks, including 23 malware crashed and 69 experienced hampered execution. Compared with previous work using a static threshold (10%), we made improvements with 24 fewer malware succeeded the infection, and 30 more malware hampered during execution (see Figure 2).

![Figure 2: Comparison between static and dynamic threshold in affecting malware in the uncertain environment.](image-url)
threshold changes from \( t_d \) to \( t_{\text{max}} \) while a static threshold remains constantly at \( t_d \). On all the system calls invoked, flooders have the highest percentage perturbed and worms have the lowest percentage perturbed, for both static and dynamic threshold. Based on previous work, this can be explained by the fact that flooders invoked most system calls in the interference set, while worms invoked least. On the percentage of connection-related calls perturbed, spyware invoked 0% with a dynamic threshold. For spyware, most of the sockets are transmitted after \texttt{sys\_dup()} system call, therefore a dynamic threshold with \( t_{\text{max}} \) perturbing \texttt{sys\_dup()} prevents further connection perturbations. For buffer-related system calls, spyware and Trojan received very small percentage of perturbations, indicating the effectiveness of a dynamic threshold in perturbing spyware and Trojan before letting them send and receive buffers.

To understand the effects of different strategies under different background workloads, we carried out experiments running a flooder and a web server together in a system. In each experiment (running for 1 minute), the flooder will be perturbed with no or one strategy or a combination of all strategies, and the server will be running with either normal or heavy workloads. We measured the number of system calls invoked by the flooder and summarized in Figure 3. With strategy failing system call only, the total number of system calls invoked increased from 6,913,041 to 7,187,094, compared with no strategy employed. The reason is that flooders would immediately retry when a packet failed, saving the time for waiting for responses. With strategy delay only, the total number of system calls decreased by one third. Strategy priority decrease did not make any difference than no strategy, and this was because the flooder program already had the lowest priority in the server. When we randomly chose a strategy each time, the result was the best with the smallest number of system calls successfully sent. In the case of heavy workload, the number of system calls a flooder can send decreased by more than half. Priority decrease showed effectiveness by making a sharp decrease on the total number of system calls invoked. A server under heavy workload runs more processes than under normal workload, and some of the processes own lower priority than the flooder. Therefore, a random decrease of the flooder’s priority saved more system resources to the other benign but low-prioritized processes. Regardless of the workloads, mixed strategies had a better result than running a single strategy.

![Figure 3: Comparison among different types of strategies in the number of system calls invoked by a flooder, with normal and heavy workloads running in the background.](image)

4.2. Performance

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: (1) 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.

One of the greatest differences between malware and benign software is the diversity of functionality of the latter. To ensure the fairness of analysis on benign software, we measured the test coverage (percentage of software instructions executed) by compiling their source code with gcov [17], EMMA [18] and Coverage.py [19] based on the software’s programming language. The average coverage was 69.49%.

Figure 4 showed the results for benign software running in the uncertain environment with a dynamically changed threshold. 90% of the software succeed the execution, 5% crashed the execution and 5% received hampered execution. Compared with the
results from a static threshold ($t_d$), the succeeded ratio increased by 20%, the crashed ratio decreased by 12% and the hampered ratio decreased by 8%.

We further analyzed on the software that received crashed execution in the uncertain environment. Different from previous work using both intrusive and non-intrusive strategies [1], this work only considers non-intrusive strategies. Thereby we aim to dig into the reasons behind the crashes or hampered execution, so that benign software can improve itself to better adapt to the interference strategies.

We randomly sampled four benign software—Vim, tar, Mozilla Firefox and Thunderbird, observed the crashes manually, and found that the crashes were in fact software bugs reported before on Launchpad and Bugzilla [27, 28]. Table 3 listed the bugs in detail.

**.viminfo**: If you exit Vim and later start it again, one would normally lose a lot of information. The .viminfo file can be used to remember that information, which enables a user to continue where left off [29]. For the bug related to viminfo, it was caused by silencing a `sys_write()` on .viminfo file in our experiment. In the reported bug, it was be caused by a mis-operation (an operation with a special character not recognized by Vim before exiting). Unless knowing the fix (removing the .viminfo file), Vim will fail to start again. With CHAMELEON, the reason for failing in opening .viminfo will be unveiled.

**tar -C empty directory**: It is a bug when you extract the empty directories inside an archive using the `-C` option to change directories. The reason of this bug is that tar used `mkdir (file name, mode)` instead of `mkdirat (chdir fd, file name, mode)` to extract a directory. From CHAMELEON, the failure of creating a new file descriptor can be observed.

**Thunderbird mail spool file**: Thunderbird uses the spool file to “help” the user set up an email account with the assumption that the email providers address SMTP, ports, and security configuration etc. Unfortunately the reality is very few of them are correctly actually configured [30]. CHAMELEON records the places that Thunderbird is looking for spool file. A user can either manually change the settings or change the spool path.

The other bugs are all similar—resulting from the “assumptions” that developers have made. To sum up, it was more of the software design imperfection that led to a crashed or hampered execution of the benign software than the interferences in the uncertain environment. Therefore, CHAMELEON can be used as a framework to locate benign software design flaws and help developers improve software reliability.

The reasons behind a software bug may be many. From our observation, some of the bugs may emerge again after being ‘fixed’ for a while—a new design flaw is triggered. CHAMELEON is capable of interfer-

<table>
<thead>
<tr>
<th>Malware</th>
<th>Percentage of sysealls perturbed</th>
<th>Percentage of connection-related sysealls perturbed</th>
<th>Percentage of buffer-related sysealls perturbed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static Threshold</td>
<td>Dynamic Threshold</td>
<td>Static Threshold</td>
</tr>
<tr>
<td>Flooders</td>
<td>9.74%</td>
<td>37.91%</td>
<td>10.13%</td>
</tr>
<tr>
<td>Spyware</td>
<td>2.89%</td>
<td>14.33%</td>
<td>7.14%</td>
</tr>
<tr>
<td>Trojan</td>
<td>8.09%</td>
<td>21.17%</td>
<td>9.52%</td>
</tr>
<tr>
<td>Viruses</td>
<td>5.02%</td>
<td>23.47%</td>
<td>9.56%</td>
</tr>
<tr>
<td>Worms</td>
<td>0.05%</td>
<td>11.04%</td>
<td>9.86%</td>
</tr>
<tr>
<td>All</td>
<td>0.41%</td>
<td>19.80%</td>
<td>9.87%</td>
</tr>
</tbody>
</table>

Table 2: Comparison between a static threshold and a dynamically changed threshold on the percentage of system calls perturbed.
Table 3: Software bugs emerged in the uncertain environment

<table>
<thead>
<tr>
<th>Software</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vim</td>
<td>viminfo: Illegal starting char [20]</td>
</tr>
<tr>
<td></td>
<td>Fail using -C option extracting archive with empty directories [21]</td>
</tr>
<tr>
<td></td>
<td>Operation not permitted when extracting [22]</td>
</tr>
<tr>
<td>tar</td>
<td>Unable to locate mail spool file [23]</td>
</tr>
<tr>
<td></td>
<td>segmentation fault (core dumped) [24]</td>
</tr>
<tr>
<td>Thunderbid</td>
<td>Fatal IO error (Operation not permitted) on X server [26]</td>
</tr>
<tr>
<td>Firefox</td>
<td>Fatal IO error (Operation not permitted) on X server [26]</td>
</tr>
</tbody>
</table>

ing every system call with a probability, and logging the execution point that triggers the crash. When benign software become more robust, the uncertain environment will better protect the system from malware.

5. Discussion

As we discussed in Section 2, 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. Even though this work only considered non-intrusive strategies, 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.

To maximize the effects of the uncertain environment, we suggest developers to write more robust software, such as targeting end-to-end checks and time-consuming testing and verification procedures. We also suggest developers to make as few assumptions as possible in software design—a simple assumption today may make a huge defect tomorrow.

6. Related Work

Our work intersects the areas of malware detection, software diversity and deception, and fault injection and fuzz testing. This section summarizes how they have been used in software design and highlights under-studied areas.

Malware Detection: There are extensive literature dating to the 1990s on detection of intrusions and malware. Malware detection techniques can be signature-based [4, 5] or behavior-based [31, 32, 33].
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% [6].

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 [32] 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 [31, 33] are based on seminal work by Forrest et al. [31]. 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 [33].

Some approaches analyze the data flow of a program to extract malware behavior. Panorama [34], for example, performs system-level taint-tracking to discover how malware leaks sensitive data. Martignoni et al. [35] 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 [36] improved on tracing granularity on single instructions and system calls via hardware virtualization extensions. Ye et al. [37] 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 [6] 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 and Deception: 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. [38] proposed guidelines and advocated the use of randomized compilation techniques, which motivated later work in this area [39]. She and her colleagues [40] 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 [40].

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 [41].

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 [42] 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 [43]. 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. The researchers considered tactics such as responding with common system errors and inducing delays to frustrate attackers. Red-teaming experiments at Sandia tested the effectiveness of network deception on attackers working in groups. The deception mechanisms at the network level successfully delayed attackers for a few hours. Almeshekah and Spafford [44] 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 [45], a model of deception that challenges the cornerstone of projecting false reality with internal consistency. Sun et al. [46, 47] also argued for the value of unpredictability and deception as OS features. In this paper we explored non-intrusive unpredictable interferences to create an uncertain environment for software being deep analyzed after an initial borderline classification.

**Fault Injection and Fuzz Testing:** Fault injection is an important method for generating test cases in fuzz testing. Through fault injection, researchers are able to study fault propagation [48] and develop flexible and robust software and systems [49, 50, 51]. Kanawati and Abraham provide a methodology and guidelines for the design of flexible software, based on their experience with the fault injection tool FERRARI [49]. Fault injection has been applied to a number of abstractions. DOCTOR [50] for example, supports memory faults, CPU faults, and communication faults. FINE [48] traces execution flow and key variables through the UNIX kernel via hardware-induced software errors and kernel software faults injection. A recent survey on assessing dependability with software fault injection [52] provides a comprehensive overview of the state of the art fault injection approaches to fit the goals of researchers and practitioners. LFI tool [53] injects errors in library-calls, in order to identify error handling faults that arise from misunderstanding of library APIs, and from poor portability across different OSes. Other possible forms of fault injection are code mutations and data interface corruptions [54]. CHAMELEON is similar by injecting faults (interferences) in the execution of software at the system call level.

Fuzz testing is an effective way to discover coding errors and security loopholes in software, operating systems, and networks by testing applications against invalid, unexpected, or random data inputs. Miller et al. [55] first proposed fuzz testing as an inexpensive mechanism to generate additional software tests. The authors later extended the work [56] to identify missed return code checks from crucial calls, such as memory allocation. Many additional fuzz testing approaches have been proposed [57, 58, 59]. Trinity [60], for example, randomizes system call parameters to test the validation of file descriptors, and found real bugs [61], including bugs in the Linux kernel [62, 63, 64]. White-box fuzzy testers [65, 66, 67, 68] were also proposed to increase the coverage of test inputs by leveraging symbolic execution and dynamic test generation. For instance, KLEE [67] uses symbolic execution and a model of system call behaviors provided by a user to generate high-coverage test cases. BALLISTA [69] tests the data type robustness of the POSIX system call interface in a scalable way, by defining 20 data types for testing 233 system calls of the POSIX standard. CHAMELEON can also be thought as a fuzz tester at the OS system call API to understand how sensitive an application is to a particular type of misbehavior.

7. Conclusion

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-based 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 dynamically changed threshold caused various levels of disruption to 10% of the analyzed software. Malware was affected more with 92% of the malware to fail to accomplish their tasks. Compared with a static
threshold, 20% more benign software succeeded and 24% more malware crashed or hampered in the uncertain environment.

We also observed the details of crashed benign software, and found that many of the crashes resulted from an imperfect design of the software. Several bugs were reproduced for Vim, tar, Mozilla Firefox and Thunderbird. We provided recommendations for developers to improve software robustness, such as targeting end-to-end checks, time-consuming tests and validations.

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.

References


Coverage.py: https://github.com/msabramo/coverage.py

viminfo: Illegal starting character in line.
URL: http://www.explorelinux.com/fix-e575-viminfo-illegal-starting-char-line/

tar: fails using `-c` option extracting archive with empty directories.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=602209

tar: operation not permitted.
URL: https://bugs.launchpad.net/ubuntu/+source/file-roller/+bug/1238266

Thunderbird: unable to locate mail spool file.
URL: https://support.mozilla.org/en-US/questions/1157285

Thunderbird crashes with segmentation fault.
URL: https://bugs.launchpad.net/ubuntu/+source/thunderbird/+bug/571308

Firefox: bus error (core dumped).

Firefox: fatal io error 11 (resource temporarily unavailable) on x server: 0.
URL: https://bugzilla.mozilla.org/show_bug.cgi?id=895947

Launchpad.net.
URL: https://bugs.launchpad.net/

Bugzilla.
URL: https://www.bugzilla.org/

Viminfo documentation.

Thunderbird spool file.
URL: https://support.mozilla.org/en-US/questions/1021171


[60] Lca: The trinity fuzz tester. 
URL https://lwn.net/Articles/536173/

[61] Bugs found by trinity. 


Table 4: List of the 100 malware families used in our evaluation. We named the malware families used in our evaluation.