A Praise for Defensive Programming: Leveraging Uncertainty for Effective Malware Mitigation

Ruimin Sun*, Marcus Botacin¶, Nikolaos Sapountzis*, Xiaoyong Yuan*, Matt Bishop‡, Donald E. Porter§, Xiaolin Li†, Andre Gregio¶ and Daniela Oliveira*

University of Florida, USA * {gracesrm,nsapountzis,chbrian}@ufl.edu, *{daniela}@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 ¶{mfbotacin,gregio}@inf.ufpr.br
AI Institute, Tongdun Technology, China †xiaolin.li@tongdun.net

Abstract— A promising avenue for improving the effectiveness of behavioral-based malware detectors is to leverage two-phase detection mechanisms. Existing problem in two-phase detection is that after the first phase produces borderline decision, suspicious behaviors are not well contained before the second phase completes.

This paper improves CHAMELEON, a framework to realize the uncertain environment. CHAMELEON offers two environments: standard—for software identified as benign by the first phase, and uncertain—for software received borderline classification from the first phase. The uncertain environment adds obstacles to software execution through random perturbations applied probabilistically. We introduce a dynamic perturbation threshold that can target malware disproportionately more than benign software. We analyzed the effects of the uncertain environment by manually studying 113 software and 100 malware, and found that 92% malware and 10% benign software disrupted during execution. The results were then corroborated by an extended dataset (5,679 Linux malware samples) on a newer system. Finally, a careful inspection of the benign software crashes revealed some software bugs, highlighting CHAMELEON's potential as a practical complementary anti-malware solution.

Index Terms—OS, Uncertainty, Malware, Fuzzing

1 Introduction

Real-time malware detection is challenging. The industry still relies on antivirus technology for threat detection [56, 82], which is effective for malware with known signatures, but not sustainable for the massive amount of new malware samples released daily (practical detection rates from 25% to 50% [2]). Thus, the AV industry started to rely on behavioral-based detectors built upon heuristics, which are more "generic" than signatures, but suffer from high false-positive rates [19, 20]. In a company, aggressive heuristics, i.e., those that are too focused *on blocking suspicious software*, can interfere with employee's productivity, resulting in employees overriding or circumventing security policies.

In addition, existing AV software mostly aims to identify the signature or monitor the runtime behavior through isolating the software for a while until a decision can be made [19, 24, 18]. However, some software behavior may be hard to define. For example, malware may start by sleeping for five minutes or even longer and then perform malicious activities, or be only active during midnight and show benign behaviors for most of the time. These type of malware makes it hard for single step malware detector to give an accurate decision.

Recently, two-phase hybrid detection methods are gaining attention [60, 78] due to their capabilities in finding malware with complicated behaviors. In [78], the solution starts by using traditional machine learning models that are fast but not very accurate in its first-stage malware detection. If a borderline classification is received, modern deep learning methods that are accurate but time-consuming are performed for further analysis. However, the problem exists in this solution and any other two-phase detection methods is that, after the first phase gives a borderline decision, potential malware experiences no obstacles in executing its malicious behaviors before the second phase detector completes.

In this paper, we present CHAMELEON, a framework that separates the OS into two environments: standard and uncertain. The standard environment is a regular environment that all software starts execution from. In the uncertain environment, the software will experience probabilistic and random perturbations whose aim is to thwart the actions of potential malware while second phase analysis is under way. We provide a detailed description of the design and implementation of CHAMELEON, as well as new extensions to our framework.

The hypothesis is that the uncertain environment will mostly disturb malicious programs instead of benign ones. This is supported due to the increasing demand for defensive programming [5] among software producing organizations. Under this paradigm, poor-written malware code would be disproportionally more affected by the perturbations than well-written benign software, since defensive programming is a form of design intended to ensure the continuing operation of a piece of software under unforeseen circumstances, making the software behave in a reliable manner despite unexpected inputs or user actions.

In addition, malicious programs are not exquisite pieces of software overall—malware developers have to be able to quickly produce variants as AV signatures are created, causing them to be poorly coded. Malware also usually depend on specific configurations or installed applications to properly work, making them more prone to crashing due to uncertainties of the operating system (OS) it should run.

We evaluate the impact of uncertainty and unpredictability on actual malware samples compared to benign software and discuss the reasons why the samples fail to address the unexpected execution effects. CHAMELEON's strategy increases the cost of attacks, as writing malware using defensive programming requires additional programming efforts and time.

In our evaluation of CHAMELEON [75], we manually inspected the execution and its effects of 100 samples of Linux malware and 113 common benign software from several categories. Our results show that at a perturbation threshold 10% (i.e., a 10% probability of perturbation for every system call that could be perturbed), intrusive perturbation strategies thwarted 62% of malware, while non-intrusive strategies caused a failure rate of 68%. At a perturbation threshold 50%, the percentage of adversely affected malware increased to 81% and 76% respectively. With a 10% perturbation 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 were three times more sensitive to perturbations than CPU-bound software.

Finally, we introduced an optional dynamic perturbation threshold to CHAMELEON. This threshold is computed so as to penalize more intensively software presenting known malicious behavior. Our analysis show that the application of such threshold caused 92% of malware to fail and impacted only 10% of benign software. Compared with a static threshold, this dynamic threshold improved in 20% the number of benign software unaffected by the perturbations and caused 24% more malware to crash or be hampered in the uncertain environment. We also analyzed the crash logs from benign software undergoing non-intrusive perturbations, and found that it was actually software bugs that caused the crashes.

CHAMELEON has the potential to advance 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 chances of success. In this paper, we improved our work described in CHAMELEON [75], and presented the following **new contributions:**

- We designed and implemented a dynamic perturbation threshold based on the behavior of software execution. We showed that such threshold is more effective than a static threshold in that it causes more adverse effects to malware execution and less impact to benign software.
- We designed and implemented a fully automated Linux testbed[1] for collecting system call traces (at kernel level) from malware and benign software when these software is under perturbations. Such testbed can be leveraged to analyze benign software behavior under OS misbehavior and help developers pinpoint portions of their software that are sensitive to misbehavior, thus leading to more resilient software.
- We further collected 5,679 Linux malware samples and analyzed this extended malware dataset on a new version system. The results corroborated our findings on previous small sample set, and indicated CHAMELEON's capability of standing the test of time.
- We provided the results of analysis of case studies of applications running under CHAMELEON, including malware using three evasive stalling techniques, and commonly used benign software (e.g. vim, tar, Mozilla Firefox and Thunderbird) affected.

This paper is organized as follows. Section 2 describes our threat model and assumptions. Section 3 describes in detail CHAMELEON's design and implementation, including the newly proposed dynamic perturbation threshold. Section 4 describes CHAMELEON's security and performance evaluation, including our analysis of causes of crashes for benign software in the uncertain environment. Section 5 discusses and summarizes CHAMELEON's results and limitations. Section 6 summarizes related work on malware detection, software diversity, and attempts on unpredictability as a security mechanism. Section 7 concludes the paper.

2 THREAT MODEL AND ASSUMPTIONS

CHAMELEON's protection is designed for corporate environments, which have adopted the practice of controlling software running at their perimeters [20].

We assume that if an organization is a target of a well-motivated attacker, malware will eventually get in (e.g., spear-phishing). If the malware is zero-day, it will not be detected by any signature-based antivirus (AV). If the malware receives a borderline classification by behavioral-based detectors, it might lurk inside the organizations for extended periods of time. With CHAMELEON, if this piece of malware might receive a borderline classification at some point by a conventional machine learning detector, it would then be placed in the uncertain environment. In this environment the malware would encounter obstacles and delays to operate, while more time and resource-consuming deep analysis is underway to definitely flag it as malicious.

CHAMELEON does not compete with standard lines of defenses, such as conventional AVs, behavioral-based detectors, and firewalls, but actually equips these solutions with a safety net in case 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, and (ii) an uncertain environment, where a subset of the OS system calls for selected processes undergo unpredictable perturbations.

The key insight is that perturbation in the uncertain environment will hamper malware's chances of success, as some system calls might return errors in accessing system resources (e.g. network connections or files) or cause malware execution delays.

1. The testbed is publicly available at https://github.com/gracesrm/Chameleon-malware-testbed

3.1 The Perturbation Set

Our first step was deciding which system calls were good candidates for perturbation. We relied on Tsai et al.'s study [81], which ranked Linux system calls by their likelihood of use by applications. Based on these insights, we selected 37 system calls for the perturbation 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 nearly all malware, regardless of its sophistication level.

We introduced new versions for all system calls in the perturbation set. For each system call `orig_<syscall_name>` in the perturbation set, CHAMELEON altered the corresponding table entry to point to `my_<syscall_name>`, in which perturbations were added if the software was executing under the uncertain environment.

TABLE 1: System call perturbation set.

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

3.2 Perturbation Strategies

We introduced two sets of perturbation strategies. The first, *non-intrusive*, perturbed software execution within the OS specification. The second, *intrusive*, could cause corruption in the software execution. From the end user point of view, intrusive strategies might cause functionalities to be temporarily unavailable. Non-intrusive strategies might cause software to run slower.

3.2.1 Non-intrusive Perturbation Strategies

1. System call silencing with error return: The system call immediately returns an error value randomly selected from the range [-255, -1]. This perturbation strategy can create difficulties for the execution of the process which do not handle errors well. Further, this strategy can cause transient unavailability of resources, such as files and network, creating difficulties for certain types of malware to operate. In this perturbation type, all error returns are within the OS specification, i.e., the expected set of return values.
2. Process delay: The system call injects a random delay during its execution to delay potential malware execution. It can create difficulties in timely malware communication with a C&C for files ex-filtration, as well as prevent flooders from sending enough packets in a very short time, rate-limiting DoS attacks. The delay range was chosen as a random number within [0,0.1] as an experimental range. A delay longer than 0.1s could cause network applications to timeout and terminate early.
3. Process priority decrease: The system call decreases the dynamic process priority to the lowest possible value, delaying process scheduling.

3.2.2 Intrusive Perturbation Strategies

1. System call silencing: The system call immediately returns a value (without being executed) indicating a successful execution.
2. Buffer bytes change: The system call experiences an increase or decrease in the size of a buffer passed as parameter. 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 making exfiltration of sensitive data more difficult. This strategy also targets viruses, which can be adversely affected by the disruption of the buffer with a malicious payload trying to be injected into a victim's ELF header—the victim process may get corrupted and lose its ability to infect other files.
3. Connection restriction: The strategy 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.
4. File offset change: The strategy 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 within a specified, configurable range.

3.3 System Architecture

The key component of CHAMELEON is a kernel-level uncertainty module (see Figure 1) which is responsible for implementing the uncertain environment. Specifically, this module (i) hooks into the Linux system call table [2] to replace system calls in the perturbation set with new versions of system calls that apply perturbations (Step 0 in Figure 1), (ii) monitors system calls invoked by processes in the uncertain environment, and (iii) applies perturbations to the system calls when required. The perturbation strategies are chosen randomly, and applied probabilistically, in the uncertain environment.

CHAMELEON added the following fields to the Linux `task_struct`:

`process_env`: a flag informing whether or not the process should run in the uncertain environment.

`fd_list`: a list of critical file descriptors during process execution. Applying perturbation to system files, such as library or devices, will likely cause the process to crash. Thus, CHAMELEON does not apply perturbations to system calls manipulating those file descriptors (see Section 3.5 for more details).

`strategy_set`: a flag informing the type of perturbation strategies the process should undergo: non-intrusive or intrusive.

`threshold`: an integer informing the probability that a system call from the perturbation set invoked by a process in the uncertain environment will undergo perturbation. In other words, the threshold represents the strength of the perturbation to be applied to a system call. The higher the

2. For system call monitoring, we choose to hook into the system call table through a loadable kernel module rather than using `strace`, because malware with anti-analysis techniques may stop executing when `strace` is detected.

threshold, the higher the probability that a perturbation strategy will be applied. By default, threshold 10% is used.

CHAMELEON's operation is illustrated in Figure 1. Consider `Process 2`, running in the uncertain environment invoking `sys_write` (Step 1), which belongs to the perturbation set. Thus, as determined by the perturbation threshold, a perturbation strategy can be applied to its execution. During CHAMELEON's operation, the hooked system call first inspects `Process 2`'s environment and finds that it runs in the uncertain environment (Step 2). Next, `sys_write` runs the corruption protection mechanism (see Section 3.5 for details) to make sure that no perturbation 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 probabilistic threshold whether or not a perturbation should be applied. If a perturbation is to be applied, `sys_write` randomly selects one of the perturbation strategies that can be applied to its execution.

3.4 Dynamic Perturbation Threshold

As explained earlier, the perturbation threshold, denoted as $T_{syscall}$ hereafter, represents the *probability* that a system call belonging to the Perturbation Set and invoked by a process in the uncertain environment will undergo perturbations. In our original work [75], we assumed the same default perturbation threshold for all system call invocations and all $syscall$ types when applicable. In this section, we go a step beyond, and propose threshold $T_{syscall}$ that will change based on the execution context and in a per-system call, per-process fashion: it will be higher for processes exhibiting known likely malicious behavior (to ensure more system calls invoked by a malicious software are perturbed) and will be lower for likely benign processes. To that end, we introduce a dynamic *perturbation threshold* that changes based on the system call type and invocation context of a given process. The goal is to reduce the chance for benign software to be false positively killed.

In the remainder of the section, we describe two families of behaviors that are considered suspicious in the literature [59, 63, 84, 50]. First, in Section 3.4.1 we consider the *"signature"* of the system call, ***namely a type of system call along with its parameters*** (Behavior family A). The literature reports some signatures for a few malicious invocations of system calls [59, 63, 84, 50]. To that end, if a system call is going to be invoked with a combination of parameters matching the signature of a known likely malicious behavior, CHAMELEON increases the threshold for that particular system call invocation to some pre-defined threshold, thus affecting the system call execution on-the-fly. Second, in Section 3.4.2 we consider the invocation *frequency* of the system call (Behavior family B). Specifically, if a system call is invoked in an abnormally frequent manner and matches a likely malicious behavior (e.g. DoS), CHAMELEON perturbs its invocation with a probability that changes according to the invocation frequency. For each family of behaviors (signature or frequency), we identify examples of some malware representative of the behavior, but one can extend these families to consider more malicious behaviors. In Table 2 we summarized the notation used.

3.4.1 Behavior family A: a signature-based threshold

In the first family of malicious activities, we propose three system call *signatures* for three different malicious behaviors. We found that these *signatures* are prevalent in approximately 95% of the malware samples, and occurred in only 5% of the benign software samples in the system call traces we collected. Therefore, it is plausible to assume that a process invoking such behavior is likely malicious.

(A1. ELF Header Injection): A strategy often employed by malware samples to get privileged access to a system is to replace existing binary contents with malicious payloads, thus benefiting from root and/or whitelist execution permissions previously attributed to the affected binary. This kind of injection can be correlated to the signature `sys_write("\177ELF")` [59], which considers the ELF (Executable and Linkable Format) magic number as argument for the `sys_write` system call.

(A2. I/O Redirection): Malware samples often rely on redirecting standard I/O descriptors to implement their malicious behaviors. The keyboard descriptor is often redirected by keyloggers to allow data collection. The screen output descriptor is often redirected by remote shells to hide attacker's commands. I/O redirection can be correlated to signatures targeting system calls that modify the standard I/O descriptors, such as `sys_dup(fd)`, where $fd = \{0, 1, 2\}$ is used as a stub to obtain the victim server's standard input, output and error, respectively [63].

(A3. Replacing system binaries): Rootkits often replace (rewrite, remove or unlink) existing legitimate applications by malicious/trojanized versions so that they can violate some security policy or hide malware traces [50]. In this sense, system binaries are the most targeted files by rootkits [84], given their ability to provide users with system information, such as the running processes list. This behavior can be correlated to the system call signatures `sys_unlink("path")` or `sys_rename("path")`, and the "path" refers to the location of system binaries, e.g. "/bin", and "/usr/bin".

For the sake of generality, we introduce three different thresholds for each cases of behavior family A, as they might refer to different levels of likelihood of malicious activity. We assume these thresholds to be input parameters, that can be changed based on a variety of factors, such as the running application, the requirements of an organization, etc. Specifically, we assume these thresholds to be t_{A1}, t_{A2}, t_{A3}, respectively, where $A1 = sys\_write("\backslash 177ELF")$, $A2 = sys\_dup(fd)$, $A3 = \{sys\_unlink("path")$ **or** $sys\_rename("path")\}$.

Note that CHAMELEON does not kill the corresponding software nor it increases the perturbation threshold to 100% ($t_{Ai} < 1$) because benign software might also exhibit such behaviors with a small probability, and CHAMELEON expects benign software written with good quality to be resilient to those perturbations.

3.4.2 Behavior family B: a frequency-based threshold

The second family of suspicious behaviors can be represented by the frequent invocation of a certain type of system call during a process execution, as shown by Ptacek and Newsham [70]. The goal of this family of malicious

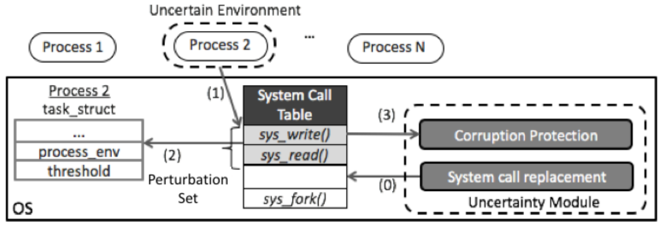

Fig. 1: CHAMELEON's architecture. When a process running in the uncertain environment invokes a system call in the perturbation set (Step 1), the *Uncertainty Module* checks if the process is running in the uncertain environment (Step 2), and depending on the execution of the corruption protection mechanism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

TABLE 2: Notation. These are per-process parameters.

(Subscripts)	
$A1$	System call Behavior A1: *sys_write("/177ELF")*
$A2$	System call Behavior A2: *sys_dup(0), sys_dup(1), sys_dup(2)*
$A3$	System call Behavior A3: *sys_unlink("path"), sys_rename("path")*
$syscall$	System call type $syscall$
max,min	The maximum/minimum allowed value of an input
Inputs	
t_{Ai}	Fixed perturbation threshold for Behavior family A
p	Coefficient of the relationship between $F_{syscall}$ and $T_{syscall}$
N_{min}	Minimum allowed N for Behavior B
F_{min}	Minimum allowed $F_{syscall}$ for Behavior B
t_{max}	Maximum allowed value for all perturbation thresholds
Variables	
$T_{syscall}$	Perturbation threshold for the system call type $syscall$
$n_{syscall}$	Total number of system calls invoked for type $syscall$
$F_{syscall}$	Invocation frequency for the system call type $syscall$
N	Total number of system calls invoked

behaviors is to aggressively consume system resources until the system cannot function for some or all of its legitimate requests (e.g. DoS). In the following, we list two suspicious behaviors belonging to this family and the corresponding system calls that are frequently invoked.

(B1. Network flooding.) This can represent malware seeking to make network resources unavailable to its intended users by temporarily or indefinitely disrupting services of a host connected to the Internet. Network flooding is typically accomplished by flooding the targeted machine with frequent invocations of `sys_sendto` or `sys_recvfrom`.

(B2. Fork bomb.) This can represent malware forking processes infinitely until the system runs out of memory. Once the fork bomb is activated, it may not be possible to resume normal operation without rebooting the system. A fork bomb, as the name implies, is characterized by frequent invocations of `sys_fork`.

In the remainder of the section, we: *first*, explain when a system call is considered as Behavior family B based on the invocation frequency, and *then*, we derive a formula that dynamically adapts their threshold based on the system call invocation frequency. Let us denote, in a process, the number of times that the system call type *system call* is invoked as $n_{syscall}$ and the current total number of system calls invoked as N. Then, we introduce the invocation frequency of the system call *syscall* as[3]:

$$F_{syscall} = \frac{n_{syscall}}{N}. \quad (1)$$

We assume that a system call $syscall$ falls into Behavior family B when both of the following conditions occur (i) the invocation frequency $F_{syscall}$ is higher than a pre-defined minimum frequency F_{min} (namely $F_{syscall} \geq F_{\text{min}}$), and (ii) the total number of system calls invoked N exceeds a minimum number N_{min} (namely $N \geq N_{\text{min}}$), where F_{min} and N_{min} are both input parameters. The first condition ensures that the system call invocation frequency is high enough to be considered as suspicious. The second condition ensures that the software has launched itself and the first system calls invoked by a process are not mistakenly perturbed.

We now derive the per-system call threshold $T_{syscall}$ as a function of frequency. The goal is to increase the threshold $T_{syscall}$ as the frequency $F_{syscall}$ increases, because the higher $F_{syscall}$ the more likely the process is exhibiting one of the behaviors in Behavior family B. One could use different functions for that objective, and "penalize" the threshold $T_{syscall}$ (and further the probability of a system call being perturbed) on $F_{syscall}$ in a linear, quadratic, cubic, etc. manner. Different powers correspond to different penalty increases. Without loss of generality, we chose to use linear function with coefficient p as an input parameter, and the threshold of *system call* is defined as follows:

$$T_{syscall} = \begin{cases} p \cdot F_{syscall}, & \text{if } p \cdot F_{syscall} \leq t_{\text{max}} \text{ ,} \\ t_{\text{max}}, & \text{otherwise.} \end{cases} \quad (2)$$

We want to keep $T_{syscall} \neq 1$ to prevent the scenario where all system calls are perturbed and software crashes at the very beginning, and thus $T_{syscall}$ takes values $0 \leq T_{syscall} < 1$. The input parameter $0 \leq t_{\text{max}} < 1$ dictates the maximum value that $T_{syscall}$ can obtain. Specifically, the threshold $T_{syscall}$ increases linearly on the product between the coefficient and the invocation frequency, namely $p \cdot F_{syscall}$, when the latter is less than t_{max} (see first branch of Eq. 2). On the other hand, if the product exceeds the maximum value t_{max}, then the threshold is t_{max} (see second branch of Eq. 2). t_{max} allows the user to customize the maximum threshold, namely the strongest perturbations desired for the software. Note that, as we need to keep $0 \leq T_{syscall} < 1$, p can take values $0 \leq p < \frac{1}{F_{syscall}}$.

We show here an example for the second family of suspicious behaviors, with $F_{\text{min}} = 0.7$. Assuming a flooding attack on a machine, that invokes a `sys_recvfrom` system call with invocation frequency $F_{\text{syscall}} = 0.9$, and has an input coefficient parameter $p = 0.8$. Then, the threshold will be $T_{\text{syscall}} = 0.8 * 0.9 = 0.72$. This is classified as the second family of suspicious behaviors and the `sys_recvfrom` system call has 72% probability to encounter perturbations. It is clear that, the higher the frequency $F_{syscall}$ or the parameter

3. Note that the invocation frequencies of all system calls should sum to 1, i.e. $\sum_{systemcall} F_{syscall} = 1$.

p, the higher the $T_{syscall}$ and thus the more frequent the software will encounter perturbations.

Note that a certain system call can simultaneously belong to the two families of suspicious Behavior families A and B. For example, system call `sys_write("\177ELF")` that is hallmark for the Behavior family A may also have high invocation frequency ($F_{\text{sys\_write}} > F_{\text{min}}$) suggesting that it also belongs to Behavior family B. In this case, each behavior would suggest a threshold: the first family would suggest t_{A1} and the second family would suggest $T_{\text{sys\_write}}$. As the probability for benign software to exhibit two suspicious behaviors at the same time is relatively small, we chose to use the higher threshold between the two behaviors i.e. $T = \max\{T_{sys\_write}, t_{A1}\}$ for the software under consideration.

For each of the input parameters, we chose an experience value to carry out our experiments. Without loss of generality, we chose $t_{A1} = t_{A2} = t_{A3} = t_{\text{max}}$ for simplicity of the experiment. We chose $t_{\text{max}} = 0.95$ because the strongest perturbations should still allow software to run and exhibit its behaviors, namely t_{max} should be close to but smaller than 1. We chose $N_{\text{min}} = 100$ because most of the software can finish loading its libraries by the time of 100 system calls are invoked. We chose $F_{\text{min}} = 0.7$ because the least aggressive DoS software can still invoke a certain type of system call as frequent as to occupy 70% of all the system calls. We chose $p = 1.2$ (since $F_{\text{min}} = 0.7$ the maximum value of p is 1.42 - see Eq. 2). These values cause less perturbations in benign software while affected malware more, as our evaluation results showed in Section 4. For system calls which do not exhibit behaviors, as described by behavior families A and B, the default threshold will be used.

3.5 Corruption Protection Mechanism

The uncertainty module employs a corruption protection mechanism to prevent perturbations while a process in the uncertain environment is accessing critical system files, which might cause process to crash at a very early stage. The files are identified through file descriptors, created by `sys_open`, `sys_openat` and `sys_creat`, and are 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 [36]).

When a process running in the uncertain environment opens a file with a path name beginning with the name or containing keywords of the critical directories, the file 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 perturbation strategies from being applied to these system calls.

Algorithm 1 shows how the OS applies the perturbation strategies on `sys_write`. First, the following conditions are checked: (i) the process is running in the standard environment ($process\_env == 0$), and (ii) the targeted file descriptor is a critical system file. If either of the two conditions is true the system call runs normally. Otherwise, the system call updates its execution counters of the current process (*i.e.* the total number of system calls invoked N and the total number of `sys_write` invoked $n_{\text{sys\_write}}$) and check whether `sys_write` exhibits frequent invocation based on the input parameter F_{min}. 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 perturbation.

The algorithm will randomly select one of the perturbation strategies based on the strategy type. If non-intrusive strategies are selected, one of the following strategies will be randomly selected for execution: *System call silencing with error return*, *Process delay*, or *Process priority decrease*. If `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 $[0, 0.1s]$. Our experiment show that a delay longer than 0.1 second will cause the program to timeout and the software to terminate at an early stage. If *Process priority decrease* is selected, the algorithm decreases the process priority to the minimum. If intrusive strategies are selected, perturbation strategies *System call silencing*, or *Buffer bytes change* will be randomly selected. If *System call silencing* is selected, CHAMELEON`sys_write` will return the buffer size without actually writing to the file. Otherwise, CHAMELEON will change the buffer bytes and return the new length of the buffer.

4 Evaluation

The goal of our evaluation was to discover the impact of CHAMELEON's uncertain environment in affecting malware and benign software behavior. We considered security, performance, and software behavior to answer the following research questions: (i) how will the uncertain environment with perturbation strategies affect software execution? (ii) is the per-process, per-system call perturbation threshold more effective than a static threshold? (iii) how different strategies impact malware in the uncertain environment?, and (iv) how can benign software be more resilient in the uncertain environment?

Our evaluation leveraged a collection of 113 software including common software from GNU projects [13], SPEC CPU2006 [23] and Phoronix-test-suite [22] (47 I/O-bound and 66 CPU-bound). The malware samples used in our evaluations were selected from THC [27] and VirusShare [33].

Our selection criteria was to have a diverse software dataset, which at the same time, could allow timely *manual* analysis of all aspects of execution. For certain cases, the outcome of software execution can only be analyzed via manual inspection (e.g., the outcome of files produced by a text editor under the uncertain environment). Other reasons for the need to perform *manual* analysis were as follows. First, we needed to reverse engineering malware binaries to discover whether the malware sample was a self-contained *desktop* binary, which libraries and versions were required for the malware execution, which input parameters were required for execution, and to correctly install the specific software and version that a certain malware would inject

Algorithm 1: Applying perturbations to `sys_write()`

```
Function long my_sys_write(fd, buf, size)
    if process_env == 0 or corruption_protection(sys_write, pid,
     fd_list) then
        return orig_sys_write(fd, buf, size);
    else
        boolean top;
        freq = isFrequentCalls(N++, n_sys_write++);
        updateThreshold(freq);
        if (random(0.0, 1.0) > threshold) then
            return orig_sys_write(fd, buf, size);
        end
        if strategy_set == Non − intrusive then
            strategy = random(1,3);
            if strategy = 1 then
                /* System call silencing with
                   error return                    */
                return random(-255, -1);
            else if strategy = 2 then
                /* Process delay                   */
                delay(random(0, MAX_DELAY));
                return orig_sys_write(fd, buf, size);
            else
                /* Process priority decrease       */
                decrease_current_priority();
                return orig_sys_write(fd, buf, size);
            end
        end
        else
            /* strategy_set == Non − intrusive    */
            strategy = random(1,2);
            if strategy = 1 then
                /* System call silencing           */
                return size;
            else
                /* Buffer bytes change             */
                newbuf = RandomBytes(buf);
                return orig_sys_write(fd, newbuf, size);
            end
        end
    end
```

payload into. Second, we needed to prepare the malware with all its required resources, such as installing libraries, setting up the environment, and launching the victim software.

We analyzed 100 malware samples belonging to different categories (22 flooders, 14 worms, 15 spyware, 24 Trojans, and 25 viruses). The samples contained executables built on both x86 and x86_64 systems. All the malware and benign software used in our experiments are detailed in Table 12 and Table 11 in the Appendix.

We deployed and evaluated CHAMELEON on two virtual machines (VMs) both running Ubuntu 12.04 with 1GB RAM, 30GB Hard Disk, and one processor. One VM used x86 architecture, and the other used x86_64 architecture.

4.1 Testbed and Data Collection

We implemented a testbed to perform our experiments. The testbed is deployed in a host desktop running Ubuntu 14.04 with 16GB RAM, 160GB Hard Disk, x86_64 architecture, and 8 processors. In each test, the testbed runs two VMs: (i) Test VM, running CHAMELEON and testing malware and benign software, and (ii) Victim VM, running with resources that the malware in the Test VM may want to attack. This subsection details the architecture of the testbed (Figure 2) and the process of automating scalable experiments. It has two virtual machines and four other components running as user land processes: (i) a central *Controller*, (ii) a *Resource Scheduler*, (iii) a *Task Scheduler* and (iv) a *Data Collector*.

Before starting the experiments, we set up the firewall to block all the possible network connections between the host and the VMs for security purposes. The VMs can communicate with each other, and can reach outside network through port 53 (DNS) and port 80 (HTTP) needed for malware downloading payloads from the Internet (Step 0 in Figure 2).

The *Controller* is responsible for managing the other components. It first starts the *Resource Scheduler* to prepare the files and parameters for all the experiments, then launches the *Task Scheduler* to run malware and benign software in the Test VM, and finally starts the *Data Collector* to record system call traces and execution results of each experiment (Step 1).

The *Resource Scheduler* is responsible for preparing the resources needed for each experiment in the Test VM. First, it reverts the Test VM into a clean snapshot—a preserved state of the VM that the user can return to repeatedly. Then, it loads the uncertainty module to the Test VM (Step 2). Finally, it copies the software and the corresponding files and parameters needed during the execution from the host to the Test VM (Step 3).

The *Task Scheduler* is responsible for executing tasks in the VMs. This involves starting the Dionaea [8] service in the Victim VM, and executing malware or benign software in the Test VM (Step 4). The Dionaea [8] service provides the required resources, such as network services, for malware running in the Test VM.

The *Data Collector* is responsible for collecting system call traces logged in `dmesg` and software execution results, including returned error and segmentation fault (Step 5).

4.2 Security

The goal of the security evaluation is to analyze the effects of CHAMELEON's uncertain environment in malware and benign software execution. We considered that a piece of malware was *adversely affected* by the uncertain environment if it crashed or experienced issues in its executed. An execution is considered *Crashed* if malware terminates before performing its malicious actions. An execution is considered *Succeeded* if malware accomplished its intended tasks, such as injecting malicious payload into an executable. The following outcomes are non-exhaustive 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 effects of the uncertain environment with 100 Linux malware samples (see Table 12 in the Appendix for the list) using intrusive and non-intrusive perturbation strategies and static and dynamic thresholds. As Figure 3 shows, on average, intrusive strategies produced approximately 10% more Crashed and 8% fewer Hampered

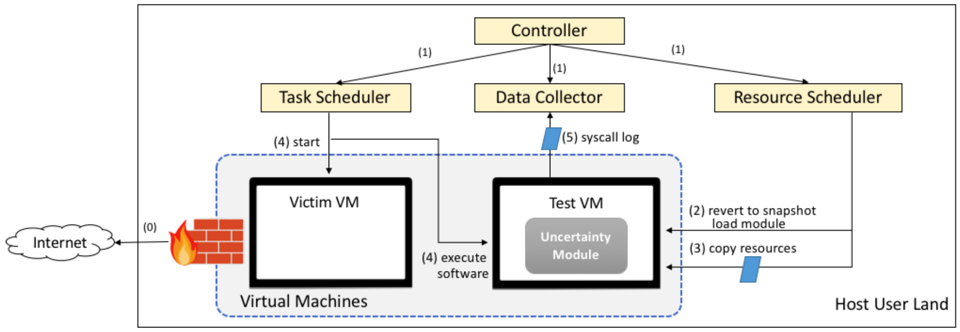

Fig. 2: The architecture of our evaluation testbed. The firewall restricts network from all VMs (Step 0). The Controller starts the Resource Scheduler, the Task Scheduler and the Data Collector (Step 1). The Resource Scheduler reverts the Test VM to a clean Snapshot and loads into it the uncertainty module (Step 2), copies the software resources (e.g. files and parameters needed) (Step 3). The Task Scheduler starts the Dionaea service in the Victim VM and executes software in the Test VM (Step 4). The Data Collector reads system call traces and execution results to aid crash analysis (Step 5).

execution results than non-intrusive strategies. For both intrusive and non-intrusive strategies, the ratios of Succeeded malware execution (infection) were almost the same. When intrusive strategies were applied, 81% of the malware samples failed to accomplish their tasks at threshold 50%, 62% failed at threshold 10%, and 92% failed with a dynamic per-system call threshold. Non-intrusive strategies yielded similar results for threshold 50%, 10% and per-system call threshold, with 76%, 68%, 93% of malware adversely affected, respectively. In general, threshold 50% caused more Crashed and fewer Succeeded malware execution results than threshold 10%. The dynamic threshold caused 25% fewer malware to succeed, and caused 30% more malware to crash during execution than static threshold. This corroborates our assumption that a dynamic threshold focusing on likely malicious behavior will be effective at targeting malware.

We also ran our samples of general software in the uncertain environment and observed their execution outcome. Non-exhaustive examples of *Hampered* executions are: (1) a text editor temporarily losing some functionality; (2) a scientific tool producing partial results; or (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 generated outputs that matched those produced with the same test case in the standard environment and with a runtime that did not exceed twice than that in the standard runtime.

As expected (Figure 4), compared to non-intrusive strategies, intrusive strategies caused more adverse effects to benign software with approximately 10% more Crashed, 7% more Hampered, and 15% fewer Succeed execution. At static threshold 10% with intrusive strategies, on average, 37% of the tasks experienced some form of Crashed or Hampered execution. With non-intrusive strategies, this percentage was 30%. For a 50% static threshold and intrusive strategies, 59% of the software was adversely affected. With non-intrusive strategies, this number was 10% smaller. A dynamic threshold with non-intrusive strategies on benign software led to 25% more Succeeded, 15% fewer Hampered, and 20% fewer Crashed executions for benign software than a static threshold for the same configurations. With intrusive strategies, the effects of the dynamic threshold were similar to using a static threshold of 10%.

These results corroborate our hypothesis that the uncertain environment with a dynamic threshold can better isolate benign software from perturbations, and disproportionately affect malware compared to using static thresholds.

4.3 Software Behavior and Performance

We compared the execution of malware and benign software at the system call level in the uncertain environment. We explored the effects of different software types, software workloads, and perturbation strategies.

In our experiments, benign software invoked more than twice the number of system calls invoked by malware (even flooders, which usually invoke a large number of system calls). For benign software the number of system calls perturbed or silenced was only half of those of malware, mainly because of the effectiveness of the corruption protection mechanism (see Section 3.5). Benign software performed a larger number of connection attempts and read/write operations than malware.

Table 3 and Table 4 show the results of system calls perturbed for different types of malware using non-intrusive and intrusive strategies. The impact of intrusive and non-intrusive strategies is similar. Generally, threshold 50% caused higher percentage of perturbations than a static threshold of 10% and dynamic threshold, especially with connection-related system calls (50% increase in the percentage of perturbations). The dynamic threshold caused higher percentage of perturbations than static threshold $t_d = 10\%$, because a dynamic threshold could vary from $t_d = 10\%$ to $t_{max} = 95\%$. For all types of system calls invoked, flooders had the highest percentage of perturbations and worms had the lowest percentage of perturbations, with both static and dynamic threshold. Based on previous work [75], this can be explained by flooders invoking a large number of system calls in the perturbation set, while worms invoking a smaller number. For network-related system calls, spyware had the lowest percentage of perturbation with both static and dynamic perturbation thresholds. With a dynamic threshold, spyware had 0% of its network-related system calls perturbed. This can be explained that network-related system calls are usually invoked after `sys_dup(0)` system call, which was perturbed with threshold $t_{max} = 95\%$. In other words, network-related system calls can hardly be invoked

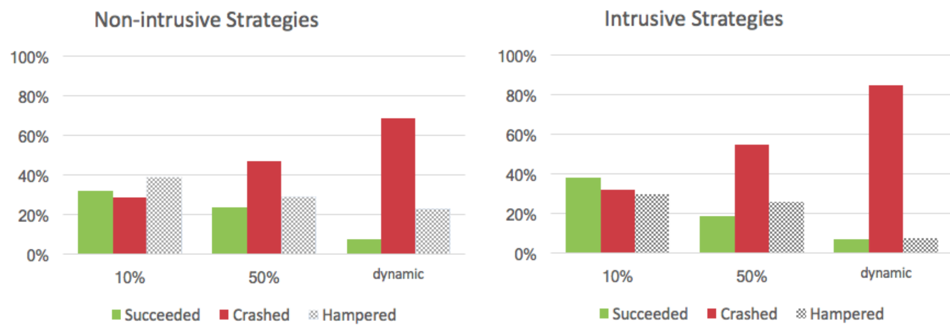

Fig. 3: Execution results for *malware* running in the uncertain environment using intrusive and non-intrusive strategies with static (10% and 50%) and per-system call thresholds.

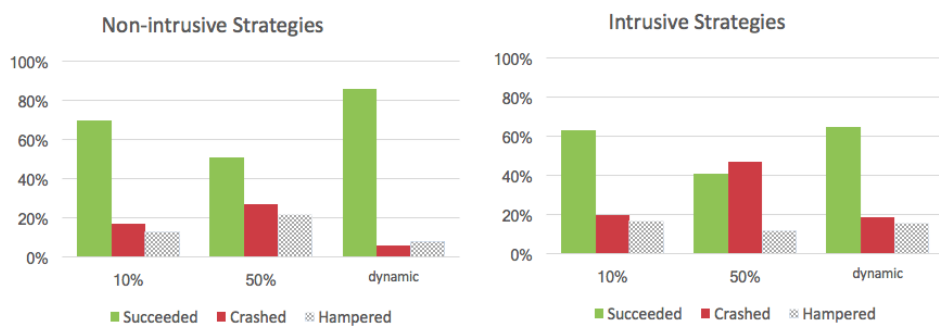

Fig. 4: Execution results for *benign software* running in the uncertain environment using intrusive and non-intrusive strategies with static (10% and 50%) and dynamic thresholds.

before spyware crashes. Similarly, for buffer-related system calls, spyware also experienced a very small number of perturbations, indicating a crash after `sys_dup(0)` system call.

Tables 5 and 6 show the results of system call perturbation for benign I/O-bound and CPU-bound software using non-intrusive and intrusive strategies. In general, a threshold of 50% caused higher percentages of system calls perturbed than a threshold of 10%. With static thresholds of 10% and 50%, compared with IO-bound software, CPU-bound software experienced a higher percentage of system calls perturbed, and lower percentages of connection-related system calls and buffer-related system calls perturbed. Dynamic threshold applied to I/O-bound software caused a higher percentage of system calls perturbed compared to CPU-bound software, mainly because the dynamic threshold is likely to affect IO-related system calls more (see Behaviors in Section 3.4).

One of the greatest differences between the malware and benign software analyzed in this study is the diversity of the latter. To ensure a fair analysis of benign software, we measured the test coverage (percentage of software instructions executed) by compiling benign software source code with gcov [12], EMMA [9], and Coverage.py [6] based on the software's programming language. The average coverage for benign software in our analysis was 69.49%.

We also analyzed the performance penalty caused by the perturbation strategies, such as process delay and process priority decrease on all 23 benchmark software whose execution could be scripted (see Table 11 in the appendix for a list of these benchmark software). Highly interactive software was tested manually and showed negligible performance overhead. Figure 5 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 was 8%; for runtimes ranging from 0.1 to 1 seconds, the average penalty was 4%; for runtimes longer than 10 seconds, the average penalty was 1.8%. This shows that the longer the runtime, the smaller the performance overhead. One hypothesis is that software with longer execution time is usually CPU-bound. Because most of the system calls in the perturbation set are I/O related, CPU-bound programs end up being perturbed less intensively.

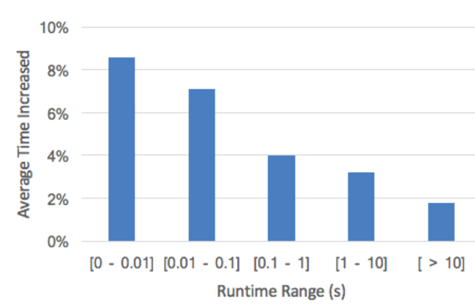

Fig. 5: Performance penalty for 23 benchmark software whose execution time could be scripted. We categorized the software according to their average runtime.

We also tested 26 benign applications with different workloads running in the standard and uncertain environment (see Table 11 in the Appendix for the list of these applications). The workloads were characterized under three levels: *light*, *medium* and *heavy*, which corresponded to *test*,

TABLE 3: Percentage of system calls perturbed running malware with different thresholds in the uncertain environment using *non-intrusive* strategies.

	Percentage of all syscalls perturbed			Percentage of connection-related calls perturbed			Percentage of buffer-related calls perturbed		
	10%	50%	Dynamic	10%	50%	Dynamic	10%	50%	Dynamic
Flooders	9.74%	39.31%	37.91%	10.13%	59.29%	36.68%	6.58%	23.35%	24.92%
Spyware	2.89%	25.79%	14.33%	7.14%	48.15%	0.00%	3.06%	31.06%	0.41%
Trojan	8.09%	27.07%	21.17%	9.52%	62.14%	17.00%	7.14%	15.22%	1.81%
Viruses	5.02%	28.62%	23.47%	9.56%	47.78%	12.69%	4.96%	21.87%	17.27%
Worms	0.05%	15.67%	11.04%	9.86%	60.97%	8.11%	8.97%	14.37%	16.27%
All	0.41%	28.39%	19.80%	9.87%	60.97%	15.69%	6.83%	21.10%	13.81%

TABLE 4: Percentage of system calls perturbed running malware with different thresholds in the uncertain environment using *intrusive* strategies.

	Percentage of all syscalls perturbed			Percentage of connection-related calls perturbed			Percentage of buffer-related calls perturbed		
	10%	50%	Dynamic	10%	50%	Dynamic	10%	50%	Dynamic
Flooders	8.22%	39.28%	39.11%	9.56%	49.57%	38.37%	3.50%	23.11%	30.69%
Spyware	4.38%	26.39%	15.02%	16.62%	51.25%	37.96%	0.64%	27.14%	0.07%
Trojan	6.90%	35.16%	25.08%	12.49%	56.62%	19.45%	3.58%	27.47%	6.08%
Viruses	6.49%	23.03%	28.14%	12.96%	52.34%	14.30%	8.94%	22.32%	24.19%
Worms	3.92%	22.93%	13.26%	6.53%	60.59%	8.15%	3.52%	19.86%	33.24%
All	6.26%	29.90%	26.27%	11.26%	53.55%	26.79%	4.47%	24.04%	19.36%

TABLE 5: Percentage of system calls perturbed running benign software with different thresholds in the uncertain environment using *non-intrusive* strategies.

	Percentage of all syscalls perturbed			Percentage of connection-related calls perturbed			Percentage of buffer-related calls perturbed		
	10%	50%	dynamic	10%	50%	dynamic	10%	50%	dynamic
IO	1.24%	7.65%	2.60%	5.94%	23.40%	2.50%	0.94%	6.34%	34.16%
CPU	3.42%	10.72%	0.10%	0.00%	1.97%	0.00	3.39%	10.16%	0.02%
All	2.40%	9.28%	1.28%	2.79%	12.04%	1.18%	2.24%	8.36%	15.99%

TABLE 6: Percentage of system calls perturbed running benign software with different thresholds in the uncertain environment using *intrusive* strategies.

	Percentage of all syscalls perturbed			Percentage of connection-related calls perturbed			Percentage of buffer-related calls perturbed		
	10%	50%	dynamic	10%	50%	dynamic	10%	50%	dynamic
IO	1.21%	5.62%	8.20%	3.57%	20.14%	2.50%	0.95%	3.88%	50.93%
CPU	3.34%	11.82%	2.40%	0.03%	2.01%	1.00%	3.49%	12.27%	8.10%
All	2.34%	8.91%	5.13%	1.69%	10.53%	1.71%	2.30%	8.33%	28.21%

train, and *ref* level for SPEC CPU2006, and first, middle-most, and last-level in the Phoronix Test Suite. On all three different workloads, our results showed that two benign software were adversely affected by non-intrusive strategies and nine software were affected by intrusive strategies (see Table 7). Further, there were no significant changes on the percentages of total system calls perturbed, connection-related system calls perturbed and buffer-related system calls perturbed with the change of workloads for both types of perturbation strategies. The results indicate that the workload type of the tested software does not impact the program outcome in the uncertain environment for the two sets of perturbation strategies we used.

4.4 Effects of Uncertainty on Application Execution

In this section, we describe how CHAMELEON can leverage random perturbations to thwart malware samples, from poorly-programmed to evasive ones, and also to aid the discovery of bugs in benign software.

4.4.1 Effects of Uncertainty on Malware

Black Vine We simulated a watering hole attack similar to the *Black Vine* malware from Symantec [1]. This attack has three main components: a Trojan, a backdoor and a keylogger. First, the attacker sends a 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 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 [17]. The backdoor script uses scp for sending the data to the C&C server.

From the system call traces we collected, the first malicious behavior occurred when the backdoor was being configured, with a `sys_write()` invoked with a buffer parameter starting with \177ELF. This behavior caused the threshold to increase to t_{A1} on the `sys_write()` system call. Later, three pairs of `sys_dup2()` with file descriptors 0 and 1 are invoked afterwards to execute the backdoor. The threshold on the three `sys_dup2()` was increased to t_{A2}. Then, when `sys_read()` on the ssh host key files was invoked, the threshold decreased to t_{d}. Finally, the keylogger started, `sys_write()` was invoked to write to a log file and `sys_connect()` and `sys_sendto()` were invoked for the backdoor to communicate with the C&C server. The probability for the simulated malware to gain privilege and

TABLE 7: Impact of non-intrusive and intrusive strategies on 26 benign software from Phoronix Test Suite and SPEC CPU for different workloads in the uncertain environment (static threshold 10%).

Workload	Percentage of syscalls perturbed		Percentage of connection-related syscalls perturbed		Percentage of buffer-related syscalls perturbed		Number of Crashes	
	Non-intrusive	Intrusive	Non-intrusive	Intrusive	Non-intrusive	Intrusive	Non-intrusive	Intrusive
Light	4.3%	5.1%	0.0%	0.1%	2.9%	4.2%	2	9
Medium	5.8%	6.3%	0.2%	0.3%	3.1%	3.7%	2	9
Heavy	5.2%	5.9%	0.2%	0.2%	3.5%	3.0%	2	9

exfiltrate data is under $(1 - t_{\mathrm{A1}}) \times (1 - t_{\mathrm{A2}})^3 \times (1 - t_{\mathrm{d}}\%)$, which is 0.14%. In our 15 experiments, nine crashed before setting up the backdoor, four crashed before starting the keylogger, and two crashed before communicating with the C&C server.

Poorly-written Malware. We also evaluated a poorly-programmed malware sample [69], which splits its infection operation into two threads: the first (Code 1) gets access permissions to a given directory whereas the second waits for access permissions to infect the files in the directory (Code 2).

```
get_permissions();
finish();
```

Code 1: Thread 1

```
sleep(TIME);
infect();
```

Code 2: Thread 2.

The threads are synchronized using a sleep call instead of a system lock, which can be considered a poor programming practice, because different systems might have different sleep times. While running in CHAMELEON's uncertain environment, unexpected sleep responses may wake up `Thread 2` before accesses permissions are granted by `Thread 1`, thus causing the malware to fail. In our 15 tests, the sample failed 12 times because of the short sleep time (CHAMELEON's perturbations caused a decrease in the sleep time). In the three times the sample got permissions, it was mitigated by the increased perturbation threshold due to the frequent invocation of a system call (the sample iterated the `/proc` folder to find process identifiers).

Evasive Samples Advanced malware use a number of evasive techniques to avoid being detected while executing in sandboxes and/or antivirus (AV) emulators. A popular evasion method is execution stalling, i.e., malware postpones its malicious behavior until the sandbox times out [71]. Execution stalling techniques were identified in up to 64% of all Linux samples considered in a longitudinal study [46]. Malware samples stall their executions by employing distinct techniques, such as:

1) *Stalling Strategy 1*: malware performs a configurable and random long sleep before and during exhibiting its malicious behavior aiming to trigger sandbox execution timeouts.
2) *Stalling Strategy 2*: malware performs a great number of memory allocations (e.g. `malloc`) to slow down its operation while running under an emulator that traps memory accesses.
3) *Stalling Strategy 3*: malware delays its execution by performing long and costly computations, such as float point-based operations—as emulating float-based operations is hard and often results in sandboxes timeouts.

We studied stalling tactics by armoring a malware sample with these three evasive techniques independently investigated how CHAMELEON thwarts their operations. The

TABLE 8: Malware with evasive techniques running in the standard OS and in CHAMELEON.

Evasive Strategies	Percentage of syscalls perturbed	Percentage of buffer bytes perturbed	Execution results in CHAMELEON
Stalling 1	3.07%	0.07%	Operation not permitted
Stalling 2	2.31%	0.68%	Segmentation Fault
Stalling 3	3.02%	0.46%	Operation not permitted

malware sample was chosen as a challenging case because it would not set up any backdoor, nor would it inject malicious payloads to other executables, and most of the files the sample accessed were under the corruption protection mechanism. The sample would: i) fingerprint the environment to get an infection identifier; ii) install a proxy to infect the browser; iii) infect the DNS subsystem modifying the default routes; and iv) finally remove all the logs generated. The analyzed sample would not set up any backdoor, nor would it inject malicious payloads to other executables.

We ran these three different scenarios using the three different stalling techniques in both the standard environment and under CHAMELEON. In the standard environment, the malware sample could effectively exhibit its malicious behavior under all three scenarios. In CHAMELEON, configured with a dynamic threshold and non-intrusive strategies, the sample failed to achieve its goal. We measured how each stalling strategy was perturbed and summarized the results in Table 8. CHAMELEON silenced approximately 3% of system calls in the perturbation set, and perturbed less than 1% of buffer bytes. The percentage is small because of the corruption protection mechanism, which prevented many perturbations. For all three evasive scenarios, CHAMELEON mitigated the malware through either crash or segmentation fault. With the `Stalling Strategy 1` and `Stalling Strategy 3`, the malware sample terminated earlier (compared to executions in the standard environment), due to the perturbation on the `read` system call. In the user space, the malware sample was deceived with an error message "Operation not permitted in accessing file". With `Stalling Strategy 2`, the malware sample terminated with a segmentation fault due to the perturbation on `mmap` system call. The malware saw the memory allocation failure and terminated without executing malicious behavior, as the malware assumed it was running under an emulator. Notice that these samples successfully executed outside of CHAMELEON environment, thus the imposed perturbations are the only responsible for their unsuccessful execution. The OS-related error messages are due to CHAMELEON's perturbations messing with malware internal states, thus resulting in invalid requests.

4.5 CHAMELEON evolves over time

Since CHAMELEON has demonstrated its potential in malware mitigation, we further discuss whether CHAMELEON can stand the test of time by extending the malware sample set and testing them on newer version systems. The extended set contains 5,679 Linux malware samples collected from Virustotal [34] and VirusShare [33]. In this set, 273 samples can successfully attack our new testing systems, running Linux Ubuntu 4.15.0-20-generic version (Ubuntu 18.04).

We carried out the experiments with per-system call threshold and randomly pick non-intrusive and intrusive strategies. The new results showed that 89.38% of the malware samples are mitigated, with 43.59% of the malware hampered and 45.79% crashed. Compared with prior experiments on Ubuntu 12.04 (six years before the new testing system), which mitigated around 90% of the malware (Figure 3), the new results corroborated the old ones. It is also safe to say that CHAMELEON can evolve with malware over time.

Further, we measured the detection rates of existing Anti-virus software on this extended set through Virustotal. The top 10 highest detection rates (ranging from 65% -85%) with their company names are listed in Table 9.

TABLE 9: **TOP-10 AV Detection.** The best rate of 85% is ssmaller than the achieved by CHAMELEON.

AV	Detection (%)	AV	Detection (%)
Ikarus	85.61%	AVG	70.58%
Kaspersky	79.83%	Symantec	69.80%
ESET-NOD32	78.81%	TrendMicro-HouseCall	68.61%
GData	73.16%	Sophos	67.96%
Avast	72.99%	Qihoo-360	64.37%

4.5.1 Effects of Uncertainty on Benign Software

TABLE 10: Software bugs found by CHAMELEON

Software	Bugs
Vim	viminfo: Illegal starting char [32]
tar	Fail using '-C' option extracting archive with empty directories [25]
	"Operation not permitted" when extracting [26]
Thunderbird	Unable to locate mail spool file [30]
	segmentation fault (core dumped) [28]
Firefox	Bus error (core dumped) [10]
	Fatal IO error (Operation not permitted) on X server [11]

CHAMELEON's perturbations also affect benign applications. Whereas we observed that common benign applications can handle the unexpected system call responses in most of all experiments, we identified some crashes during their execution. Therefore, to better understand CHAMELEON's impact and to identify how benign software could improve to better adapt to the perturbations, we *manually* inspected all crashed execution traces.

We analyzed the execution logs, from the last system call executed (including its parameters) until the first system call executed in a reversed order. We observed that usually the failure of one system call with a specific parameter would lead to application early termination. Therefore, locating the corresponding system call causing the crash and its corresponding parameters would reveal the reason for the crash, and could potentially help the process of finding bugs. CHAMELEON is capable of perturbing every system call with a probability (given by the perturbation threshold), and logging the execution details (perturbations, system calls invoked and their parameters) about the crash.

During our analysis, we found that the crashes in Vim, tar, Mozilla Firefox and Thunderbird were in fact software bugs previously reported on Launchpad and Bugzilla [15, 4]. Because each system call was perturbed with a probability, the perturbations causing the crash in different tests varied on the same software (we ran each software fifteen times and averaged the results). Therefore, different bugs could be found for one piece of software. Table 10 lists the bugs in detail. Besides general bugs, e.g. Segmentation Fault, Fatal I/O error and Bus error, we found several bugs of particular interest.

.viminfo [32]: This bug causes Vim to fail to launch because of an erroneous .viminfo file. The .viminfo file is used to record information about the last edits from a user. If the user exits Vim and later re-starts it, the .viminfo file enables the user to continue where he left it off [31]. In our experiment, the .viminfo error was caused by silencing a `sys_write` on .viminfo file. In the reported bug, the error was caused by an operation using a special character not recognized by Vim before exiting. With CHAMELEON, we identified that the reason for Vim stopping launching is the failure of `sys_open` on .viminfo. This shows the lack of fail-safe defaults from Vim.

tar -C empty directory [25]: This bug occurs when one extracts empty directories inside an archive using the '-C' option to change directories. The cause for the bug is tar using `mkdir (file_name, mode)` instead of `mkdirat (chdir_fd, file_name, mode)` to extract a directory. With CHAMELEON, we identified the failure of creating a new file descriptor with `sys_openat` in our log file, showing that tar currently do not handle failures on that particular invocation of the system call well.

Thunderbird mail spool file [30]: The bug causes Thunderbird to hang when linking an existing email account. Thunderbird uses the spool file to "help" the user set up an email account with the assumption that the email providers set up SMTP, ports, and security configurations very well. Unfortunately, few of them are correctly configured [29]. From the log files of CHAMELEON, we identified that the failure in linking an account was caused by a failure in `sys_read` of spool file.

Our results show that the crashes and adverse effects in the analyzed benign software were in many cases actually caused by bugs (previously reported) instead of the perturbations applied by CHAMELEON. It appears that the perturbations just accelerated the exposure of such bugs, thus, showing that CHAMELEON could be also potentially applied to test software reliability.

5 DISCUSSION

This section will discuss about the insights and limitations based on CHAMELEON's findings.

5.1 Defensive Programming

The adoption of defensive programming for malware writing increases its development costs both in human and in time resources. The paradigm of defensive programming is based on the assumption that runtime errors in software are going to arise for a variety of reasons (including a potentially malicious OS) and that software needs to be written so as to be resilient to such errors [7]. This requires software developers (or malware writers for that matter) to include assertions in programs at runtime, and to write tests suites that anticipate different error scenarios and discover unknown behaviors.

Upon encountering invalid inputs, it is advantageous for malware to fail early and quickly to prevent leaving fingerprint of their actions in the system. Therefore, it is plausible to hypothesize that, under these conditions, malware writers would need to write more reliable code so as to successfully operate under runtime uncertain conditions.

Currently, a great number of malware source code found in the wild exhibit poor programming practices, such as lack of error checking routines. Even more noticeable, many malware implementations are flawed, as seen on numerous ransomware decryption keys retrieved via exploitation of flawed implemented routines [80, 44, 54, 72, 83].

We acknowledge that the adoption of a paradigm such as CHAMELEON could further incentivize malware writers into adopting defensive programming and, consequently, writing "resilient" malware. However, this would come under higher costs for malware writers.

Defensive programming might seem difficult to grasp in the beginning, like any new concept in software development. In fact, this concept has been well established and many programming languages, such as D, have already supported pre- and post-conditions as fundamental parts of their syntax [14]. With the increasing need of defensive programming, programmers and testers can work on developing automated test suites to anticipate different error scenarios and discover unknown behaviors. In the era of cloud computing combined with the Internet of Things (IoT) and Artificial Intelligence (AI), more and more small services will interact with one another dynamically, making defensive programming essential to ensure composed system reliability. We hypothesize that defensive programming will become built-in design characteristics of software development.

5.2 Trade-off: Performance vs. Mitigation Effect

Strategy: There are trade-offs in selecting a perturbation strategy. Non-intrusive perturbation strategies that delay system calls, decrease the priority, or silence the system calls with error returns, aim to slow down program execution, potentially buying time for a deep learning model to operate. End users might experience a system slowdown in exchange for more security. Intrusive perturbation strategies, which are more aggressive, are designed for organizations with higher security expectations.

Process Delay strategy is different from suspending software execution. A suspended execution stops suspicious software from running and would not generate data for a potential deep learning analysis. *Process Delay*, actually, slows down software execution, potentially buying time for deep analysis and allowing for a more accurate classification of software which received borderline confidence levels in classifications by a fast conventional machine learning detector. Moreover, suspension of execution can be detected by malware just by checking wall clock time.

Threshold: We acknowledge that the degree of uncertainty is not a one-size-fits-all solution. Based on the needs of the organization and its applications, CHAMELEON's perturbation threshold can be adjusted by a system administrator according to the organization's expectations and requirements. Based on our manual analysis, an acceptable static threshold is no more than 50% and a dynamic threshold is no more than 95%. The perturbation threshold can also be automatically adjusted following security policies. For example, CHAMELEON can raise the threshold for a given application if it passes a round of deep learning classification.

Moreover, CHAMELEON can leverage an attribution-based scheme for initial threshold assignment. If the software has no attribution of origin, no supply chain, or its origin is not trusted, CHAMELEON can set a higher default threshold for the application. Conversely, if the application is trusted, CHAMELEON can set a lower default threshold for it. We leave such approaches for future work.

Comparison with existing AV: As Table 9 shows, the best performed AV software produces a smaller detection rate than CHAMELEON. In addition, CHAMELEON complements existing AV software, such as FireEye [19], whose goals are either to identify the signature or monitor the runtime behavior through isolating the software for a while until a decision can be made. However, some software may not exhibit its malicious behavior at the very beginning, or confuse the AV software by demonstrating benign behaviors for most of the time.

CHAMELEON enables lifelong execution for software whose behaviors are hard to define and which cause borderline classification decisions. During the software execution, suspicious software will be placed in the uncertain environment when some borderline malicious behavior is detected, and be transferred between the standard environment and the uncertain environment multiple times.

Limitations: CHAMELEON is limited in mitigating well-written malware. Highly fault-tolerant malware will have a chance to succeed CHAMELEON. Further, if an administrator decides to whitelist common benign software from CHAMELEON, in-memory-only attacks [40] that inject themselves in the address space of a benign software will also succeed.

5.3 Linux vs. Windows

Linux Malware: Linux promotes open source code and streamlines prototyping. This makes CHAMELEON to be deployed in Linux because of its resources in kernel programming. Linux also makes it easier for manual analysis at the beginning to understand the malware behavior However, finding Linux-based malware samples is challenging. First, the availability of Linux malware samples is more restricted in comparison to other platforms, such as Windows. For instance, the Cozzi et al's work [46], the largest and most current Linux malware study, leveraged 10,000 malware samples whereas Windows studies may encompass million samples [51]. Second, Linux malware is distributed among multiple architectures, as the Linux environment itself, which also limits the number of available samples for a particular platforma (e.g., x86 desktops). As an example, only 30% of all samples considered in Cozzi et al's study [46] were x86 desktop malware samples. Finally, Linux malware is not only distributed in a self-contained form, as most Windows executables, but also as shared objects and payloads, which must be executed with a proper loader or hosting environment for its injection.

In CHAMELEON, we reverse-engineered all collected samples to understand their loading requirements and library dependencies. We installed all required library (in their outdated versions) and provided configuration files for every sample to ensure a successful execution. From Cozzi's x86 desktop malware samples [46], we successfully provided a suitable execution environment for 100 samples, and selected them for analysis. Even though this number is small especially compared to malware studies in Windows, we ensured that these samples were diverse, fully demonstrated their malicious behaviors, and allowed the evaluation of CHAMELEON's proof-of-concept prototype.

For future work, a usability study in CHAMELEON with a variety of benign software is warranted. We also plan to evaluate CHAMELEON with more diverse threats. For instance, we plan to extend CHAMELEON protection scope to cover in-memory only attacks [39] by adding support to memory-related system calls interposition and memory snapshots acquisition, and also to detect behaviors of frequent reads and writes into the Windows export table, required by in-memory-only malware to resolve imports and exports and get the malicious payload to execute.

CHAMELEON in Windows: Porting CHAMELEON Windows is necessary because Windows system is the most targeted OS by malware writers. Windows and Linux system calls are different, while there are some correspondences between the two systems [21], which are relevant for a future implementation of CHAMELEON in Windows. For example, for process control, Windows has `CreateProcess`, `ExitProcess` and `WaitForSingleObject`, while Linux has `fork`, `exit` and `wait`. For file manipulation, Windows has `CreateFile`, `ReadFile` and `WriteFile`, while Linux has `open`, `read` and `write`.

System call hooking in Windows can be implemented similarly to what we did in Linux for CHAMELEON by leveraging a driver that hooks the System Service Dispatch Table (SSDT), a task still allowed in 32-bit systems, but made challenging in 64-bit kernels because of the Kernel Patch Protection (KPP) mechanism. Therefore, CHAMELEON's implementation in 64-bit kernels would require Windows support. Alternatively, CHAMELEON could be implemented in Windows by relying on OS callbacks and filters, a solution adopted by newer sandbox solutions [42]. For instance, a file-system filter provides a pre-operation callback which allows one to deny access to a given file right before an access attempt, thus achieving the same goal of introducing uncertainty to OS operations. In Windows, the `EPROCESS` [62] is an equivalent structure to the Linux `task_struct`. Therefore, CHAMELEON can be implemented in Windows by adding the four fields mentioned earlier in Section 3.3 (the environment, the file descriptor list, the strategies and the threshold) to `EPROCESS` to control the process environment. However, this modification requires access to Windows source code and must be performed in-house.

Future work porting CHAMELEON to Windows OS and performing an evaluation with a larger sample size of malware is warranted.

5.4 CHAMELEON evolves over time

Since CHAMELEON is capable of disturbing malware execution and discovering benign software bugs, the next question becomes whether this solution could stand the test of time. We consider the decay of CHAMELEON are affected by three factors: malware detection scheme, system design and implementation, and configuration parameters.

Malware detection scheme: CHAMELEON targets for any hybrid detection scheme that works in a two-phase manner. Such detection scheme has been proposed more than a decade ago [45, 67], and is still active in the literature [88, 60]. Therefore, we believe the speed for CHAMELEON to decay because of the malware detection scheme is relatively slow.

System design: CHAMELEON by design is a loadable kernel module. As most kernel modules or drivers, e.g. USB drivers or Bluetooth drivers, CHAMELEON requires to be updated when operating systems upgrade. For example, when `sys_call_table` is no longer exported, a driver will have to add the `kallsyms_lookup_name` function; when `set_memory_rw` is no longer exported, a driver will have to implement its own. As these new functions usually require fewer than 5 lines of code, it is safe to say CHAMELEON will not decay fast because of the system design.

Configuration parameters: CHAMELEON uses the `threshold` parameter to determine the strength of the uncertainties. As malware and benign applications evolve over time, the `threshold` parameter requires to be fine tuned by the system administrator. In the future, the fine tuning can be done more efficiently with the advance of machine learning algorithms. Given the needs of organizations and companies, the machine learning algorithms can provide the most suitable set of parameters. Therefore, CHAMELEON is able to stand the test of time by integrating advanced algorithms.

As the result from the new experiments shows, CHAMELEON is able to survive from systems ranging from Ubuntu 12.04 to Ubuntu 18.04.

6 RELATED WORK

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

Malware Detection Techniques have been evolving from static, signature-based approaches [82] to dynamic, behavior-based techniques [49, 45]. Whereas the first may be defeated by code obfuscation and malware variants, the latter overcome these issues by continuously monitoring binaries execution, either at API [85, 73] or system-call [41] levels. Dynamic solutions are able, for instance, to detect sensitive data leaking via system-level taint tracking [86] and keystroke logging via data-flow analysis [61]. In this work, we leveraged the knowledge developed by previous dynamic malware detection solutions to implement CHAMELEON's API monitoring modules.

To detect malware, the data collected during dynamic analysis procedures is often modelled as behaviors and these are used as input for some decision algorithm. Machine learning-based approaches has been leveraged for behavior modelling and decision with reasonable results. Kumar et al. used K-means clustering [57] to differentiate legitimate and malicious behaviors based on the NSL-KDD dataset. Abed et al. used bags of system calls to detect malicious applications in Linux containers [35]. Mohaisen et al. [64] introduced AMAL to dynamically analyze malware using SVM, linear regression, classification trees, and kNN.

Behaviors modelling, however, has become challenging as applications are becoming increasingly diverse [58], which raises false positive rates. In this scenario and as alternative for machine-learning, recent efforts to apply DL for malware detection have made great successes. Pascanu et al. [66] used recurrent neural networks and echo state networks to model API system calls and C run-time library calls, and achieved accurate results. Li et al. leveraged an AutoEncoder and a deep belief network on the now outdated KDD99 dataset, and achieved a higher detection rate [60]. As a drawback, current DL-based malware detectors work in an offline manner due to the long detection time and large computation resource needed. Therefore, CHAMELEON emerges as an alternative to bridge the gap between the efficiency of ML classifiers and the effectiveness of DL classifiers while monitoring binaries execution in real time.

Most of the malware detection solutions were first implemented as software components, such as using patched libraries or implementing kernel hooks, a strategy also followed by CHAMELEON. Recently, hardware-based approaches such as Virtual Machine-powered solutions [47, 2] emerged as alternatives for system monitoring without requiring patching. Whereas these approaches cannot be considered practical due to the need of developing a hypervisor, it opens opportunity for the development of an unobtrusive CHAMELEON's implementation in the future.

Deception: To a limited extent, deception has been an implicit technique for cyber warfare and defense, but is under-studied as a fundamental abstraction for secure systems. Honeypots and honeynets [74] 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 [53]. 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 [38] 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* [65], a model of deception that challenges the cornerstone of projecting false reality with internal consistency. Sun*et al.* [77, 76] also argued for the value of unpredictability and deception as OS features. CHAMELEON explored non-intrusive unpredictable interferences to create an uncertain environment for software being deep analyzed after an initial borderline classification.

Fuzzing: Fuzzing 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. Fuzzers can be divided into two categories: generational fuzzers, that construct inputs according to some provided format specification (e.g. SPIKE [37] and PEACH [48]), and mutational fuzzers, that create inputs by randomly mutating analyst-provided or randomly-generated seeds.(e.g. AFL [87], honggfuzz [79], and zzuf [52]). Generational fuzzing requires significant manual effort to create test cases and therefore is hard to be scalable. Most of recent fuzzers are based on mutational fuzzers [68]. CHAMELEON is a mutational fuzzer that randomly applies perturbations to invoked system calls during software execution.

Fuzz testing leveraging system call behaviors has shown its potential in scalability and effectiveness. Trinity [16], for example, randomizes system call parameters to test the validation of file descriptors, and found real bugs [3], including bugs in the Linux kernel. BALLISTA [55] 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 considered as a fuzz tester at the OS system call API to understand how resilient an application is to a particular type of misbehavior. KLEE [43] uses system call behaviors to build a model and generate high-coverage test cases to the users, and this motivated following work in coverage guided fuzzers, such as AFL [87], honggfuzz [79], and zzuf [52], which use coverage as feedback from the target program to guide the mutational algorithm to generate inputs. While CHAMELEON's goal is not to find software bugs, CHAMELEON can borrow this idea by keeping track of a set of interesting perturbations that triggered new code paths and focus on mutating the interesting inputs while generating new perturbations.

7 CONCLUSION

This work introduces a detailed description of the design and implementation, and new extensions of CHAMELEON, a Linux framework which allows the introduction of uncertainty as an OS built-in feature to rate-limit the execution of possible malware. CHAMELEON allows the scheme of two-phase malware detection during software lifelong execution. After the first phase makes a borderline detection, potential malware will be disturbed when the second phase detection is under way.

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 from the first phase detection, where a set of perturbations will be included.

We evaluated CHAMELEON by first manually analyzing the execution of 113 common applications and 100 malware samples from various categories. The results showed that a dynamic, per-system call threshold caused various levels of disruption to only 10% of the analyzed benign software. The effects of the uncertain environment in malware was more pronounced with 92% our studied malware samples failing to accomplish their tasks. Compared to the results obtained for a static threshold, 20% more benign software succeeded and 24% more malware crashed or were hampered in the uncertain environment. The results were then further corroborated by an extended dataset (5,679 Linux malware samples) on a newer system.

We also analyzed the behavior of crashed benign software, and found that many of the crashes were actually caused by software bugs. Several bugs were reproduced for Vim, tar, Mozilla Firefox and Thunderbird.

Besides filling the gap between two-phase malware detection scheme, CHAMELEON increases attackers' work factor. The effort of writing good and small malware is not lengthy endeavor. However, in newer system today, it is hard to use a single small malware to bypass all the protection mechanisms. The goal of CHAMELEON is to have attackers spend at least the same effort as software engineers.

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.

ACKNOWLEDGMENTS

We thank NSF CNS-1464801, CNS-1228839, CNS-1161541, DGE-1303211, ACI-1229576, CNS-1624782, US-Brazil Workshop on Cyber Security and Privacy, and Brazilian CNPq.

REFERENCES

[1] The black vine cyberespionage group http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-black-vine-cyberespionage-group.pdf.
[2] Bromium end point protection https://www.bromium.com/.
[3] Bugs found by trinity.
[4] Bugzilla.
[5] Code – fu: The art of defensive programming https://sov.tech/code-fu-defensive-programming/.
[6] Coverage.py https://github.com/msabramo/coverage.py.
[7] Defensive programming - swc osg workshop.
[8] Dionaea - a malware analysis honeypot http://www.edgis-security.org/honeypot/dionaea/.
[9] Emma: a free java code coverage tool http://emma.sourceforge.net/.
[10] Firefox:Bus error (core dumped).
[11] Firefox:fatal io error 11 (resource temporarily unavailable) on x server :0.
[12] Gcov https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[13] Gnu project http://www.gnu.org/software/software.html.
[14] An in-depth look at defensive programming https://www.zuehlke.com/blog/an-in-depth-look-at-defensive-programming/.
[15] Launchpad.net.
[16] Lca: The trinity fuzz tester.
[17] Logkeys ubuntu http://packages.ubuntu.com/precise/admin/logkeys.
[18] McAfee Virtual Criminology Report - 2009.
[19] Modern malware exposed http://www.nle.com/literature/FireEye_modern_malware_exposed.pdf.
[20] The modern malware review http://media.paloaltonetworks.com/documents/The-Modern-Malware-Review-March-2013.pdf.
[21] One-to-one correspondence of windows and linux system calls http://www.cs.columbia.edu/~jae/4118-LAST/L02-intro2-osc-ch2.pdf.
[22] The phoronix test suite http://www.phoronix-test-suite.com/.
[23] Spec cpu 2006 https://www.spec.org/cpu2006/.
[24] Symantec Internet Security Threat Report 2013. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf.
[25] tar:fails using '-c' option extracting archive with empty directories.
[26] tar:operation not permitted.
[27] Thc: the hacker's choice https://www.thc.org/.
[28] Thunderbird crashes with segmentation fault.
[29] Thunderbird spool file https://support.mozilla.org/en-US/questions/1021171.
[30] Thunderbird:unable to locate mail spool file.
[31] Viminfo documentation http://vimdoc.sourceforge.net/htmldoc/starting.html#viminfo-file.
[32] viminfo:illegal starting char in line.
[33] Virusshare https://virusshare.com/.
[34] Virustotal https://www.virustotal.com/.
[35] Amr S Abed, T Charles Clancy, and David S Levy. Applying bag of system calls for anomalous behavior detection of applications in linux containers. In *2015 IEEE Globecom Workshops*, pages 1–5. IEEE, 2015.
[36] National Security Agency. Security-enhanced linux.
[37] Dave Aitel. An introduction to spike, the fuzzer creation kit. *presentation slides), Aug*, 1, 2002.
[38] Mohammed H. Almeshekah and Eugene H. Spafford. Planning and integrating deception into computer security defenses. In *New Security Paradigms Workshop*, 2014.
[39] Meisam Navaki Aref, Geoffrey Alexander, Hooman Rokham, Aokun Chen, Michalis Faloutsos, Xuetao Wei, Daniela Seabra Oliveira, and Jedidiah R. Crandall. Faros: Illuminating in-memory injection attacks via provenance-based whole-system dynamic information flow tracking. In *DSN 18*, 2018.
[40] Meisam Navaki Arefi, Geoffrey Alexander, Hooman Rokham, Aokun Chen, Michalis Faloutsos, Xuetao Wei, Daniela Seabra Oliveira, and Jedidiah R Crandall. Faros: Illuminating in-memory injection attacks via provenance-based whole-system dynamic information flow tracking. In *International Conference on Dependable Systems and Networks (DSN)*, pages 231–242. IEEE, 2018.
[41] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. Ttanalyze: A tool for analyzing malware. In *15th European Institute for Computer Antivirus Research (EICAR)*, 2006.
[42] Marcus Felipe Botacin, Paulo Lício de Geus, and André Ricardo Abed Grégio. The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 14(1):87–98, Feb 2018.
[43] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation*, 2008.
[44] Carbonite. Ransomware developers learn from the mistakes of wannacry, notpetya. https://www.carbonite.com/blog/article/2017/10/ransomware-developers-learn-from-the-mistakes-of-wannacry-notpetya/, 2017.
[45] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2005.
[46] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In *IEEE Symposium on Security & Privacy*, 2018.
[47] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security*, 2008.
[48] Michael Eddington. Peach fuzzing platform. *Peach Fuzzer*, page 34, 2011.
[49] 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*, pages 120–128, 1996.
[50] Tal Garfinkel et al. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, volume 3, pages 163–176, 2003.

[51] William Hardy, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin Li. Dl4md: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Mining (DMIN)*, page 61. The Steering Committee of The World Congress in Computer Science, 2016.
[52] S Hocevar. zzuf—multi-purpose fuzzer, 2011.
[53] N. Rowe 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.
[54] Kaspersky. A flawed ransomware encryptor. https://securelist.com/a-flawed-ransomware-encryptor/69481/, 2015.
[55] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *The Sixteenth Symposium on Reliable Distributed Systems*. IEEE, 1997.
[56] Sandeep Kumar and Eugene H. Spafford. An application of pattern matching in intrusion detection, 1994.
[57] Vipin Kumar, Himadri Chauhan, and Dheeraj Panwar. K-means clustering approach to analyze nsl-kdd intrusion detection dataset. *International Journal of Soft*, 2013.
[58] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: Using system-centric models for malware protection. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 399–412, 2010.
[59] Lhall. Introduction to writing shellcode.
[60] Yuancheng Li, Rong Ma, and Runhai Jiao. A hybrid malicious code detection method based on deep learning. *methods*, 9(5), 2015.
[61] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 78–97, 2008.
[62] Microsoft. Windows epblock.
[63] MMD-0027-2014. Linux/bashdoor & small backdoor - at 0day (shellshock): The fun has only just begun (part 1).
[64] Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen. Amal: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, 52:251 – 266, 2015.
[65] Vicentiu Neagoe and Matt Bishop. Inconsistency in deception for defense. In *New Security Paradigms Workshop*, pages 31–38, 2007.
[66] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1916–1920, 2015.
[67] Udo Payer, Peter Teufl, and Mario Lamberger. Hybrid engine for polymorphic shellcode detection. In *DIMVA, 2005*.
[68] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.
[69] Brian Prince. Script fragmentation attack could allow hackers to dodge anti-virus detection, 2018.
[70] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.
[71] Bert Rankin. Evasive malware – the art of doing nothing, Dec 19, 2016.
[72] Softpedia. Truecrypter ransomware dev leaves flaw in code that lets victims decrypt files. https://news.softpedia.com/news/truecrypter-ransomware-dev-leaves-flaw-in-code-that-lets-victims-decrypt-files-503537.shtml, 2016.
[73] Norman Solutions. Norman sandbox whitepaper, 2003.
[74] L. Spitzner. *Honeypots: Tracking Hackers*. Addison Wesley Reading.
[75] R. Sun, X. Yuan, A. Lee, M. Bishop, D. E. Porter, X. Li, A. Gregio, and D. Oliveira. The dose makes the poison-leveraging uncertainty for effective malware detection. In *2017 IEEE Conference on Dependable and Secure Computing*, pages 123–130, 2017.
[76] Ruimin Sun, Andrew Lee, Aokun Chen, Donald E. Porter, Matt Bishop, and Daniela Oliveira. Bear: A framework for understanding application sensitivity to os (mis)behavior. In *ISSRE*, 2016.
[77] Ruimin Sun, Donald E. Porter, Daniela Oliveira, and Matt Bishop. The case for less preditable operating system behavior. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.
[78] Ruimin Sun, Xiaoyong Yuan, Pan He, Qile Zhu, Aokun Chen, Andre Gregio, Daniela Oliveira, and Xiaolin Li. Learning fast and slow: Propedeutica for real-time malware detection. *arXiv preprint arXiv:1712.01145*, 2017.
[79] Robert Swiecki. Honggfuzz. *Available online at: http://code.google.com/p/honggfuzz*, 2016.
[80] ThreatPost. Key flaw enables recovery of files encrypted by torrentlocker. https://threatpost.com/key-flaw-enables-recovery-of-files-encrypted-by-torrentlocker/108210/, 2014.
[81] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern linux api usage and compatibility: What to support when you're supporting. In *Eurosys*, 2016.
[82] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-Based Intrusion Detection Approach. ACSAC '98, 1998.
[83] WeLiveSecurity. Ransomware is everywhere, but even black hats make mistakes. https://www.welivesecurity.com/2016/04/28/ransomware-is-everywhere-but-even-black-hats-make-mistakes/, 2016.
[84] Sheng Wen, Wei Wu, and Aniello Castiglione. *Cyberspace Safety and Security: 9th International Symposium, CSS 2017, Xi'an China, October 23–25, 2017, Proceedings*, volume 10581. Springer, 2017.
[85] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. volume 5. IEEE, 2007.
[86] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. *ACM CCS 07*, pages 116–127, November 2007.
[87] M Zalewski. Online; accessed 17-may-2017. american fuzzy lop.
[88] Symantec Endpoint Security - Prevent breaches with modern endpoint security. https://www.symantec.com/products/endpoint.

Bibliography

Ruimin Sun is a Postdoctoral Research Associate at Northeastern University. She received her Ph.D. from University of Florida. She works on secure, reliable systems, and machine learning model protection.

Xiaoyong Yuan is a Ph.D student at University of Florida. His research fields include deep learning, information security, and cloud computing.

Marcus Botacin is a Ph.D. student at Federal University of Paraná (UFPR). His main research interests are malware analysis and hardware-assisted security.

Matt Bishop is a Professor at the University of California at Davis. His research focuses on election processes, data sanitization, and the insider problem. He is the author of *Computer Security: Art and Science*.

Donald E. Porter is an Associate Professor at University of North Carolina at Chapel Hill. Porter's research involve improving efficiency and security of computer systems.

André Grégio is an Assistant Professor at the Federal University of Paraná (UFPR). His research intersects computer security and data science, e.g., effective malware analysis and attack detection systems.

Xiaolin (Andy) Li is a Partner of Tongdun Technology, heading the AI Institute. He was a Professor at the University of Florida. His research interests include deep learning, cloud computing, security, and IoT.

Daniela Oliveira is an Associate Professor at University of Florida. Her research includes human factors security and IoT security, especially the application of dynamic information flow to thwart attacks.

Appendix

TABLE 11: List of the 113 benign software tested in our evaluation. Software adversely affected by non-intrusive strategies (*) and intrusive strategies (′) are marked. Software tested under different workloads are bold.

CPU-bound						**I/O-bound**				
specrand	**lbm**	h264ref	libquantum	**sjeng** ′	**diff**	sendxmpp	nslookup ′	netstat	tcpdump	nmap
soplex ′	**mcf**	**tar*** ′	namd	**gromacs**	cksum	sendmail	ls	ss	dig	ifconfig
omnetpp	**astar**	a2ps	**tail**	accton	spell	route	**ping**	arp	nano	pico
lastcomm	**head**	dump-acct	**sort**	cppi	ebizzy	lpstat	vim * ′	xte ′	echo	wget * ′
teseq	grep	gcal	gcal2txt	tcal	crafty	emacs′	mkdir	traceroute ′	truncate	nice
txt2gcal	**wdiff**	moe ′	screen ′	**h246ref**	c-ray	cut	service	df	du	host
find	paxutils	shar	unshar	uuencode	hmmer	firefox *	skype	thunderbird * ′	gedit	**fs-mark** ′
enscript	ad2c	libextractor	csv2rec	**python** ′	nero2d	**cp**	rm	ab	chrome	libreoffice
recdel	tar	recinf	**ruby**	**gcc**	mrbayes	surblhost	gurgle	gv	ctags	mkafmmap
javac ′	recfmt	**cyclictest** ′	**multichase**	himeno	blogbench	barcode	**iozone** ′			
dolfyn	encode-ogg	espeek	ffte	fhourstones	gcrypt					

TABLE 12: List of the 100 malware samples used in our evaluation. We named malware with their executable file name.

Malware Category	**Malware Name**	**Malware Name**
Flooders	VirusShare_0a6c05d448d41a549bf8949a41a8e4d3	VirusShare_0df5910e6e5f865fddf2d4a4911893fb
Flooders	VirusShare_1a39b759416597743a7357634cb29743	VirusShare_1cc96351edd803bdaf849978d3e6c1cf
Flooders	VirusShare_1d254d60fc8c588e3ad23ea55e84af1e	VirusShare_1d57994e9ee7b308ea5f767dcd04195a
Flooders	VirusShare_1da85ad45cb7e66738cdb0e050dca2e2	VirusShare_2b9125e77e18926fe6b99b93f79da92e
Flooders	VirusShare_6b174d94b2b20a506cfdd4074be6df05	VirusShare_47e6947dad6821745d9d24e31a894400
Flooders	VirusShare_25689d63d0476435e752c9bf61bf2942	VirusShare_a3b5646f130a129edef7273606de8952
Flooders	VirusShare_b8f97d0ba7d21e5b08d98f32ccb97fec	VirusShare_e1bc6b6911feba3692579c771cc451e4
Flooders	VirusShare_edf4d6003c9c68774438e4fb25198dab	VirusShare_ffc7be26912b5aca63e55dc7c830f28a
Flooders	VirusShare_ff4dbe26278bfda759ee8b1f10d94d3b	VirusShare_b16de6aa853cab944503825e08cca9b3
Flooders	VirusShare_b31ae7e6de5da850f91bd4c9c4a47da0	VirusShare_b74a48a7555c6ae6a260b0a3ff7e6aa2
Flooders	VirusShare_b96ee50d33a6b376b67f257718e211f8	VirusShare_b367a540ef865acea0fb00d41c91f378
Spyware	VirusShare_0a07cf554c1a74ad974416f60916b78d	VirusShare_0b283a19a141030be3e8188896d9510b
Spyware	VirusShare_0d2dfefb9cfa7d082e9e0d13a28e9722	VirusShare_1dc810f0f0d905046caaad1ea6f79b0e
Spyware	VirusShare_00f7adbe9895699b07a114e383787c74	VirusShare_0d057b1b2d81728cb97f5484e9344fc0
Spyware	VirusShare_986f442fea7f98579e8a2b4a52f961ab	lkl
Spyware	logkeys	VirusShare_fd46acb36263b0155e644941a9e6f03a
Spyware	VirusShare_fe2df5014dd6f67dc15dffdba25ddd9d	VirusShare_b5c3730f4c373ea6cd9f8e770b332de6
Spyware	VirusShare_b7aedd8e6907acde2c2ca72ea18e1ac8	VirusShare_b67d0e7a6fdc712aded5bd1a64cfb3e0
Spyware	VirusShare_b790de9fc2921caa97be21236446d6bf	
Trojan	VirusShare_0b57ab2f37580a84219dd2faaa9f3444	VirusShare_1b7f7a6af703002c56754c826459e109
Trojan	VirusShare_01c5f86372a4f31e72675f8be9b4e6c7	VirusShare_1a6e2a1ebaa423ba2974c3e66f734f2c
Trojan	VirusShare_5a16c12e1abe11317465ea4032aa25aa	VirusShare_52be89c6e6b108b610dfe2cb67b9fe4e
Trojan	VirusShare_753d5e7af271c12e0803956dd8c2b8e6	VirusShare_bc3c6f56c85ecd1a7b8bbded84f7e6cd
Trojan	Botnet	VirusShare_e414088f88b329c99aac2ebdfa5aad23
Trojan	VirusShare_ef112ebf02f0ded52207eed236084cd9	ncrack
Trojan	Trojanit	httpd
Trojan	VirusShare_ff4bdef83f191cd6451e26a09311bcd2	VirusShare_fda2e5692426454eaf663f7684281955
Trojan	VirusShare_fdbfffa9bbc918a1b780e54249d3fc99	VirusShare_b2cc8e0741d07be0d34b4ea5cdd00f31
Trojan	VirusShare_b21c7770a6b16c166a1dee6eefcf68a1	VirusShare_b24ef5e799b9569377d8705d91bceb38
Trojan	VirusShare_b62f3df6160145643a2f30d635f2476c	VirusShare_b77e81d28c2585489e07ccdbec8eb883
Trojan	VirusShare_b94ab416758569167a54abef295c599b	VirusShare_b747e8639341958e9c172b6e0c973355
Viruses	VirusShare_0a4b022d6865dc32bb246c8b57aadb06	VirusShare_1c41538ccd680edfc0a5e36021fc37e5
Viruses	VirusShare_2c45f0f3dc02d8772f875cc5184459ec	VirusShare_1a46e25a5c3419f1dbc7b63b59053ab3
Viruses	VirusShare_3deec7f4fa618f6a97e7f7af33edb299	VirusShare_8b1ba12a6246829d774ccfdba27db0d6
Viruses	VirusShare_8b754b0219aad9bed5da083b9a034352	VirusShare_8b7521ab69a46902087af19455b21e19
Viruses	VirusShare_13ee81ea357a97f1d879b91c827a5629	VirusShare_56c7dcc249a715e05e5604d142d6d1e8
Viruses	VirusShare_217ece604b4e4a0f766c4ea8aa218519	VirusShare_522a4cdb1f05fdda5f390b31a95f7ae3
Viruses	VirusShare_7139ce345ffdacfc92d0cc70e8830320	VirusShare_79927828f9c2e7a015b94c59f4cfe2bb
Viruses	VirusShare_d19ad58b5807415b2e1cb3a503755c59	dataseg
Viruses	VirusShare_ed91cd156c154865859deac9e635acb0	VirusShare_f0e879988dd417dc02da7d5def6367cf
Viruses	VirusShare_f11ccc4acb53495a871b278c5efa6b98	VirusShare_fbf05500579ac4c597998d3359a76c42
Viruses	iamsick	manpage
Viruses	vlpi	VirusShare_fd40c417fb687341b8673f6de4e34aef
Viruses	VirusShare_b50ddaa7162db9817938af18940c81ab	
Worms	VirusShare_0a477a043a8b3deba999bbbfa1c32f47	VirusShare_0ba3e70816496e5f9d45912f9f15fb76
Worms	VirusShare_0cd70e3262a214fd104813826dd612c9	VirusShare_0e4cca1b162c3a9035214058f93a97b4
Worms	VirusShare_1d6fa5ed0080a0997f51a86697b8392c	VirusShare_3c2bd0548bf8c33566a5bda743441cf0
Worms	VirusShare_447bc42013537c5173e575cf0d166937	VirusShare_986f442fea7f98579e8a2b4a52f961ab
Worms	VirusShare_20719a9e850c07ac60d548a546ec0a7f	VirusShare_e8ff4cb488fea8e0ad78e8dc28ae884c
Worms	kaiowas10	ssl-crack
Worms	VirusShare_fe2b2560121db8d08d044bc2d579eac4	VirusShare_b4ba07c4d9b781635b33d485b73a614f