LEVERAGING UNCERTAINTY TO IMPROVE SYSTEM SECURITY AND RELIABILITY

By

RUIMIN SUN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2019

I dedicate this to people who see my worth and build my value.

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Dr. Daniela Oliveira for all her support during my PhD study. She is a great professor who provides prudent guidance throughout various research problems, and maintains a rather interacting, inspiring, and yet enjoyable atmosphere within work. I would also like to thank my committee members Dr. Xiaolin Andy Li, Dr. Kevin Butler and Dr. Tuba Yavuz for their technical guidance and valuable resource provided to make this dissertation to complete.

Additionally, I would like to thank the professors I collaborated with: Dr. Matt Bishop, Dr. Donald E. Porter, Dr. Andre Gregio, and Dr. Yier Jin. They supported me with their strong technical expertise and positive attitude towards tough research problems. They served as role models that are hard-working and modest. They are brilliant researchers with great patience.

I was lucky to work with a group of great colleagues: Dr. Nikolaos Sapountzis, Xiaoyong Yuan, Aokun Chen, Marcus Botacin, and Fabricio Ceschin. They were always willing to discuss about research problems and offer me valuable advice. My dissertation would not be interesting without the considerable amount of meetings and discussions over email, Skype, and Whatsapp.

I would like to thank as well as dedicate this dissertation to my family—my parents who always encourage me to overcome difficult problems and teach me optimistic attitude toward life, and my sister who relaxes me when I am feeling down. Finally, I would like to thank Yang Liu, who always trusts me, respects my choices, and has been doing the utmost to support me during the past seven years. Their constant support, love and boundless encouragement, significantly helped me towards completing this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

LEVERAGING UNCERTAINTY TO IMPROVE SYSTEM SECURITY AND RELIABILITY

By

Ruimin Sun

May 2019

Chair: Daniela Oliveira
Major: Electrical and Computer Engineering

Modern operating systems are designed to be predictable. This improves compatibility among different instances of the system, including older programs running on newer systems. But predictability raises the "monoculture" problem, allowing vulnerabilities that are exploitable on one system to be exploitable on all systems of the same type. On the other hand, because of the predictable design, benign applications are not well equipped to be resilient against unplanned unpredictability at the operating system (OS) level. This unpredictability can arise from different OS versions, network protocols, and APIs across different platforms, and can cause bugs and security vulnerabilities in applications.

This dissertation aims to investigate methods for leveraging unpredictability/uncertainty in the OS to improve system security against attacks and software resilience against OS faults. To address the issue, this dissertation introduces three pieces of work - BEAR, CHAMELEON and PROPEDEUTICA.

Bear is a framework for statistical analysis of application sensitivity to OS unpredictability, which can help developers discover challenging bugs and identify the scenarios that most need validation. The results provided insights for software developers in building more resilient software.

CHAMELEON is a novel OS paradigm that leverages uncertainty to protect systems against malware that cannot be detected by standard anti-malware approaches. It studies two environments for its processes: standard (for mission critical and approved software) and uncertain (for

unknown software including potential malware). In the uncertain environment, malware will run slower and will experience transient unavailability to needed resources, such as network connections and files. The results showed that the uncertain environment can disproportionately hurt malware more than benign software.

PROPEDEUTICA is a framework that can help CHAMELEON to determine which environment the software should be executed in. PROPEDEUTICA first classifies software as benign or malicious using fast machine learning models, and if the classification results are borderline, the software will go through more accurate but also more costly deep learning models. Results showed that PROPEDEUTICA is promising for real-time malware detection with high detection accuracy and acceptable false positive rates.

Finally, this dissertation concludes the accomplishments of these three frameworks and discusses promising future research directions that leverage uncertainty for system security and reliability.

CHAPTER 1
INTRODUCTION

1.1 Motivation Of The Dissertation

Predictability is a first-class system design goal. It simplifies application engineering and usability issues, such as compatibility among different versions of the system. The downside of predictability is computer system monoculture, where vulnerabilities become reliably exploitable on all systems of the same type. With a very small number of existing operating system kernels, libc implementations, and language runtimes deployed in practice, any predictable exploit will apply to a significant fraction of computers in the world. Current system with monoculture prevents system defenses to adapt well to new conditions, whereas motivated attackers have effectively unlimited time and resources to find and exploit weaknesses in computer systems.

The security community is aware that combating the computer system monoculture is the correct approach, with proposals such as software diversification [1–5], address space layout randomization (ASLR) [6], compiler specialization [7, 8], and ISA randomization [9]. System diversification is a limited form of unpredictability. The intent of diversity is independence, which means that multiple instances yield the same result, but in such a way that the only common factor is the input. One example is N-version programming [10], in which multiple teams create different software implementations to perform the same actions. The results are considered more reliable since the probability for all versions to share the same vulnerability is relatively low. A similar case is fault-tolerant system designs, which require sufficient software diversity so that faults are independent, and can be masked by voting or Byzantine protocols [11, 12].

Paradoxically, because of the predictability design, applications are not well prepared to endure unpredictability at the OS level that was not part of the OS specification. Recent research [13–15] has shown that bugs and security vulnerabilities were found in applications because of the natural unpredictabilities that our systems already have. These bugs or vulnerabilities can arise from: (i) different ways OSes handle network events and protocols [13], (ii) subtle and

undocumented differences in the behavior of common APIs across different platforms [14], and (iii) OS changes over time. Further, the OS can be buggy or malicious which further breaks the predictability assumption. For example, in Iago attacks [15] a malicious kernel induces a protected process to act against its interests by manipulating system call return values. These application bugs and vulnerabilities are hard to reproduce in a testing environment, and thus are difficult to detect and prevent. Although recent technology trends, such as Intel SGX, are attempting to introduce mutual distrust between the OS and its applications [16–19], such as by protecting the application's memory from the OS, a significant burden still falls to the application developer to reason about the semantic impact of return values from the OS on their applications.

To sum up, the effects of predictability in computer systems are two-fold. On one hand, our community strives to design systems that are consistent and predictable for the sake of the execution of benign applications, thus creating a monoculture that benefits attackers. On the other hand, our applications are not well prepared to be resilient against unplanned unpredictability at the OS level anyway.

This dissertation addresses the issue of system unpredictability from three aspects. First, given that applications need to withstand unplanned unpredictability at the OS level anyway and developers are simply not equipped to write robust applications in the face of unpredictable or even adversarial OSes, this dissertation introduced a statistical framework for the Linux OS to understand the application sensitivity to unpredictability at OS level (adversarial or not). This framework [20], called BEAR, statistically analyzes a program using a set of unpredictable strategies on a set of commonly used system calls, to discover the most impactful strategies, the most sensitive system calls for each application, and how they predict abnormal program outcome (crashes, segmentation faults, and etc). Rigorously understanding application sensitivity to OS misbehavior can help developers discover challenging bugs and edge cases, as well as identify the scenarios that most need end-to-end checks, targeted testing and verification procedures. Results showed that developers should be careful on handling buffer type and return length of OS system calls, considering how commonly used `read` and `write` system calls are,

especially when application workload is heavy. Developers and testers should balance their time not only based on how severe the impact of a system call or a strategy, but also on how frequently it is invoked. If resources are limited, failure-oblivious computing can be a promising way for saving developers and testers from memory bugs.

Second, this dissertation proposed CHAMELEON [21–24] that can provide three environments for process execution: (i) a diverse environment to protect benign software (ii) an uncertain environment to disturb unknown software, and (iii) a deceptive environment to analyze likely malware. In the diverse environment, implementations of the program APIs are randomized to reduce instances with the same combinations of vulnerable code. In the uncertain environment, a subset of the system calls have their parameters modified or silenced probabilistically, and the execution of processes may lose some I/O data and functionality. A malicious process in the uncertain environment will have difficulty accomplishing its tasks. The deceptive environment is similar with a honeypot, in which identified malware will have the system calls modified to deceive an adversary with a consistent but false appearance while forensic data is collected and forwarded to response teams such as CERT. This dissertation thoroughly studied the effectiveness of the uncertain environment with Linux malware and benign software samples, and the results showed that "uncertainty" can disproportionally hurt malware more than benign software.

Third, this dissertation presented PROPEDEUTICA [25], a Windows framework for malware detection that can be potentially combined with CHAMELEON. PROPEDEUTICA contains two phases in malware detection with the first phase in the standard environment and the second phase in the uncertain environment, and determines which environment a process should go through during the lifetime of the process. In practice, real-time malware detection is hard - conventional machine-learning (ML) based models suffer from high false-positive rates, while modern deep-learning (DL) models require more time to do an accurate detection. To address this problem, this dissertation introduced and evaluated PROPEDEUTICA, a novel methodology and framework for efficient and effective real-time malware detection. In Medicine, *propedeutics* refers to diagnosing a patient condition by first performing initial non-specialized, low-cost

exams, and only proceeding to specialized, possibly expensive and diagnostic procedures if preliminary exams are inconclusive. Similarly, PROPEDEUTICA first classifies a piece of software using fast conventional machine learning models. If the classification results are borderline, the software is subjected to more accurate but also more performance expensive deep learning models. For software subjected to deep learning analysis, delay strategies (a light perturbation from CHAMELEON) were added to the execution as a way of "buying time" for the most expensive deep learning models to operate. PROPEDEUTICA introduced a novel deep learning model DEEPMALWARE using n-gram embedding, (Atrous) Convolutional Layers, Long Short Term Memory (LSTM) layers and Fully Connected Layers. Through an experiment of more than 10,000 Windows malware samples collected from a real banking institution, PROPEDEUTICA achieved an accuracy of 95.54% and a false positive rate (fpr) of 4.10% (pure ML has a fpr of 20%) and reduced the detection time by 54.86% compared with pure DL methods.

1.2 Challenges And Related Work

There is much prior work in the literature on software diversity and deception, fuzz testing, and malware detection. In the following, we describe these works and their limitations.

1.2.1 Diversity And Deception

The ability to diversify behavior within a system is an essential building block for unpredictability. We define the distinction between diversity and unpredictability as whether the variations stay within the API specification. Several projects [7, 8] randomize the selection of instructions at compile time, breaking unnecessarily predictable sequences of potentially exploitable instructions. Each instance of a system binary will have different, but functionally equivalent instruction sequences. Compile-time techniques can improve diversity, but cannot adapt to changing attacks.

In addition to ASLR, several proposals have dynamically diversified other aspects of application behavior at runtime. Several projects mitigate buffer overflows and other memory errors by randomizing system call mappings, global library entry points, stack placement, stack direction, and heap placement—often in conjunction with running multiple versions in parallel

to detect divergence [1–5]. Holland *et al.*[9] proposed a strategy to randomize the ISA of a virtual environment, undermining portability of attacks leveraging low-level features, such as code injection attacks. The Synthetix project [26] specialized code dynamically using automatic compiler analysis and programmer annotations, primarily to improve performance; Program slicing has also been used to bound the cost and complexity of automatic diversification [27]. Finally, several projects have combined existing diverse implementations of file systems [28], databases [29], and language implementations [30]. As discussed above, dynamic diversity reduces predictability but is often limited to easily-randomized features of the software.

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 [31] are systems designed to look like production systems in order to deceive intruders into attacking the systems or networks so that the defenders can learn new techniques.

Several technologies for providing deception have been studied. Software decoys are agents that protect objects from unauthorized access [32]. The goal is to create a belief in the attacker's mind that the defended systems are not worth attacking or that the attack was successful. 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 [33] further investigated the biases of the adversaries 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 inputs of the attacker.

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* [34], a model of deception that challenges the cornerstone of projecting false reality with internal consistency. Sun *et al.* [20, 21] also argued for the value of unpredictability and deception as OS

features. CHAMELEON explored non-intrusive unpredictable perturbations to create an uncertain environment for software being deep analyzed after an initial borderline classification.

1.2.2 Fuzz Testing

Fuzz testing is an effective way to discover coding errors and security loopholes in software, operating systems, and networks by testing applications against invalid, unexpected, or random data inputs. Fuzzers can be divided into two categories: generational fuzzers, that construct inputs according to some provided format specification (e.g. PROTOS [35], SPIKE [36], and PEACH [37]), and mutational fuzzers, that create inputs by randomly mutating analyst-provided or randomly-generated seeds (e.g. AFL [38], honggfuzz [39], and zzuf [40]). 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 [41]. BEAR has 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 [42], for example, randomizes system call parameters to test the validation of file descriptors, and found real bugs [43], including bugs in the Linux kernel. BALLISTA [44] 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 [45] 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 [38], honggfuzz [39], and zzuf [40], 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.

Failure-oblivious computing allows a system or program to continue execution in spite of errors. Rinard *et al.*[46] published a classic paper in the area which introduced a C compiler to

insert checks to dynamically detect invalid memory accesses. On errors, instead of terminating the program or throwing an exception, the program would discard invalid writes and manufacture values to return for invalid reads, enabling the program to continue its execution. ASSURE [47] introduces rescue points in software (discovered via fuzzing) to recover it from unknown faults, while maintaining both system integrity and availability.

BEAR's work on improving resilience is complementary to failure oblivious approaches. By determining the most impactful types of perturbations and the most sensitive system calls, it can help failure oblivious approaches to better target checks and rescue points, greatly improving the effectiveness and the high performance overhead of such approaches.

1.2.3 Malware Detection

Malware detection techniques have been evolving from static, signature-based approaches [48, 49] to dynamic, behavior-based techniques [50–52]. 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 [53, 54] or system-call [55] levels. This dissertation pertains to the areas of behavior-based malware detection, and this section summarizes the state-of-the-art in these areas and highlights topics currently under-studied. Dynamic solutions are able, for instance, to detect sensitive data leaking via system-level taint tracking [56] and keystroke logging via data-flow analysis [57]. 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 decisioning with reasonable results. Kumar et al. used K-means clustering [58] 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 [59]. Mohaisen et al. [60] introduced AMAL to dynamically analyze malware using SVM, linear regression, classification trees, and kNN. Fan et al. used a

sequence mining algorithm to discover malicious sequential patterns and trained an All-Nearest-Neighbor (ANN) classifier based on these discovered patterns for malware detection [61].

Behaviors modelling, however, has become challenging as applications are becoming increasingly diverse [52], 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. [62] 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 [63]. Hou et al. constructed weighted directed graphs on collected system calls and used a deep learning framework to make dimension reduction [64]. 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 [65, 66] 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.

CHAMELEON's uncertain environment is orthogonal and complementary to malware detection approaches, such as PROPEDEUTICA. CHAMELEON can provide the system with a Plan B when malware evades all signatures and all heuristics applied.

1.3 Contributions Of The Dissertation

Based on the previous discussion, this dissertation, leveraging system unpredictability to improve software and system security and reliability, has the following contributions.

1. It thoroughly investigates previous work intersecting the areas of software diversity and deception, fuzz testing, and malware detection.
2. It studies the reliability of applications in facing with different OS unpredictabilities, specifically with the following:
   - discover bugs that would only appear after program deployment and would be hard to reproduce;
   - efficiently target end-to-end checks, and time-consuming testing and verification procedures;
   - design and implement applications that can withstand malicious or buggy OS misbehavior [15].
3. It introduces a system with three different environments to mitigate the "monoculture" problem, and designs and implements the uncertain environment that will bring unpredictabilities to the execution of borderline software (not sure benign or malicious from a malware detector), specifically with the following:
   - make systems diverse by design because of the unpredictable execution in the uncertain environment;
   - increase the work factor of the attackers, as successful malware will need to be resilient to an unpredictable unfavorable environment;
   - decrease the probability of success and speed of attacks.
4. It designs and implements a framework for on-the-fly software classification (malware detection) using knowledge in the cutting-edge machine learning and deep learning techniques, specifically with the following:
   - PROPEDEUTICA, a new framework for efficient and effective on-the-fly malware detection for Windows OS;
   - DEEPMALWARE, a novel deep learning algorithm, learning multi-scale spatial-temporal system call features with multi-stream inputs that can handle software heterogeneity;
   - an evaluation of PROPEDEUTICA with a collection of 9,115 malware and 1,338 benign software.

The remainder of this dissertation is organized as follows. Chapter 2 describes BEAR's architectural design, implementation and evaluation. Chapter 3 describes CHAMELEON's design, implementation and evaluation. Chapter 4 describes PROPEDEUTICA's design ,implementation and evaluation. Chapter 5 concludes this dissertation.

CHAPTER 2
LEVERAGING UNPREDICTABILITY TO IMPROVE SOFTWARE RELIABILITY

2.1 Background

Applications are generally written with the assumption that the OSes on all deployed systems will behave predictably and identically to the testing environment. Recent research [13–15], however, has shown that unpredictability and misbehavior at the OS level are more common than once thought. Unpredictability in OS behavior can cause bugs and security vulnerabilities in applications. These bugs or vulnerabilities can arise from: (i) different ways OSes handle network events and protocols [13], and (ii) subtle and undocumented differences in the behavior of common APIs across different platforms[14] and from OS changes over time. Further, the OS can be buggy or malicious, which breaks the predictability assumption. For example, in Iago attacks [15], a malicious kernel induces a protected process to act against its interests by manipulating system call return values. These application bugs and vulnerabilities are hard to reproduce in a testing environment, and thus are difficult to detect and prevent. Although recent technology trends such as Intel SGX are attempting to introduce mutual distrust between the OS and its applications [16–19], such as by protecting the application's memory from the OS, a significant burden still falls to the application developer to reason about the *semantic impact* of return values from the OS on their applications. In reality, developers are simply not equipped to write robust applications in the face of unpredictable or even adversarial OSes.

Developers need techniques to understand the impact of OS unpredictability and misbehavior on program execution. With a better understanding of this sensitivity, application developers can (i) discover bugs that would only appear after program deployment and would be hard to reproduce, (ii) efficiently target end-to-end checks, and time-consuming testing and verification procedures, and (iii) design and implement applications that can withstand OS malicious or buggy misbehavior [15]. This understanding can also improve OS design, allowing designers to introduce diversity to software without affecting application execution, thereby alleviating the security shortcomings of today's software monoculture. OS developers currently fear any

variability short of bug-for-bug compatibility, lest the OS harm benign programs; with an understanding of which behaviors programs were sensitive or insensitive to, OSes could diversify behavior and implementation.

This chapter introduces BEAR, a Linux-based framework for fine-grained statistical analysis of application sensitivity to OS unpredictability. BEAR statistically analyzes a program using a set of unpredictability strategies on a set of commonly used systems calls, to discover the most sensitive system calls for each application, the most impactful strategies, and how they predict abnormal program outcome (crash, segmentation fault etc). Rigorously understanding on application sensitivity to OS misbehavior can help developers discover challenging bugs and edge cases, as well as identify the scenarios that most need end-to-end checks, targeted testing and verification procedures. BEAR performs a correlation and regression analysis on the program outcome after the program is run for a number of times and for every selected combination of system call, perturbation strategy, and perturbation threshold. The correlation analysis informs whether or not, for a particular program, there is a relationship between a system call, strategy, threshold, and a program execution outcome. If there is a correlation, regression analysis predicts the likelihood of an abnormal program outcome for different system calls, strategies and thresholds. This analysis allows a developer to have insights on the following questions: *(i) which system calls are the most sensitive to OS unpredictability and by what degree? (ii) which strategies cause the most impact in program execution and by what degree? and (iii) do program type and execution workloads affect the strategy impact or system call sensitivity?*

2.2 Design

With unpredictability, we designed BEAR to allow a developer to analyze a target program against various degrees of OS unpredictability. BEAR is comprised of several modules residing both in user-space and in the OS kernel. In summary, a testing harness component (the *Manager*) loads a module (*Perturbation Module*) in the kernel that implements a set of perturbation strategies on a set of system calls. The *Manager* also runs the target program and invokes a statistical module for analysis.

Figure 3-2 illustrates BEAR's general architecture. More specifically, BEAR runs the application inside of a testing harness, called the *Manager*. The manager is responsible for receiving user input and invoking all other components. The user (typically an application developer) inputs the target program and a test case, the set system calls to be perturbed, the set of perturbation strategies to be used, a perturbation threshold, and a timeout for hung programs.

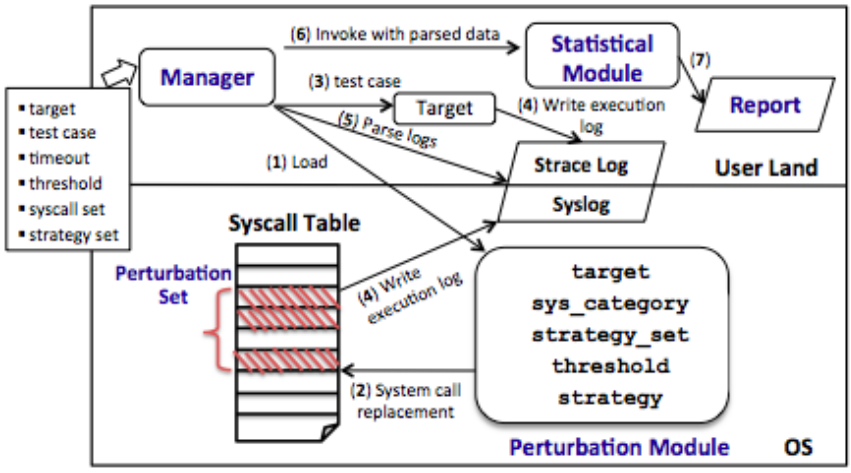

Figure 2-1. BEAR's architecture.

The user can select to perturb all system calls or can specify a system call category for perturbation, for example, file management, signal control, process scheduling, and network communication. The user can also specify that all perturbation strategies should be used during the analysis, or select a particular strategy. For example, consider a developer running BEAR and passing Chrome as the target program, "All" as the set of system calls, and 10% as the perturbation threshold. BEAR will perform a series of program runs with the test case, where in each run it applies one of the strategies to a system call in the selected set with 10% probability. In the Chrome example, during the analysis of `sys_read` approximately 10% of its invocations will be perturbed with a certain strategy. Each run represents a combination $\{sys\_read, strategy_i\}$, where i represents each strategy that can be applied on $\{sys\_read\}$. BEAR runs each combination $\{sys\_read, strategy\}$ a number of times for statistical significance.

The next component in BEAR's architecture is the *Perturbation Module* residing at the OS level. This module is loaded by the Manager (*Step 1* in Figure 3-2), and is responsible for implementing all perturbation strategies. Table 2-1 lists part of the system calls that can be perturbed (the *Perturbation Set*), their categories and the perturbation strategies that can be applied to them (strategies are discussed further in this section). During initialization the *Perturbation Module* replaces the original versions of the system calls in the perturbation set with new versions implementing the perturbation strategies (*Step 2*).

The target program is then run by the *Manager* with `strace`, and during the program runs (*Step 3*), the modified versions of the perturbed system calls and `strace` will record relevant information about the run, respectively in the system logger and in the `strace` logger (*Step 4*). After the program runs finish, the *Manager* captures these logs (*Step 5*) and formats them for input to the *Statistical Module* (*Step 6*).

Table 2-1. Applicable strategies and part of system calls in the perturbation set.

Category	System call example	Strategies	Common related bugs
Memory management	sys_unmap	Fail to deallocate system call (failMem)	Memory leak
Memory management	sys_mmap	Empty buffer in memory system call (nullMem)	Null dereferencing
Signal control	sys_mlock	Failure to lock related system call (failLock)	Synchronization error
Signal control	sys_kill	Failure to signal control system call (failSig)	Signal delivery error
File operation	sys_read	Different data type to buffer parameter (diffType)	Value outside domain
File operation	sys_write	Injection of bytes to system call with a buffer(bufOf)	Buffer overflow
File operation	sys_access	Failure to access system call (failAcc)	Dealing with volatile objects
Network communication	sys_sendto	Reduction of buffer size/length parameter (redLen)	Buffer return not checked
Process scheduling	sys_mq_notify	Fail to notify system call (failNoti)	Naked notify in method
Process scheduling	sys_setuid	Fail to change user id (chUid)	Privilege degradation

Note: The full set of system calls can be found in [67]. Later analysis uses `syscall` rather than `sys_syscall` for short.

The *Statistical Module* then performs a correlation analysis to discover (i) whether there is a relationship or a correlation between a system call, strategy and the program outcome (normal or abnormal), and (ii) the strength of this relationship. Next, the *Statistical Module* performs a regression analysis to predict the likelihood of an abnormal program outcome (an erroneous exit) for each system call and strategy analyzed. Finally, it writes the results of the analysis in a report for the developer (Step 7).

2.2.1 Perturbation Strategies

The *Perturbation Module* applies unpredictability to the *Perturbation Set*. In this work we chose common OS system calls from various categories: memory management, signal control, file operation, network communication and process scheduling.

The perturbation set (Table 2-1) targets major system misbehavior that will cause common software bugs and security flaws [68], such as memory leak, synchronization error, values outside domain, and buffer overflow, etc. In this work, we introduced a set of perturbation strategies to cover the following scenarios of OS misbehavior:

- **Fail to deallocate a system call (failMem)**: this misbehavior strategy simulates a memory leak when a program's allocated memory is not freed subsequently, which can cause the program to substantially increase its memory usage and crash. We implement this strategy by returning -1 when `sys_unmap` is invoked.

- **Empty buffer in memory system call (nullMem)**: This strategy simulates a missing initialization which can lead to a NULL reference error. This strategy is implemented by changing the buffer parameter to `null` for memory related system calls, such as `sys_mmap`.

- **Fail of lock related system call (failLock)**: This strategy simulates synchronization errors (deadlocks, race conditions and live lock) in the control of the execution of multi-threads programs with shared data. We implement this strategy by returning -1 when lock related system calls are invoked, such as `sys_mlock`.

- **Failure to signal control system call (failSig)**: This strategy simulates signal delivery error and returns -1 when a signal control system call is invoked, such as `sys_rt_sigaction`.

- **Different data type to parameter (diffType)**: This strategy simulates an incorrect input passed, such as passing an integer parameter, where the expected type is a character. We implement this strategy by changing `char *` parameter to `int` when a system call such as `sys_write` is invoked.

- **Injection of bytes to system call with buffer (bufOf)**: This strategy simulates a buffer overflow error and is implemented by injecting random bytes to a buffer parameter in a network or file related system call such as `sys_write` or `sys_sendto`.

- **Failure to access system call (failAcc)**: This strategy simulates the failure of checking access for global volatile objects that are going to be shared between several threads, such as a library. We implement this strategy by returning -1 for the `sys_access` system call.

- **Reduction of buffer size/length parameter (redLen)**: This strategy simulates the scenario of an incorrect return value check that can cause byte loss, if only part of a buffer passed as a system call parameter is read or written. We implement the strategy by reducing the buffer length for network and file related system calls.

- **Fail to notify system call (failNoti)**: This strategy simulates a failed call of `sys_mq_notify`, which can cause a shared object states to be modified without knowledge. The strategy is implemented by returning -1 to `sys_mq_notify`.

- **Fail to change user id (chUid)**: This strategy simulates a failure to change privileges, such as failing to change the user id to root when `sys_setuid` is invoked.

2.2.2 Implementation Details

BEAR's *Manager* executes at the user-level and automates analysis by (i) loading the *Perturbation module* with the input parameters passed by the developer; (ii) executing the target program for each combination of system call, strategy, and threshold enough times for statistically significant results; (iii) capturing the results of each run from the `syslog` and `strace` logs; (iv) parsing the results of each run into a format required by the open-source statistical module R [69]; and (v) invoking R for analysis. The user inputs into *Manager* the target program pathname, the category of system calls to be perturbed (all, file, process, signal, or network), a perturbation threshold, a test case, and a timeout, in case a process hangs. Figure 2-3 illustrates the format of BEAR's input and output.

Upon initialization, the *Perturbation module* replaces the system call table pointers for the system calls in the perturbation set with new versions implementing the strategies. [1] In newer versions of Linux, the system call table is read-only; we remap the table read-write. The Perturbation module introduces four new variables: (i) `target`, the pathname for the target program being analyzed; (ii) `sys_category`, the category of system calls being analyzed (all, file, net, process, signal, or mem); (iii) `strategy_set`, the strategy(ies) to be used in the

[1] We decided not to implement the strategies with `strace` because it has limitations on changing the parameters.

analysis (see Section 2.2 for details); and (iv) `threshold`, a number in $[0, 1]$, which represents the probability of perturbation for each system call that is part of the analysis.

Algorithm 1 shows how the OS applies perturbation strategies to `sys_write`. The system call performs three checks that all must be true before a strategy is applied: (i) the current program must be `target`, (ii) the current system call to be analyzed is `sys_write`, and (iii) a randomly selected number in $[0, 1]$ should be smaller than the threshold. If all conditions are true the selected strategy is applied on the system call.

Algorithm 1: Perturbing `sys_write()`

```
Function long my_sys_write(fd, buf, size)
    if (current == target and syscall == sys_write and (random(0.0, 1.0) < threshold))
     then
        switch strategy do
            case diffType do
                /* Pass different data type                    */
                intNum = getRandomInt();
                return orig_sys_write(fd, intNum, size);
            end
            case redLen do
                /* Reduce buffer length                        */
                newsize = random(0, size);
                orig_sys_write(fd, buf, newsize);
                return size;
            end
            case bufOf do
                /* Inject buffer bytes                         */
                newsize = random(0, 4);
                append newsize bytes in buf ;
                orig_sys_write(fd, buf, newsize);
                return size;
            end
        end
    else
        return orig_sys_write(fd, buf, size);
    end
```

The *Manager* checks every `check_frequency` (also a configurable parameter) for whether the target program is still running by searching the program in the list of running

programs. If the program exits normally, it will invoke system call `exit_group` with exit code 0. Otherwise, if the exit code is non-zero, the program ended abnormally. The exit code is recorded in the `strace` log.

2.2.3 Statistical Methods

This section explains the statistical methods underlying BEAR. In our study, the independent variables are system call, perturbation strategy, program type (CPU-bound or I/O bound), and perturbation threshold. The dependent variable is the program outcome, which can be normal or abnormal. The user of BEAR manipulates one or more of these independent variables and measures the change in the dependent variable.

Our statistical analysis used standard hypothesis testing [70]. In our context, the null hypothesis (H_0) is that the execution of a particular system call or a perturbation strategy is unassociated with the program outcome, *i.e.,* that misbehavior of the system call will not be associated with nor will it predict abnormal program outcome.

A statistical test produces a *p*-value based on the results under analysis. The *p*-value is the probability of the test making a **Type I error** [70]: *rejecting the null-hypothesis when it is actually true*. When the *p*-value is very low (0.05 or 0.01), the experimenter will only observe this error 5% or 1% of the time.

Before applying the appropriate statistical test [2] , the experimenter selects an appropriate *significance level* called α, which is also a very low probability (0.10, 0.05 or 0.01). α is the experimenter's threshold for statistical significance: the test is statistically significant if the produced *p*-value $\leq \alpha$.

Figure 2-2 summarizes the statistical analyses used in BEAR's evaluation. We performed a *correlation analysis*, to discover and measure the strength of the association between the

[2] the type of statistical test depends on the goals of the experimenter, the type of the study design – single group, between groups, repeated measures, *etc.*, the type of data – continuous, discrete, categorical, ordinal, *etc.*

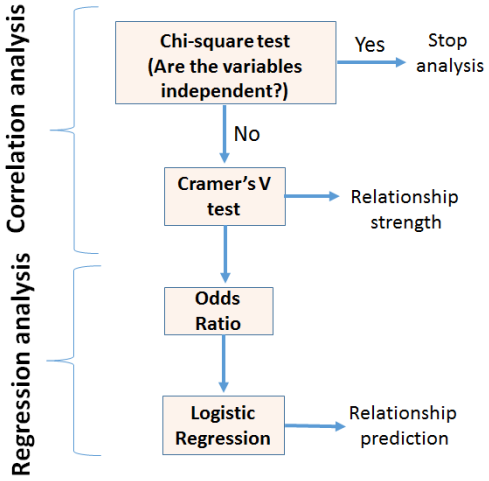

Figure 2-2. Statistical tests used in this study.

variables, and a *regression analysis* to discover the cause and effect of the variables' relationship. The next subsections detail each one of these tests.

2.3 Evaluation

This section describes the experiments we conducted to validate BEAR. We analyzed 113 applications including applications from GNU Core Utilities [71], SPEC CPU2006 [72] and Phoronix Test Suites [73]. We selected Core Utilities because their applications provide the most commonly used functions in Linux. SPEC and Phoronix benchmarks allowed our evaluation to test different workloads and provided sufficient off-the-shelf test cases. BEAR framework doesn't rely on source code for white-box testing, so test coverage doesn't make a difference to our result. For the CoreUtils applications, we selected test case as their commonly used functions. The studied applications fall into four categories—processor (70 applications), network (11 applications), disk (27 applications) and encoding (5 applications).

Our evaluation lasted for 200 hours with approximately 100,000 runs in total. Each run is a combination of the following tuple: $\{program, syscall, strategy, threshold\}$. The evaluation covered every system call and strategy in Table 2-1 and every threshold in $\{10\%, 50\%, 90\%\}$. Each combination ran five times and the number of runs per combination (five) was determined

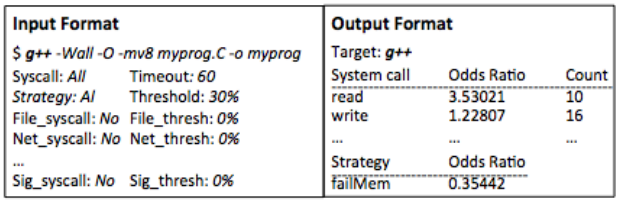

Figure 2-3. Format of BEAR input and output.

empirically based on the requirements of the chi-squared tests [74]: the chi-square matrix's expected counts should be at least five counts in each cell.

We installed the BEAR framework on a virtual machine with 1GB RAM, 40GB Hard Disk, x86_64 architecture, and single processor running Ubuntu 12.04 with kernel release 3.11. The host machine has 16GB RAM, 160GB Hard Disk, x86_64 architecture, and 8 processors running Ubuntu 14.04 with kernel release 3.13.

Figure 2-3 shows an example of the input format a developer needs to provide. By default, BEAR tests all system calls with all strategies, with 1.5 times of the estimated time to complete as timeout in case any test hangs in the middle of a run. In the example given in Figure 2-3, `g++` is tested by generating optimized code on a Solaris machine with warnings. After the analysis, BEAR outputs system calls and strategies ranked by impact. The number of times each system call is invoked (*Count*) is also provided.

As discussed in Section 2.2.3, for each collection of program runs we performed a correlation and a regression analysis. The next subsections describe each one of these analyses and how they were evaluated. In our evaluation, we considered the significance level $\alpha = 0.05$ (see section 2.2.3). The evaluation is organized around the following questions:

1. Which perturbation strategies are the most and least impactful on abnormal program execution? Does it vary based on threshold?
2. Which system calls are the most and least sensitive to perturbation? Does it vary based on perturbation threshold?
3. How does the strategy impact and system call sensitivity vary based on the program type (processor, disk, network, and encoding)?

4. Will the result differ when applications are executed with different workloads?

2.3.1 Correlation Analysis

As illustrated in Figure 2-2, the first analysis phase in BEAR is a correlation analysis with a chi-squared test to discover whether the program outcome is independent of applying perturbation strategies on system calls. Table 2-2 and 2-3 illustrate the chi-square test results between abnormal execution outcome and system calls/strategies. df represents the degrees of freedom and, in our example, equals the number of strategies/system calls minus one. We considered all applications in categories processor and encoding as CPU-bound, and all applications in categories network or disk as I/O-bound.

As discussed in section 2.2.3, the evidence to reject the null hypothesis is a chi-square statistic value higher than that in the chi-square distribution table [75]. For system calls ($df = 20$), the chi-squared distribution table shows a value of 39.997, which is lower than every X-squared value in Table 2-2. For strategies ($df = 9$), the chi-squared distribution table shows a value of 18.548, which is also lower than every X-squared value in Table 2-3. This leads us to reject the null hypothesis of independence and conclude that there is an association between a program execution outcome and a perturbation system call, and likewise on a perturbation strategy.

Table 2-2. Chi-square result for *system call* and program execution outcome.

	X-squared value	df	p-value
All programs	262.39	20	<0.0001
CPU-bound	48.732	20	0.0003
IO-bound	486.21	20	<0.0001

Table 2-3. Chi-square result for *strategy* and program execution outcome.

	X-squared value	df	p-value
All programs	66.428	9	<0.0001
CPU-bound	42.667	9	<0.0001
IO-bound	112.02	9	0.0017

To understand the strength of the association between these variables, we did follow-up tests using the Cramer's V method. The tests produced association levels from 15% to 35%, which are

acceptable and even desirable based on Cramer's V criteria [76] (see Section 2.2.3). Given these results we could proceed to the next phase in our evaluation: logistic regression, which allows us to better understand the influence of system call and strategy on the program outcome.

2.3.2 Regression Analysis

This section describes the logistic regression model generated by BEAR.

2.3.2.1 Strategy impact on program outcome

BEAR uses two types of program outcomes: normal and abnormal. We initially classified the possible program outcomes into four levels: normal, crash, abort, and segfaults. During the analysis of 113 programs, we observed that the number of aborts and segfaults was 1-2 orders of magnitude smaller than the number of normal executions and crashes, which would decrease the accuracy of the statistical model. Coupled with controversial opinions on how to classify normal and crashes [44], this prompted us to consider crashes, aborts, and segfault as a single category of program outcome: abnormal. In other words, result **normal** referred to a correct execution or a graceful exit of a program and result **abnormal** referred to all the other results.

Figure 2-4 shows the impact of perturbation strategies as predictors on program outcome for all programs and for the thresholds of 10%, 50% and 90%. With $p<0.05$, every strategy except `failNoti` and `chUid` are statistically significant in predicting an abnormal program outcome. Odds ratios on the x-axis show the odds of a strategy causing an abnormal program outcome when compared to a reference strategy. The reference strategy doesn't make a difference on the analysis result. We selected *nullMem* as the reference strategy and plotted how much more or less likely (odds ratio) the other strategies are in causing an abnormal outcome compared to *nullMem*, the most significant strategy in predicting an abnormal program outcome.

For example, for the perturbation threshold of 90%, strategy *failAcc* has an odds ratio of 0.5 ($p<0.001$) when compared with *nullMem*, which means that it is 50% less likely to cause an abnormal program outcome than *nullMem* for the same threshold. Notice that the odds ratios vary depending on the perturbation threshold. For example, *diffType* is more likely to cause an abnormal program outcome for 10% and 50% perturbation threshold than for 90%.

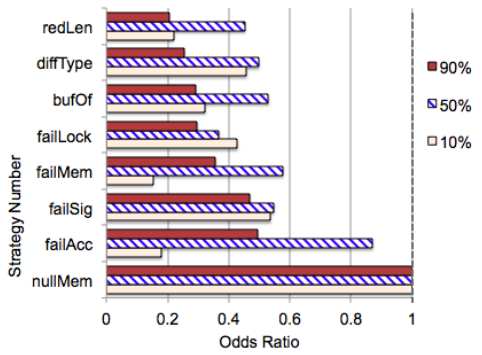

Figure 2-4. The impact of perturbation strategies in predicting abnormal program outcome, tested on all programs with thresholds of 10%, 50% and 90%. Reference strategy is *nullMem*. Odds ratio shows how much more or less likely a strategy is to cause an abnormal program outcome compared to the reference strategy. Absence of `chUid` and `failNoti` means that they are not significant in predicting an abnormal execution.

On threshold 50% and 90%, the impact of strategies follows a similar trend, with `nullMem` `failAcc` and `failSig` as the three most impactful strategies and `failLock` `diffType` and `redLen` as the three least impactful strategies. The impact of strategies on threshold 10% differed by some degree. When the threshold increases from 50% to 90%, the overall impact of the strategies decreases compared to reference strategy `nullMem`, which shows that the impact of `nullMem` increases faster than that of the rest of strategies. Also, the impact of `failSig` and `failLock` increases faster than that of buffer and memory-related strategies such as `failAcc`, `failMem`, `bufOf`, `diffType` and `redLen`.

The absence of strategies `chUid` and `failNoti` in Figure 2-4 does not necessarily mean that they have no impact on an abnormal program execution. They are just not statistically significant in predicting an abnormal result under the defined level $\alpha = 0.05$. In other words, their impact is relatively little compared to the other strategies.

We evaluated the tested programs execution based on whether the application was CPU or I/O-bound. We considered all applications in categories processor and encoding as CPU-bound, and all applications in categories network or disk as I/O-bound.

Figures 2-5 and 2-6 show the impact of the strategies on program outcome when we analyze I/O-bound and CPU-bound programs separately. Figure 2-5 shows that only five strategies related to buffer and memory demonstrate statistical significance in predicting an abnormal program outcome for I/O-bound programs. `nullMem` still has the strongest impact while `bufOf`, `redLen`, and `diffType` show similar impacts at around 20% of `nullMem`. For CPU-bound programs, the result shows some differences as displayed in Figure 2-6. Besides the five statistically significant strategies in I/O-bound programs, signal-related strategies such as `failLock` and `failSig` are also significant in CPU-bound programs. `nullMem` still has the strongest impact, and the rest of the strategies are at about 50% to 70% of its impact. `failAcc` does not appear in Figure 2-5 and 2-6 because of the *complete separation* statistical phenomenon that occurred after we separated the data from CPU and IO-bound programs. *Complete separation* occurs when a linear combination of the predictors yields a perfect prediction of the response, which skews the model and reduces its overall predictive strength. In our experiment, any system call/strategy presenting a number of normal outcomes 20 times more than the number of abnormal outcomes is considered affected by complete separation.

The lesson learned is that `nullMem` is the highest impactful misbehavior for predicting an abnormal execution in both I/O-bound (disk and network) applications and CPU-bound (processor and encoding) applications. Buffer overflow, value outside domain, unchecked return length, and memory leak should be carefully considered during software development. CPU-bound software is especially sensitive to signal-related errors. In case of limited resource (time, performance hit limit, man hours) for testing/verification, naked notify and privilege degradation can be given the lowest priority as they bring the least impact on program execution.

2.3.2.2 System call impact on program execution

At threshold 10%, every system call makes equal impact in predicting an abnormal program outcome. Thus in this section, we only consider system call sensitivity with the thresholds of 50% and 90%. At these threshold levels, every system call is statistically significant in predicting an abnormal program outcome. Figure 2-7 shows the nine most impactful system calls compared

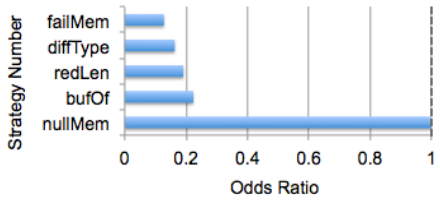

Figure 2-5. Impact of perturbation strategies in predicting abnormal program outcome, tested on I/O-bound programs. Each strategy is compared with reference strategy nullMem.

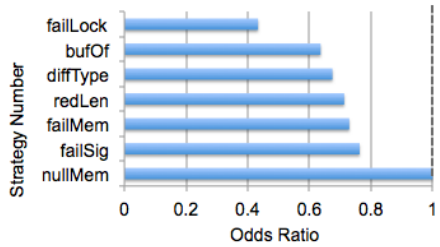

Figure 2-6. The impact of perturbation strategies in predicting abnormal program execution, tested on CPU-bound programs. Each strategy is compared with reference strategy nullMem.

to the reference system call `mmap` on abnormal program outcome. The likelihood of an abnormal program outcome is again demonstrated by using odds ratios on the x-axis. `mmap` is chosen as the reference system call since it is the most statistically significant in predicting an abnormal program outcome. This finding corroborates our previous finding that `failMem` and `nullMem` are the two most impactful strategies in causing abnormal program execution.

At threshold 50%, system call `access` is approximately 60% less sensitive than `mmap` and is the second-most sensitive for abnormal execution. A failure in `access` prevents a program from accessing a library and can abruptly cause its termination. The other system calls demonstrate a relatively equal significance. Network-related system calls such as `sendto` and `recvfrom` are not in the top nine most significant list, which can be attributed to their robustness against perturbation or the presence of robust end-to-end checks and retry mechanisms over network layer. At threshold 90%, every system call presents only 10% to 15% of `mmap`'s sensitivity. This happens because the sensitivity of `mmap` increases faster than the other system calls when the threshold increases.

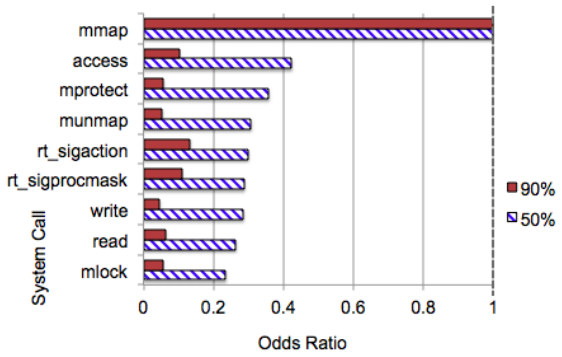

Figure 2-7. Top nine most impactful system calls on predicting abnormal program outcomes for all programs, compared with reference system call `mmap`.

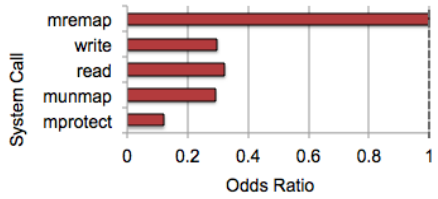

Figure 2-8. Statistically significant system calls on predicting abnormal program outcomes for IO-bound programs, compared with reference system call `mremap`.

Figures 2-8 and Figure 2-9 evaluate the I/O-bound and CPU-bound subsets respectively. In I/O-bound programs, only five system calls are statistically significant in predicting an abnormal outcome. `mmap` was removed from the analysis model for I/O-bound programs due to the *complete separation* problem. Any perturbation on `mmap` for all thresholds almost always caused an abnormal program outcome, and adding a variable where the outcome is already known skews the model and reduces its overall predictive strength.

Thus after eliminating `mmap` from the model, `mremap` becomes the most sensitive system call and works as the reference system call in Figure 2-8. Again, no network-specialized or signal-related system calls show up in the figure, corroborating the result in Figure 2-7 that no network-specialized system calls are highly significant and the result in Figure 2-5 that no signal-related strategies are highly significant.

Figure 2-9 listed all the system calls that are statistically significant for an abnormal CPU-bound program outcome compared with reference system call `mmap`. Generic system calls (such as `write` and `read`), memory-related system calls (such as `munmap` and `mprotect`) and

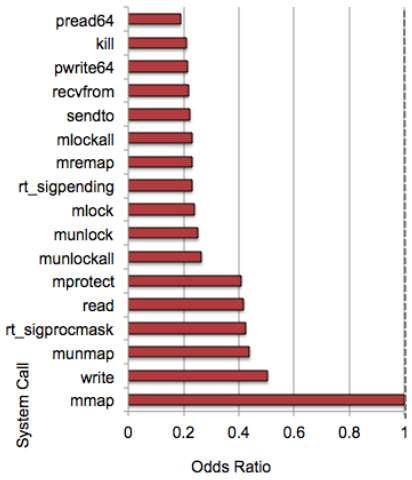

Figure 2-9. Statistically significant system calls on predicting abnormal program outcomes for CPU-bound programs, compared with reference system call `mmap`.

signal-related system calls (such as `rt_sigprocmask`) have more than 40% of the sensitivity of `mmap`. Network-related system calls, such as `sendto` and `recvfrom`, show lower sensitivity at around 20% of `mmap`. System calls such as `readv/write` and `preadv/pwritev` with a vector buffer do not appear in Figure 2-9, while `read/write` and `pread64/pwrite64` with a general buffer do, indicating that system calls with a vector buffer are less impactful in predicting an abnormal program execution. Notice that even though some system calls present equal sensitivity, such as `sendto` and `mlockall`, the frequency at which they are called is totally different. Developer and testers should balance their time, not only based on how severe the impact on the system call might be, but also on how frequently the system call is invoked.

2.3.2.3 Workload impact on program outcome

SPEC CPU2006 and Phoronix Test Suite provide applications that can be configured to run under different workloads. Twenty-six applications are selected because they can be installed successfully on our experiment virtual machine. The workloads contain three levels: light, medium, and heavy, which correspond to test, train, and ref levels for SPEC CPU2006, and first, middle-most, and last level in Phoronix Test Suite.

Figure 2-10 shows the statistically significant strategies in predicting abnormal program outcome on three workloads compared with reference strategy `nullMem`. From Figure 2-10, we

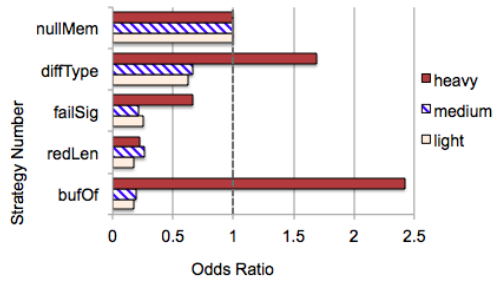

Figure 2-10. Impactful strategies on predicting abnormal program outcomes on three workloads, compared with reference strategy `nullMem`.

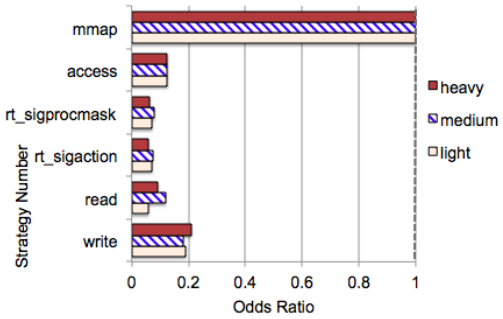

Figure 2-11. Statistically sensitive system calls on predicting abnormal program outcomes on different workloads, compared with reference system call `mmap`.

can see that the results for light and medium workloads follow the same trend, with `nullMem` and `diffType` highly impactful and `failSig`, `redLen` and `bufOf` less impactful. However, when workload increases from medium to heavy, `bufOf` rapidly grows the impact to 10 times of the original, and `diffType` and `failSig` double their impacts. Figure 2-11 demonstrates the sensitive system calls on predicting abnormal outcome for the three different workloads. Every system call in the figure has 10% to 20% sensitivity of `mmap`'s sensitivity and presents equal impact on three workloads, indicating that the workload does not influence the system call sensitivity on predicting abnormal program outcome.

This tells the developers that regardless of the system call type, buffer overflows and value outside domain bugs should be taken seriously when heavy workloads are used.

2.3.3 Summary And Recommendations For Developers

Our results showed that for CPU-bound applications, the four most sensitive system calls are `mmap`, `write`, `munmap` and `rt_sigprocmask`. While for I/O-bound applications, the

four most sensitive system calls are `mremap`, `write`, `read` and `munmap`. For both I/O-bound and CPU-bound applications, null dereferencing and buffer overflow strategies are the two most severe in predicting an abnormal program outcome. For CPU-specific applications, signal-related errors are of the equal importance. The two least impactful strategies are naked notify in method and privilege degradation, and the rest of the strategies are of similar impact.

Surprisingly, network-specialized system calls (`read` and `write` excluded) did not show high impact on abnormal execution, taking into account their wide use in programs and APIs, and this should be attributed to the effectiveness of a perfect end-to-end checking and retry mechanism over network layer.

When workload is heavy, the impact of buffer overflow on abnormal execution doubled five times than medium workload and far exceeded the impact of null dereferencing. The same thing happens on wrong-type buffer strategy, which doubles the impact on heavy workload. On light and medium workload, strategy impact and system call sensitivity remains invariant for all applications.

When resources are limited (time, man hours, performance), developers should focus on buffer type and return length checking, considering how commonly used `read` and `write` system calls are, especially when application workload is heavy. One strategy developers should take is to use system calls that are specialized and that use vector arrays in a struct parameter, rather than a standard array in the parameter list. Moreover, given a choice of more than one system call for a particular functionality developers should choose those with a larger parameter list. Null dereferencing is a severe problem as well and almost the hardest to debug when a segmentation fault occurs. Therefore, failure-oblivious computing can be a promising way for saving developers and testers from memory bugs.

CHAPTER 3
LEVERAGING UNPREDICTABILITY TO IMPROVE SYSTEM SECURITY

3.1 Background

After investigating which types of OS uncertainty are most impactful to software resilience and which system calls are more sensitive to perturbations, this chapter aims to understand whether uncertainty impacts malware and benign software differently, and to what extent it affects them.

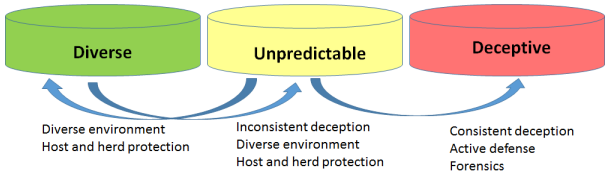

Figure 3-1. CHAMELEON can transition processes among three operating modes: Diverse, to protect benign software; Unpredictable, to disturb unknown software; and Deceptive, to analyze likely malware.

Previous work has proposed CHAMELEON [21] that combines inconsistent and consistent deception with software diversity for active defense of computer systems and herd protection. It provides three distinct environments for process execution (Figure 3-1): (i) a diverse environment for whitelisted processes, (ii) an unpredictable/uncertain environment for unknown or suspicious processes (inconsistent deception), and (iii) a consistently deceptive environment for malicious processes. Known benign or whitelisted processes run in the **diverse** operating system environment, where the implementation of the program APIs are randomized to reduce instances with the same combinations of vulnerable code. In some sense, the diverse environment combines ASLR and other known randomization techniques with N-version programming [10], except that CHAMELEON doesn't run the versions in parallel, but rather diversifies across processes. The insight is that a modular library OS design makes the effort of manual diversification more tractable. Rather than require multiple complete OS implementations, the CHAMELEON design modularizes the Graphene library OS [77] and components are reimplemented at finer granularity and possibly in higher-productivity languages. The power of this design is that mixing and matching pieces of N implementations multiplies the diversity by the granularity of the pieces.

Unknown processes run in the unpredictable /uncertain environment, where a subset of the system calls have their parameters modified or are silenced probabilistically. Unpredictability is primarily implemented at the system call table, or library OS platform abstraction layer. The execution of processes in this environment is unpredictable as they can lose some I/O data and functionality. A malicious process in the unpredictable environment will have difficulty accomplishing its tasks, as some system call options used to exploit OS vulnerabilities might not be available, some sensitive data being collected from and transferred to the system might get lost, and network connectivity with remote malicious hosts is not guaranteed.

Unpredictability raises the bar for large-scale attacks. An attacker might notice the hostile environment, but its unpredictable nature will leave her with few options, one of them being system exit, which from the host perspective is a winning outcome.

Processes identified as malicious run in a **deceptive** environment, where a subset of the system calls are modified to deceive an adversary with a consistent, but false appearance while forensic data is collected and forwarded to response teams such as CERT [78]. Shadow Honeypots [79] have been used similarly for testing the effects of anomalous network traffic and protecting against potentially unknown attacks. In this environment, files the attacker intends to leak will be honeyfiles, and any system privileges she thinks she has will be limited to a sandbox. Connections and activities with malicious remote hosts will be intercepted and logged.

CHAMELEON can adjust its behavior over the lifetime of a process. Its design includes a dynamic, machine learning-based process categorization module that observes behavior of unknown processes, and compares to training sets of known good and malicious code. Based on its behavior, a process can migrate to the diverse or deceptive environment. This chapter mainly investigates the effects of the uncertain environment.

3.2 Threat Model And Assumptions

CHAMELEON's protection is designed for corporate environments, which have adopted the practice of controlling software running at their perimeters [80]. 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.

3.3 Design And Implementation

We designed and implemented CHAMELEON for the Linux OS, mainly investigating the effectiveness of the **uncertain** environment, compared with a standard environment. The standard environment works predictably as any OS would, and the uncertain environment has a subset of the OS system calls undergoing unpredictable interferences.

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

3.3.1 The interference set

Our first step was deciding what system calls were good candidates for interference. We relied on Tsai et al.'s study [81], which ranked Linux system calls by their likelihood of use by applications. Based on these insights, we selected 37 system calls for the interference set to represent various OS functionalities relevant for malware (file, network, and process-related). Most of these system calls (summarized in Table 3-1) are I/O-bound, since I/O is essential to most malware, regardless of its sophistication level.

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

Table 3-1. System call interference 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

3.3.2 Interference Strategies

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

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

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

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

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

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

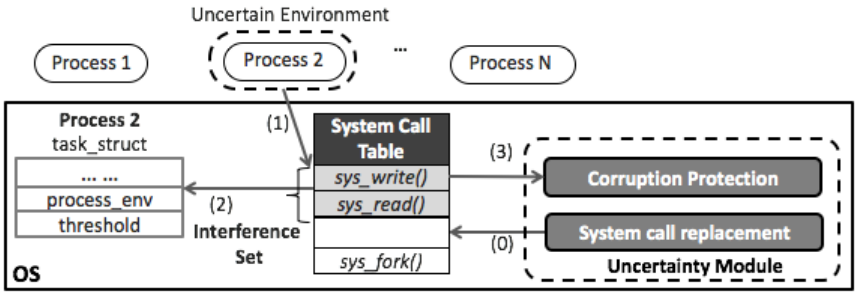

Figure 3-2. System architecture. When a process running in the uncertain environment invokes a system call in the interference set (1), the Uncertainty Module checks if the process is running in the uncertain environment (2), and depending on the execution of the corruption protection mechanism (3),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

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

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

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

3.3.3 System Architecture

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

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

- fd_list: Keeps a list of critical file descriptors during runtime execution. Interference on system files, such as library or devices, will likely crash the program execution. Thus, interference is not applied to system calls manipulating those file descriptors (see Section 3.3.4 for more details).

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

Figure 3-2 illustrates the architecture and operation of the uncertain environment. A key component of the architecture is a loadable kernel module, the *Uncertainty Module*, which monitors the execution of all system calls in the interference set, and applies a randomly-chosen interference strategy to the system call, depending on the process environment and the interference threshold.

For example, consider Process 2 in Figure 3-2, loaded in the uncertain environment and invoking `sys_write` (Step 1). Because `sys_write` is in the interference set, it can introduce uncertainty in its own execution. First the system call inspects Process 2's environment and finds that it runs in the uncertain environment (Step 2). Next, `sys_write` runs the corruption protection mechanism (Section 3.3.4) to make sure that no interference will occur if the system call is accessing a critical file (Step 3). If `sys_write` is not accessing a critical file, CHAMELEON decides based on the threshold whether or not a strategy should be applied. If a strategy is to be applied, `sys_write` randomly selects one of the strategies that can be applied to its execution.

3.3.4 Corruption Protection Mechanism

The uncertainty module employs a corruption protection mechanism to prevent interference while a process in the uncertain environment is accessing critical system files, which might cause early termination of the process. The files are identified through file descriptors, created by `sys_open`, `sys_openat` and `sys_creat`, and deleted by `sys_close`. System calls whose parameters are file descriptors, such as `sys_lseek`, `sys_read` and `sys_write`, are under this protection mechanism. These protected files are determined by an administrator and tracked by setting an extended attribute in the file's inode in the `.security` namespace; a similar strategy is employed by SELinux [82].

When a process running in the uncertain environment opens a file with a pathname beginning with critical directories or containing keywords, the file's descriptor (`fd`) is added to a new per process data structure fd_list. Later, when this process invokes `sys_read` or `sys_write`

referring to an fd in fd_list, the protection mechanism will prevent interference strategies from being applied to these system calls.

3.3.5 Evaluation

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

Our evaluation leveraged a collection of 113 software including common software from GNU projects [83], SPEC CPU2006 [72] and Phoronix-test-suite [73], and 100 Linux malware from THC [84] and VirusShare [85]. The 100 malware samples were randomly selected from different categories (22 flooders, 14 worms, 15 spyware, 24 Trojans and 25 viruses). In total, our evaluation set contained 147 I/O-bound and 66 CPU-bound software samples, containing both common benign software and malware.

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

3.3.6 General Software

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

As shown in Table 3-2, at threshold 10% with intrusive strategies, on average 37% of the tasks experienced some form of crash or hamper. With non-intrusive strategies, this percentage was 30%. For a 50% threshold and intrusive strategies, 59% of the software was adversely affected. With non-intrusive strategies, this number was 10% smaller. Programs that wrote into files were the most sensitive to uncertainty as all of the text editors (for 50% threshold) and half of them (for 10% threshold) were adversely affected by uncertainty.

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

We analyzed the performance penalty caused by the interference strategies, such as process delay and process priority decrease on all 23 benchmark software whose execution could be scripted. Highly interactive software were tested manually and showed negligible overhead. Figure 3-3 shows the average runtime overhead for software whose execution could be scripted running in the uncertain environment. For runtimes ranging from 0 to 0.01 seconds, the average penalty is 8%; for runtimes ranging from 0.1 to 1 seconds, the average penalty is 4%; for runtimes longer than 10 seconds, the average penalty is 1.8%. This shows that the longer the runtime, the smaller the overhead is. One hypothesis is that software with longer execution time are usually CPU-bound programs performing time-consuming calculations. Because most of the system calls in the interference set are I/O related, CPU-bound programs are perturbed less and thus smaller overhead are incurred.

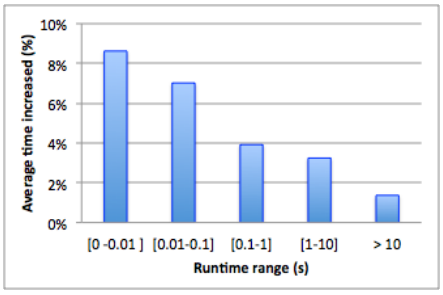

Figure 3-3. Performance penalty for 23 benchmark software whose execution time could be scripted. We categorized the software according to their average runtime.

During our analysis we also measured the code coverage by compiling the analyzed software source code with gcov [86], EMMA [87] and Coverage.py [88] based on the software's programming language. On the 71 applications for which we had source code, the average coverage was 69.49%.

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

3.3.7 Malware

We also analyzed how malware were affected by the uncertain environment, for intrusive and non-intrusive strategies. We considered that malware were adversely affected by the uncertain environment if they crashed or executed in a hampered fashion. An execution is considered *Crashed* if malware terminates before performing its malicious actions. An execution is considered *Succeeded* if malware accomplished its intended task, such as injecting malicious payload into an executable. The following outcomes are examples of hampered malware execution in the uncertain environment: (1) a virus that injects only part of the malicious code to an executable or source code file; (2) a botnet that loses commands sent to the bot herder; (3)

a cracker that retrieves wrong or partial user credentials; (4) a spyware that redirects incomplete stdin, stdout or stderr of the victim; (5) a flooder that sends only a percentage of the total number of packets it attempted.

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

Table 3-2 and Table 3-3 show the impact of intrusive and non-intrusive strategies in the uncertain environment for different types of malware for the thresholds of 10% and 50%. For each type of malware samples, threshold 50% caused about 20% more Crashed and 13% fewer Succeeded outcomes in the uncertain environment than threshold 10% for both intrusive and non-intrusive strategies. Non-intrusive strategies caused about 9% more Hampered and 5% fewer Crashed execution outcomes than intrusive strategies for different thresholds, showing as expected that non-intrusive strategies are less disturbing to malware than intrusive strategies.

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

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

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

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

Table 3-2. Execution for different types of malware under intrusive and non-intrusive strategies in the uncertain environment.

	threshold = 50%						threshold = 10%					
Malware	Intrusive			Non-intrusive			Intrusive			Non-intrusive		
category	Succ	Crash	Hamper	Succ	Crash	Hamper	Succ	Crash	Hamper	Succ	Crash	Hamper
Spyware	27%	67%	7%	40%	40%	20%	53%	27%	20%	60%	13%	27%
Viruses	24%	48%	28%	24%	40%	36%	24%	44%	32%	28%	36%	36%
Worm	21%	36%	43%	21%	57%	21%	29%	50%	21%	21%	29%	50%
Trojan	17%	63%	21%	29%	50%	21%	46%	29%	25%	38%	25%	38%
Flooders	9%	59%	32%	9%	50%	41%	41%	14%	45%	18%	36%	45%
All	19%	55%	26%	24%	45%	29%	38%	32%	30%	32%	29%	39%

Note: for the ratios of succeeded, crashed and hampered execution outcomes, Succ, Crash and Hamper were used.

Table 3-3. Execution for different types of benign software under intrusive and non-intrusive strategies in the uncertain environment.

	threshold = 50%						threshold = 10%					
Software	Intrusive			Non-intrusive			Intrusive			Non-intrusive		
category	Succ	Crash	Hamper	Succ	Crash	Hamper	Succ	Crash	Hamper	Succ	Crash	Hamper
Text editors	0%	73%	27%	33%	33%	33%	53%	20%	27%	73%	20%	7%
Compilers	18%	55%	27%	36%	18%	45%	55%	27%	18%	73%	18%	9%
Network tools	38%	56%	6%	50%	31%	19%	56%	19%	25%	56%	31%	13%
Scientific tools	33%	63%	3%	40%	43%	17%	53%	30%	17%	60%	20%	20%
Others	82%	7%	11%	79%	7%	14%	86%	7%	7%	86%	4%	11%
All	41%	47%	12%	51%	27%	22%	63%	20%	17%	70%	17%	13%

3.3.8 Behavior Comparison

Table 3-4 compares the execution of malware and benign software at the system call level in the uncertain environment. Modern software invoked more than twice the number of system calls monitored than malware, even with the existence of Flooders, which usually largely increased the average number of system calls invoked. For benign software the number of system calls perturbed or silenced was only half of those for malware, mainly because of the effectiveness of the corruption protection mechanism introduced in Section 3.3.4. Benign software had a larger

number of connection attempts and read/write operation monitored than malware. However, a smaller fraction of benign software system calls was perturbed in the uncertain environment.

Table 3-4. Comparison of malware on system call perturbation under the uncertain environment (with non-intrusive strategies at threshold 10%).

Malware category	Number of syscalls monitored	Percentage of syscalls perturbed	Percentage of syscalls silenced	Number of connection-related syscalls monitored	Percentage of connection-related syscalls perturbed	Number of buffer-related syscalls monitored	Percentage of buffer-related syscalls perturbed
Flooders	930.50	9.74%	3.39%	626.64	10.13%	91.18	6.58%
Spyware	50.73	2.89%	1.05%	4.67	7.14%	26.13	3.06%
Trojan	523.80	8.09%	2.85%	396.53	9.52%	182.67	7.14%
Viruses	423.44	5.02%	1.66%	211.75	9.56%	15.32	4.96%
Worms	68880.64	0.05%	0.02%	332.56	9.86%	43.79	8.97%
All	9992.49	0.41%	0.14%	266.16	9.87%	77.78	6.83%

Table 3-5. Comparison of benchmark software on system call perturbation under the uncertain environment (with non-intrusive strategies at threshold 10%).

Software category	Number of syscalls monitored	Percentage of syscalls perturbed	Percentage of syscalls silenced	Number of connection-related syscalls monitored	Percentage of connection-related syscalls perturbed	Number of buffer-related syscalls monitored	Percentage of buffer-related syscalls perturbed
Scientific tools	2071.59	1.13%	0.34%	4.00	0.00%	1878.28	0.46%
Compilers	167303.36	0.04%	0.01%	4.00	0.00%	164418.73	0.00%
Network tools	515.50	2.85%	1.01%	106.43	10.99%	206.38	1.54%
Others	566.31	0.54%	0.19%	2.00	0.00%	394.77	0.19%
Text editors	6693.20	0.42%	0.14%	3404.00	0.04%	4377.93	0.40%
All	20863.74	0.10%	0.03%	1283.64	0.40%	20023.82	0.03%

3.3.9 Case Study: Advanced Persistent Threat (APT)

In this section we show the evaluation of the interference strategies with an APT attack. We simulated a watering hole attack similar to the *Black Vine* APT from Symantec [89]. This attack has three main components: a Trojan, a backdoor and a keylogger. First, the attacker sends a spear-phishing e-mail to a user with a link for downloading the Trojan encryption tool. If the user clicks on the link and later uses the Trojan tool to encrypt a file, the tool downloads and executes a backdoor from a C&C server while encrypting the requested file. Then, the backdoor copies the directory structure and the ssh host key from the user's machine into a file and sends it to the C&C server. After the backdoor executes, the attacker deletes any traces of the infection without affecting the Trojan's encryption/decryption functionality. The attacker will also install a keylogger to obtain root privileges. Next, the backdoor runs a script that uploads sensitive data to the C&C server.

The Trojan is written in C using *libgcrypt* for encryption and decryption. It uses the curl library for downloading the backdoor from the Internet. In our simulation we used the *logkeys* keylogger from [90]. The backdoor script uses scp for sending the data to the C&C server.

Table 3-6. Execution details of the *Black Vine* APT in the standard and uncertain environment.

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

Note: under the uncertain environment, the experiment was carried out with intrusive and non-intrusive strategies for the 10% and 50% thresholds.

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

3.4 Discussion

As we discussed in Section 3.2, a resourceful and motivated adversary can bypass any protection mechanism. Even though the uncertain environment is designed to rate-limit stealthy

malware, it can still be eluded by attacks. For example, highly fault-tolerant malware will be resilient to the uncertain environment.

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

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

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

Although this approach was implemented for Linux (to allow the release of the open source code), it can also be implemented in other operating systems, such as Windows, which is a popular target of malware attacks.

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

CHAPTER 4
COMBINING TRADITIONAL MACHINE LEARNING AND DEEP LEARNING FOR REAL-TIME MALWARE DETECTION

4.1 Background

Malware is continuously evolving [91] and existing protection mechanisms have not been coping with their sophistication [92]. The industry still heavily relies on signature-based technology for malware detection [49], but these methods have many limitations, such as (i) they are effective only for malware with known signatures; (ii) they are not sustainable, given the massive amount of samples released daily; and (iii) they can be evaded by zero-day and polymorphic/metamorphic malware (practical detection 25%-50%) [66].

Behavior-based approaches attempt to identify malware behaviors using instruction sequences [51, 93], computation trace logic [94], and system or API call sequences [55, 95, 96]. These solutions have been mostly based on conventional machine learning (ML) models, such as K-nearest neighbor, SVM, neural networks, and decision tree algorithms [97–99]. However, current solutions based on ML and still suffer from high false-positive rates, mainly because of (i) the complexity and diversity of modern benign software and malware [52, 80, 91, 96], which are hard to capture during the learning phase of the algorithms; (ii) sub-optimal feature extraction; (iii) limited training/testing datasets, and (iv) the challenge of concept drift [100], which makes it hard to generalize learning models to reflect future malware behavior.

The accuracy of malware classification depends on gaining sufficient context information about software execution and on extracting meaningful abstraction of software behavior. For system/API-call based malware classification, longer sequences of calls likely contain more information. However, conventional ML-based detectors (e.g., Random Forest [101]) often use short windows of system calls during the training phase to avoid the curse of dimensionality (when the dimension increases, the classification needs more data to support and becomes harder to solve [102]), and may not be able to extract useful features for accurate detection. Thus, the main drawback of current behavioral-based approaches is that they might lead to low accuracy and many false-positives because it is hard to analyze complex and longer sequences of malicious

behaviors with limited window sizes, especially when malicious and benign behaviors are interposed.

In contrast, emerging deep learning (DL) models [103] are capable of analyzing longer sequences of system calls and making more accurate classification through higher level information extraction, while circumventing the curse of dimensionality. However, DL requires more time to gain enough information for classification and to predict the probability of detection. This trade-off is challenging: fast and perhaps not-so-accurate (ML methods) *vs.* time-consuming and more accurate classification (DL methods).

This chapter introduces and evaluates PROPEDEUTICA, a framework for efficient and effective on-the-fly malware detection, which combines the best of ML and DL methods. In PROPEDEUTICA, all software in the system is subjected to ML for fast classification. If a piece of software receives a borderline malware classification probability, it is then subjected to additional analysis with a more performance expensive and more accurate DL classification. The DL classification is performed via our newly proposed DEEPMALWARE algorithm. Compared to existing DL methods [104–106], DEEPMALWARE can learn spatially local and long-term features and handle software heterogeneity via the application of both (Atrous) convolutional neural networks and recurrent neural networks with multi-stream inputs.

The inspiration for our methodology is the practice of propedeutics in medicine, which refers to diagnosing a patient condition by first performing initial non-specialized, low-cost exams or by patient data collection based on observation, palpation, temperature measurement, and proceeding to specialized, possibly expensive and diagnostic procedures only if preliminary exams are inconclusive. This chapter first attempts to classify ("diagnose") a piece of software ("patient") as malicious ("experiencing a medical condition") using fast conventional ML (simple and non-expensive "diagnostic procedures"). If classification results are borderline ("inconclusive"), the software is subjected to accurate, but more performance expensive DL methods (complex, expensive "diagnostic procedures").

Evaluation of PROPEDEUTICA includes a set of 9,115 malware samples and 1,338 common benign software from various categories for the Windows OS. PROPEDEUTICA leveraged sliding windows of system calls for offline analysis and on-the-fly detection. In offline analysis, DEEPMALWARE achieved a 97.03% accuracy and a 2.43% false positive rate on average. In on-the-fly detection, considering the tradeoff between accuracy and detection time, PROPEDEUTICA achieved an 83.32% accuracy and a 12.56% false positive rate on average, with approximately 45% of the system software needed to be subjected to DEEPMALWARE (compare to 25-50% practical detection rates for current real time detectors [66]). Further, for the 45% of software with inconclusive classification, DEEPMALWARE achieved precision of 93.18% and false positive rate of 10.90%. With PROPEDEUTICA the whole-system on-the-fly detection F1-score increased 8.07% compared to pure ML and reduced the detection time by 40.50% compared to pure DL. There is a discrepancy between detection accuracy offline (analysis after all traces are collected) and on-the-fly (analysis in tandem with trace collection): for all learning models, the offline results were slightly better than on-the-fly results in terms of performance (accuracy, precision, F1-score and fp rate). For example, Random Forest (ML) experienced a decrease of 13% in accuracy when executed offline (89.05%) compared to online (78%) with the same traces.

This shows that real-time detection might differ from offline detection because in real-time there are interactions between the system and the classifiers that do not occur in offline analysis. These results provide evidence that ML and DL-based malware classifiers *in isolation* might not be able to meet the needs of high accuracy, low false positive rate, and short detection time in real-time malware detection.

This chapter presents the following contributions: (i) PROPEDEUTICA, a new framework for efficient and effective real-time malware detection for Windows OS, (ii) an evaluation of PROPEDEUTICA with a collection of 9,115 malware and 1,338 benign software, and (iii) DEEPMALWARE, a novel DL algorithm, learning multi-scale spatial-temporal system call features with multi-stream inputs that can handle software heterogeneity.

The remainder of this chapter is organized as follows. Section 4.2 describes the threat model. Section 4.3 describes PROPEDEUTICA architectural design. Section 4.4 discusses its implementation details. Section 4.5 presents our experimental evaluation.

4.2 Threat Model

We proposed PROPEDEUTICA to be deployed in corporate environments, which usually require precise, on-the-fly malware detection, i.e., accuracy greater than 90% and few false-positives (less than 5%).

We designed PROPEDEUTICA to protect organizations against the most prevalent malware threats, and not against directed attacks, such as specifically-crafted, advanced and persistent threats. Hence, we assume that malware will eventually get in if an organization is targeted by a well-motivated attacker. Therefore, attacks either targeting PROPEDEUTICA's software implementation, or to the machine learning engine (adversarial machine learning [107, 108]) are outside PROPEDEUTICA's scope.

PROPEDEUTICA operation assumes that, before it loads, the system is on a pristine state. Thus, PROPEDEUTICA will not scan all running processes at startup, which limits its checking procedure to each new process created.

The main goal of this chapter with PROPEDEUTICA is to present the concept of a multi-stage malware detector that combines two distinct classification models—ML and DL. Therefore, any presented implementation should be understood as a proof-of-concept (PoC) to accomplish such goal, and not as a platform-specific solution. We intend that this PoC leads to a PROPEDEUTICA version that will be implemented in the future by OS vendors to be integrated to their products.

For the sake of simplicity, our PoC was implemented in user land, i.e., as an user process in a non-privileged ring of execution. Thus, PROPEDEUTICA's trusted computing base includes the OS kernel, the learning models running in user land, and the hardware stack. We modelled PROPEDEUTICA's PoC for MS Windows, since it is the most targeted OS by malware writers [109]. We limited our PoC to the 32-bit MS Windows version, which currently is the most popular Windows architecture.

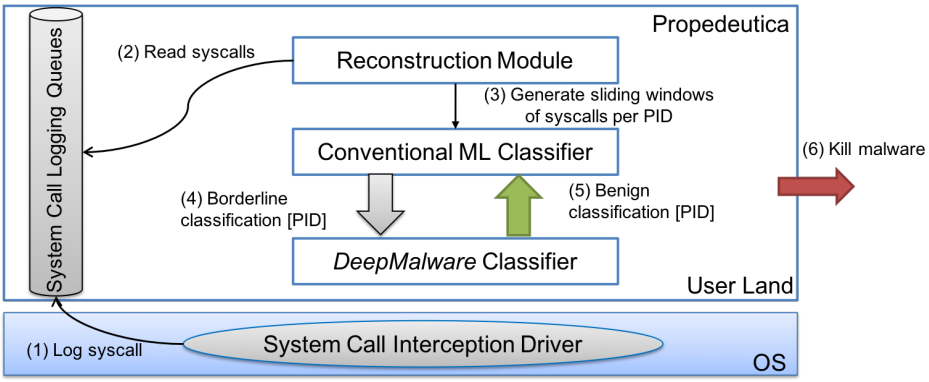

Figure 4-1. PROPEDEUTICA's architecture with steps performed for malware detection.

4.3 Architecture

PROPEDEUTICA (Figure 4-1) is composed of: (i) a system call reconstruction module, (ii) an ML classifier, and (iii) our newly proposed DEEPMALWARE classifier. PROPEDEUTICA also leverages a system call interception driver, which intercepts and logs all system calls invoked in the system (Step 1) and associate them with the PID of the invoking process.

The logged system calls are then read and parsed by the RECONSTRUCTION MODULE (Step 2), which compresses the system calls for input to the two classifiers. As input, these classifiers always receive the sliding windows of system call traces with the PID of the invoking process (Step 3).

Then, the ML classifier introduces a configurable borderline probability interval [lower bound, upper bound], which determines the range of classification that is considered inconclusive for detection. For example, let's consider that the borderline interval is [30%-70%]. If the ML classifier assigns the label "malware" for a piece of software whose classification range is within the predefined borderline interval, PROPEDEUTICA considers its classification result inconclusive. If the software receives a "malware" classification probability less than 30%, PROPEDEUTICA considers that the software is not malicious. If the classification probability is greater than 70%, PROPEDEUTICA considers the software malicious. For the inconclusive case ([30%-70%] interval), the software continues to run, but is subjected to analysis by DEEPMALWARE for a definitive classification (Step 4). If the software is classified as benign by

DEEPMALWARE, it continues running under monitoring of the ML classifier (Step 5), otherwise the software is considered malicious and will be killed (Step 6).

4.4 Implementation Details

This section discusses implementation details of three main aspects of PROPEDEUTICA's operation: the system call interception driver, the reconstruction module, and our newly proposed DEEPMALWARE algorithm.

4.4.1 The System Call Interception Driver

Intercepting system calls is a key step for PROPEDEUTICA operation as it allows collecting system call invocation data to be inputed to ML and DL classifiers. Despite many existing tools for process monitoring, such as Process Monitor [110], drstrace library [111], Core OS Tool [112], and WinDbg's Logger [113], we opted to implement our own solution to have more flexibility for deploying a real-time detection mechanism. PROPEDEUTICA collection driver was implemented for Windows 7 SP1 32-bit and will be publicly available at [114].

The driver hooks into the System Service Dispatch Table (SSDT), which contains an array of function pointers to important system service routines, including system call routines. In Windows 7 32-bit system, there are 400 entries of system calls [115], of which 155 system calls (listed in the Appendix A-1) are interposed by our driver. We selected a comprehensive set of system calls to be able to monitor multiple, distinct subsystems, which includes network-related system calls (e.g., *NtListenPort* and *NtAlpcConnectPort*), file-related system calls (e.g., *NtCreateFile* and *NtReadFile*), memory related system calls (e.g., *NtReadVirtualMemory* and *NtWriteVirtualMemory*), process-related system calls (e.g., *NtTerminateProcess* and *NtResumeProcess*), and other types (e.g., *NtOpenSemaphore* and *NtGetPlugPlayEvent*). To the best of our knowledge, this represents the largest Windows system call set that has been hooked by a driver in the literature [116–121].

System call logs (timestamp, PID, syscall) were collected using DbgPrint Logger [122], which enables real-time kernel message logging to a specified IP address (localhost or remote

IP), thus allowing the logging pool and the PROPEDEUTICA to reside on different hosts for scalability and performance.

The driver monitors all processes added to the *Borderline_PIDs* list, which keeps track of processes which received borderline classification from the conventional ML classifier.

4.4.2 The Reconstruction Module

In PROPEDEUTICA, both the ML and the DEEPMALWARE classifiers use the RECONSTRUCTION MODULE to preprocess the input system call sequences and obtain the same preprocessed data.

The RECONSTRUCTION MODULE splits system calls sequences according to the PIDs of processes invoking them and parses them into three types of sequential data: process-wide n-gram sequence, process-wide density sequence, and system-wide frequency feature vector. Then, the RECONSTRUCTION MODULE converts the sequential data into windows using the sliding window method, which is usually used to translate a sequential classification problem into a classical one for every sliding window [50].

Process-wide n-gram sequence and density sequence. We use the n-gram model, widely used in problems of natural language processing (NLP) [123], to compress system call sequences. n-gram is defined as a combination of n contiguous system calls. The n-gram model encodes low-level features with simplicity and scalability and compresses the data by reducing the length of each sequence with encoded information. The process activity in a Windows system can be intensive depending on the workload, e.g., more than 1,000 system calls per second, resulting in very large sliding windows. Therefore, for PROPEDEUTICA, we decided to further compress the system call sequences and translate them into two-stream sequences: n-gram sequences and density sequences. n-gram sequence is a list of n-gram units, while density sequence is the corresponding frequency statistics of repeated n-gram units. There are many possible n-gram variants such as n-tuple, n-bag, and other non-trivial and hierarchical combinations (e.g., tuples of bags, bags of bags, and bags of grams) [96]. PROPEDEUTICA uses 2-gram because (i) the n-gram model is considered the most appropriate for malware system call-based classification [96]

and (ii) the embedding layer and the first few convolutional neural layers can automatically extract high-level features from neighbor grams, which can be seen as hierarchical combinations of multiple n-grams. Once n-gram sequences fill up a sliding window, the RECONSTRUCTION MODULE delivers the window of sequences to either ML or DL analysis and redirects the incoming system calls to the new n-gram sequences.

System-wide frequency feature vector. Our learning models leverage system calls as features from all processes running in the system. This holistic, opposite to process-specific approach is more effective for malware detection compared to current approaches because modern malware can be multi-threaded [56, 124]. System-wide information helps the models learn the interactions among all processes in the system. To gain whole system information, the RECONSTRUCTION MODULE collects the frequency of different types of n-grams from all processes during the sliding window and extracts them as a frequency feature vector. Each element of the vector represents the frequency of an n-gram unit in the system call sequence.

4.4.3 DEEPMALWARE

PROPEDEUTICA aims to address spatial-temporal dynamics and software execution heterogeneity. Spatially, advanced malware leverages multiple processes to work together to achieve a long-term common goal. Temporally, a malicious process may demonstrate different types of behaviors (benign or malicious) during the lifetime of execution, such as keep dormant or benign at the beginning of the execution. To thoroughly consider available behavior data in space and time, this chapter used a novel DL model, DEEPMALWARE. To capture spatial features, DEEPMALWARE used multi-scale dilated convolutional neural networks [125]. To capture temporal features, DEEPMALWARE used bi-directional LSTM [126]. To facilitate harmonious coordination, DEEPMALWARE stacks spatial and temporal models for joint training.

System call sequences are taken as input for all processes subjected to DEEPMALWARE analysis (see Figure 4-2 for a workflow of the classification approach). DEEPMALWARE leverages n-gram sequences of processes and frequency feature vectors of the system. First, two streams (process-wide n-gram sequence and density sequence) model the sequence and density

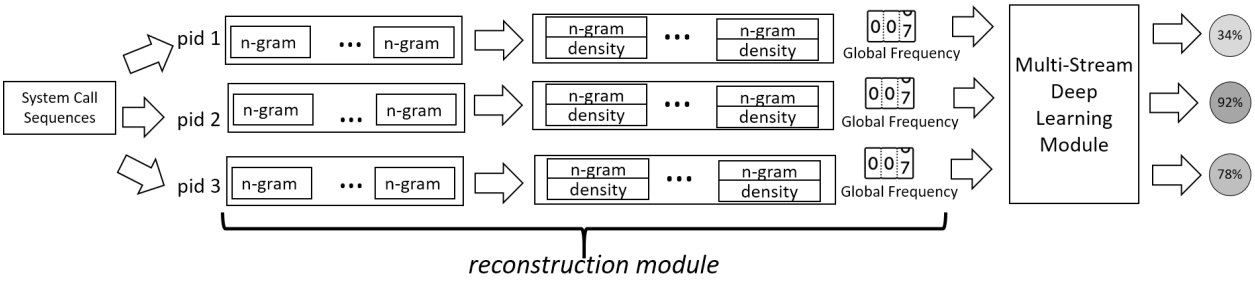

Figure 4-2. Workflow of DEEPMALWARE classification approach.

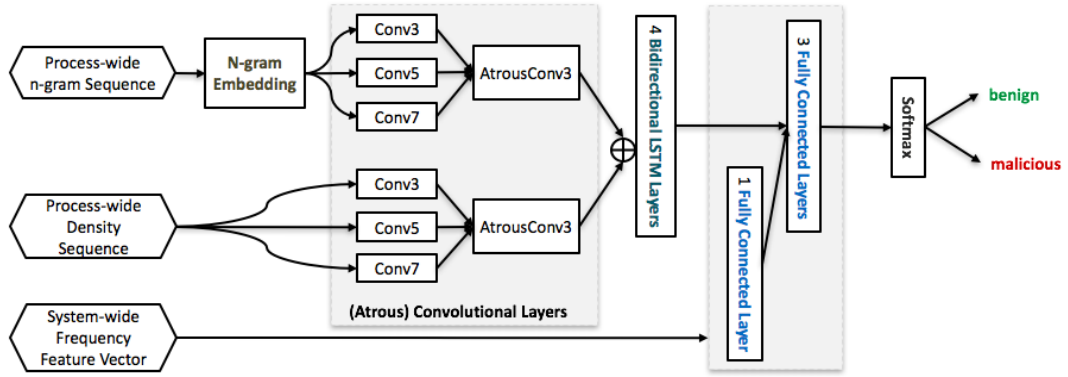

Figure 4-3. DEEPMALWARE architecture.

of n-gram system calls of the process. The third stream represents the global frequency feature vector of the whole system. DEEPMALWARE consists of four main components: (i) N-gram Embedding, (ii) (Atrous) Convolutional Layers (Conv3, Conv5, Conv7, and AtrousConv3), (iii) Long Short-Term Memory (LSTM) Layers, and (iv) Fully Connected Layers (Figure 4-3).

N-gram Embedding. DEEPMALWARE adopts an encoding scheme called N-gram Embedding, which converts sparse n-gram units into a dense representation. DEEPMALWARE treats n-gram units as discrete atomic inputs. N-gram Embedding helps to understand the relationship between functionally correlated system calls and provides meaningful information of each n-gram to the learning system. Unlike vector representations such as one-hot vectors (which may cause data sparsity), N-gram Embedding mitigates sparsity and reduces the dimension of input n-gram units. The embedding layer maps sparse system call sequences to a dense vector. This layer also helps to extract semantic knowledge from low-level features (system call sequences)

and largely reduces feature dimension. The embedding layer uses 256 neurons, which reduce the dimension of the n-gram model (number of unique n-grams in the evaluation) from 3,526 to 256.

(Atrous) Convolutional Layers. Conventional sliding window methods leverage small windows of system call sequences and, therefore, experience challenges when modeling long sequences. These conventional methods represent features in a rather simple way (e.g., only counting the total number of system calls in the windows without local information of each system calls), which may be inadequate for a classification task. DEEPMALWARE applies multi-scale convolutional kernel filters [127] on the same inputs in parallel to gain a wider view on the system calls and to concatenate the output features. Features are extracted with different receptive fields, and lower and higher level features are fused together. This design has been proved to be robust and to accelerate large-scale image classification time. Thus, DEEPMALWARE uses convolutional layers with kernel size 1×3, 1×5, and 1×7 , and applies padding on each convolutional branch to keep their lengths aligned.

DEEPMALWARE uses the Astrous convolutional layer [128] (a variant of convolutional layers) after the inception model to enlarge the field-of-view on filters and employs spatially small convolution kernels to keep both computation and number of parameters contained without increasing the number of parameters and the amount of computation.

DEEPMALWARE leverages batch normalization after convolutional layers to speed up the training process [129] and a non-linear activation function, ReLU to avoid saturation.

Long Short-Term Memory (LSTM) Layers. The internal dependencies between system calls include meaningful context information or unknown patterns for identifying malicious behaviors. To leverage this information, DEEPMALWARE adopts one of the recurrent architectures, LSTM [126], for its strong capability on learning meaningful temporal/sequential patterns in a sequence and reflecting internal state changes modulated by the recurrent weights. LSTM networks can avoid gradient vanishing and learn long-term dependencies by using forget gates. LSTM layers gather information from the first two streams: process-wide n-gram sequence and density sequence.

Fully Connected Layers. DEEPMALWARE deploys a fully connected layer is deployed to encode system-wide frequency. Then, it is concatenated with the output of bi-directional LSTMs to gather both sequence-based process information and frequency-based system-wide information. The last fully-connected layer with softmax activation function outputs the prediction.

4.5 Evaluation

The main goal of our evaluation is to determine PROPEDEUTICA's effectiveness for real-time malware detection. More specifically, we sought to discover to what extent PROPEDEUTICA outperforms ML on accuracy and DL on classification time.

We first describe the dataset used in our evaluation. Next, we evaluate the performance of our newly proposed DEEPMALWARE algorithm in comparison with the most relevant ML and DL classifiers from the literature. Then, we present the results of our experiments with PROPEDEUTICA operating offline (classifying post-processed traces), and in on-the-fly (classifying traces collected in real-time). In our evaluation, we compare the performance of DL classifiers when operating on a CPU and a GPU. GPUs not only reduce the training time for DL models but also help these models achieve classification time one order of magnitude less than conventional CPUs.

DEEPMALWARE was implemented by Pytorch [130]. The training and testing procedures ran on an Ubuntu 14.04 Server with four Nvidia Tesla K80 GPUs and 56 Intel(R) Xeon(R) E5-2695 CPU cores.

4.5.1 Dataset

Our dataset is divided in two classes of samples, labeled as malicious and benign. For the malicious part, the dasatet was composed of 9,115 Windows PE32 format malware samples collected between 2014 and 2018 from threats detected by the incident response team of a major financial institution (who chose to remain anonymous) in the corporate email and Internet access links of the institution's employees, and 7 APTs collected from Rekings [131]. Figure 4-4 shows

the categorization of our malware samples using AVClass [132][1] . Some malware samples, however, caused blue screen of death (BSOD) before the five minutes had elapsed. Thus, we set a minimum running time of one minute for a malware sample to be included in our analysis. For the benign part, our dataset was composed of 1,338 samples, including 866 Windows benchmark software, 50 system software, 11 commonly used GUI-based software, and 409 other benign software. GUI-based software (e.g., Chrome, Microsoft Office, Video Player) had keyboard and mouse operations simulated using WinAutomation [133].

In each experiment, we run malware and benign software for five minutes, during which we collect system-wide system calls. The experiments are carried out under different scenarios: 1) running one general malware, one benchmark/GUI-based/other benign software, and dozens of system software. 2) running two general malware, several benchmark/GUI-based/other benign software, and dozens of system software. 3) running one APT, one benchmark/GUI-based/other benign software, and dozens of system software.

We randomly sampled half malware and benign software samples for training and half for testing, which is a standard procedure in machine learning to avoid overfitting [134]. We collected 493,095 traces in total. During the running of one malware sample, multiple benign processes (especially system processes) would run concurrently. Therefore, the number of benign process executions we collected would be much larger than that of malicious process executions. Thus, our classifier would end up handling imbalanced datasets, which could adversely affect the training procedure, because the classifier would be prone to learn from the majority class. We, therefore, under-sampled the offline dataset by reducing the number of sliding windows from benign processes to the same as malware processes.

[1] AVClass is an automatic, vendor-agnostic malware labeling tool. It provides a fine-grained family name rather than a generic label (e.g., Trojan, Virus, etc.), which contains scarce meaning for the malware sample.

Window size and stride are two important hyper-parameters in sliding window methods, indicating the total and the shifting number of n-gram system calls in each process. In on-the-fly detection, large window sizes need more time to fill up the sliding window, provide more context for DL models, and therefore achieve higher accuracy. We aim to choose the proper two hyper-parameters to ensure malware are detected accurately and in time (before completing its malicious behavior). In our experiments, we observed that malware can successfully steal user information in one to two seconds, and 1000 system calls can be invoked in one second. Therefore, we selected and compared three window size and stride pairs: (100, 50), (200, 100), (500, 250). The results showed that with pair (500, 250), DEEPMALWARE needs 1 to 2 seconds to fill up the window and analyze. With pair (200, 100), DEEPMALWARE needs 0.5 to 1 second. While with pair (100, 50), DEEPMALWARE needs only 0.1 to 0.5 seconds. The F1-scores for DEEPMALWARE are 97.02% , 95.08% and 94.97% respectively. Consider that a window size of 500 may not be able to give the detection result until the malware almost finishes execution, we chose the window size 100 and stride 50 in the following evaluation.

We included not only benchmark software, but also GUI-based software which are challenging to configure and automate because of the need to simulate user activity. Anubis collected hybrid traces from benign and malicious behaviors and used to play a major role in many research projects, which, however, is not public anymore.

4.5.2 Offline Post-processing of Traces

We started by comparing the performance of DEEPMALWARE with relevant ML and DL classifiers in isolation (offline post-processing), i.e., when the system call traces from malware and benign software are completely collected before being analyzed by the models. Our metrics for model performance were accuracy, precision, recall, F1 score, and false positive (fp) rate.

For ML, we considered the following algorithms: Random Forest (RF), eXtreme Gradient Boosting (XGBoost) [135], and Adaptive Boosting (AdaBoost) [136]. Random Forest, neural networks, and boosted trees have been shown to be the best three supervised models on high-dimensional data [137]. We used AdaBoost and XGBoost, as representatives of boosted trees.

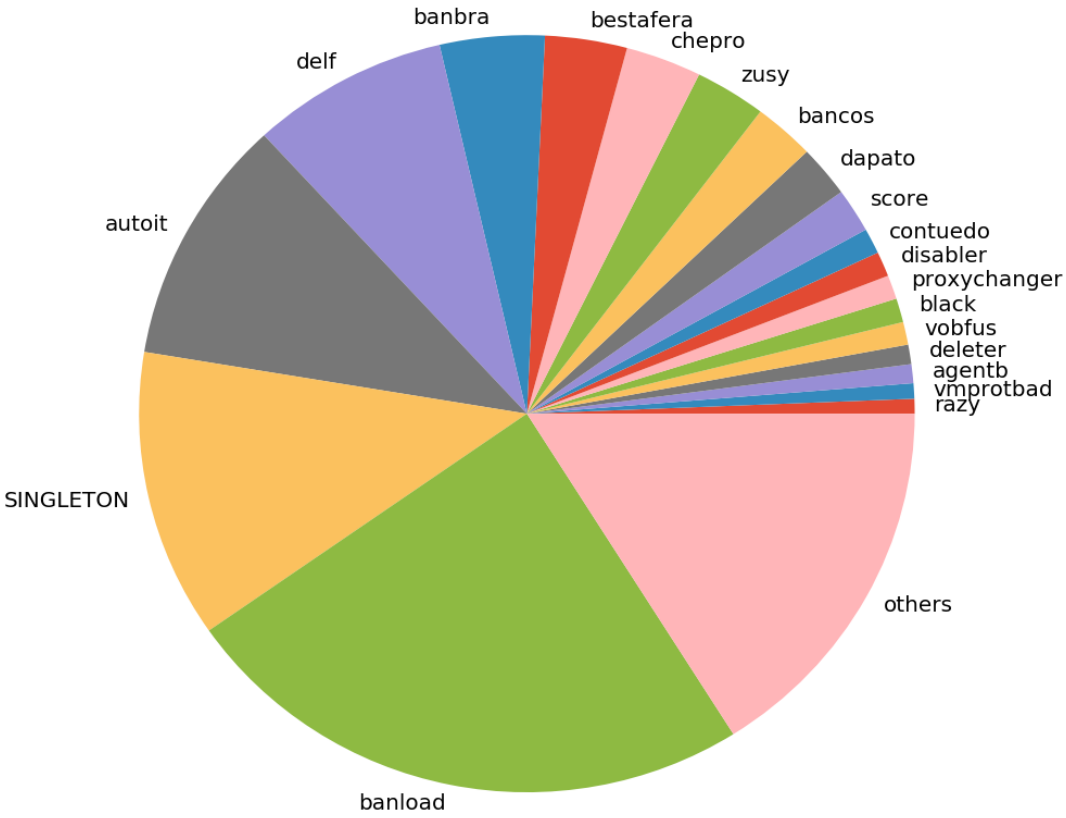

Figure 4-4. Malware families in our dataset classified by AVClass. Banload and Banbra install other malware to steal banking credentials. AutoIt distributes files attached to fake (IM) messages or emails. Delf and Bestafera capture private data, such as keystrokes. Chepro invokes Control Panel applets, takes screenshots, registers keystrokes, and reads the contents of the clipboard. Bancos targets banking websites. Singleton means AVClass could not classify the malware.

The input features of these algorithms are the frequency of n-gram in a sliding window of system calls belonging to a process.

For DL, we considered DEEPMALWARE (our newly proposed algorithm), and two DL control models—CNNLSTM, LSTM. Compared to DEEPMALWARE (Section 4.4.3), CNNLSTM and LSTM do not have dilated and convolutional layers respectively, which lack sufficient spatial information.

PROPEDEUTICA used Random Forest for ML classification and DEEPMALWARE DL classification. As described later in this section, we chose these algorithms because they produced the best overall performance when executed in isolation (see Table 4-1 for details).

We leverage sliding windows for classification, in which two important parameters are the window size and stride. The window size indicates the total number of system calls in a given window. The window stride indicates how many system calls are skipped between adjacent windows. Ideally, the larger the window size and the smaller the stride, the more context information are available to the classifier. However, smaller windows and larger strides lead to an increase in prediction time. Thus, there is a trade-off between accuracy and performance that must be taken into account. We selected and compared three window size and stride pairs: (100, 50), (200, 100), (500, 250).

Table 4-1 shows the results. We compared the detection time for each algorithm using CPU and GPU. We did not apply GPU devices to ML models because their classification time with conventional CPUs is much smaller (at least one order of magnitude) than those measured for DL algorithms. We denote 'N/A' as not applicable in Table 4-1. In real life, GPU devices, especially specific DL GPUs for accelerating DL training/testing are still not widespread in end-user or corporate devices. Thus, anti-malware solutions for such public currently do not rely on GPUs.

The analysis of deep learning approaches reveals that DEEPMALWARE is in the best case almost 2 percentage points more accurate while being marginally faster (0.01s) than the current approach (CNNLSTM). With window size 100 and stride 50, DEEPMALWARE achieved an accuracy of 94.84% and false positive rate 7.76%. With window size 500 and stride 250, DEEPMALWARE achieved an accuracy of 97.03% and false positive rate 2.43%. The performance of DEEPMALWARE increased with the increase of window size and stride.

For machine learning models in isolation, Random Forest performs the best with an accuracy of 89.05% at window size 100 and stride 50, and an accuracy of 93.94% at window size 500 and stride 250, approximately 10% better than AdaBoost and 5% better than XGBoost. Random Forest also achieves the fastest speed among all the machine learning methods. While DEEPMALWARE model is 3.09% more accurate than Random Forest, its classification time is approximately 100 times slower on CPU and 3 to 5 times slower on GPU.

We observed that malware could successfully steal user information in one to two seconds, and 1,000 system calls can be invoked in one second. Moreover, traces from one process may take a significant amount of time to fill up one sliding window because the process might be invoking system calls at a low rate. Thus, in the remainder of this section, we chose to compare all the algorithms using window sizes of 100 and strides of 50 system calls. For all window sizes and strides, the best ML model was Random Forest and the best DL model was DEEPMALWARE. Therefore, for PROPEDEUTICA, we chose Random Forest for ML classification and DEEPMALWARE for DL classification.

Table 4-1. Comparison between ML and DL models in isolation and in offline analysis.

	Model	Window size	Stride	Accuracy	Precision	Recall	F1 score	FP rate	Time GPU (s)	Time CPU (s)
	AdaBoost	100	50	79.25%	78.11%	81.31%	79.67%	22.80%	N/A	0.0187
ML	Random Forest	100	50	89.05%	84.63%	95.44%	89.71%	17.35%	N/A	0.0089
	XGBoost	100	50	84.78%	90.99%	77.22%	83.54%	7.66%	N/A	0.0116
	CNNLSTM	100	50	94.82%	92.98%	96.96%	94.93%	7.32%	0.0410	1.5533
DL	LSTM	100	50	94.10%	90.91%	98.01%	94.32%	9.81%	0.0098	2.8076
	DEEPMALWARE	100	50	94.84%	92.63%	97.43%	94.97%	7.76%	0.0383	1.4104
	AdaBoost	200	100	78.27%	72.84%	90.19%	80.59%	33.64%	N/A	0.0132
ML	Random Forest	200	100	93.49%	89.99%	97.90%	93.77%	10.91%	N/A	0.0063
	XGBoost	200	100	70.81%	63.65%	97.08%	76.89%	70.81%	N/A	0.0073
	CNNLSTM	200	100	95.78%	93.54%	98.36%	95.89%	6.80%	0.0492	1.4332
DL	LSTM	200	100	94.86%	92.96%	97.08%	94.97%	7.36%	0.0947	2.6715
	DEEPMALWARE	200	100	94.96%	93.05%	97.20%	95.08%	7.27%	0.0543	1.3784
	AdaBoost	500	250	81.81%	79.44%	85.86%	82.52%	22.24%	N/A	0.0108
ML	Random Forest	500	250	93.94%	97.16%	90.54%	93.73%	2.65%	N/A	0.0048
	XGBoost	500	250	79.19%	94.14%	62.27%	74.95%	3.88%	N/A	0.0062
	CNNLSTM	500	250	95.33%	97.20%	93.34%	95.23%	2.69%	0.0902	1.8908
DL	LSTM	500	250	95.81%	97.38%	94.16%	95.74%	2.54%	0.1529	3.5626
	DEEPMALWARE	500	250	97.03%	97.54%	96.50%	97.02%	2.43%	0.0797	1.3344

Note: DEEPMALWARE is in the best case almost 2% more accurate and marginally faster (0.01s) than the other deep learning models. Random Forest is approximately 8% more accurate and much faster than the other machine learning models.

Next, we evaluated PROPEDEUTICA's performance for different borderline intervals offline. As discussed before, in PROPEDEUTICA, if a piece of software receives a malware classification probability from Random Forest that is smaller than the lower bound or greater than the upper bound, it is considered benign software or malware respectively. However, if the classification falls within the borderline interval, the software is subjected to DEEPMALWARE. Table 4-2 shows the performance of PROPEDEUTICA using various (configurable) borderline intervals. We chose lower bound as 10%, 20%, 30%, and 40%, upper bound as 60%, 70%, 80%, and 90%.

Table 4-2. Comparison on different borderline policies for the PROPEDEUTICA in offline analysis with window size 100 and stride 50.

Lower bound	Upper bound	Accuracy	Precision	Recall	F1	FP rate	Classification time with GPU (s)	Classification time with CPU (s)	Move percentage
10%	90%	94.71%	92.41%	97.43%	94.85%	8.00%	0.0223	0.3810	55.95%
20%	80%	94.61%	92.23%	97.43%	94.76%	8.21%	0.0173	0.1543	48.32%
30%	70%	94.34%	91.77%	97.43%	94.51%	8.75%	0.0146	0.0884	41.45%
40%	60%	93.72%	90.79%	97.37%	93.96%	9.92%	0.0123	0.0497	34.96%

Note: borderline intervals are described with a lower bound and an upper bound. The move percentage represents the percentage of software in the system that received a borderline classification with Random Forest (according to the borderline interval) and was subjected to further analysis with DEEPMALWARE.

With a borderline interval of [30%-70%], PROPEDEUTICA achieved an accuracy of 94.34% and a false positive rate of 8.75%, with approximately 40% of the samples moved for DEEPMALWARE analysis. This highlights the potential of PROPEDEUTICA: approximately 60% of the samples were quickly classified with high accuracy as malware or benign software using an initial triage with faster Random Forest.

4.5.3 On-the-Fly Processing of Traces

In this section, we show our experiments for malware classification using PROPEDEUTICA on-the-fly, i.e., analysis performed by traces gathered in tandem with software execution. We randomly sampled 10% of the malware and benign software used in the offline testing dataset for the on-the-fly detection. In each experiment, we run one malware, one benign software and many other system software.

For all the configurable borderline intervals of classification, on average, PROPEDEUTICA improved the detection accuracy from 78% (Random Forest) to 86.37% (8.37% increase), and reduced the detection time by 44.15% (DEEPMALWARE). Furthermore, the amount of software that were subjected to DL analysis was just a fraction (approximately 35% on average for online detection) of the software running in the system, as ML was able to provide a high probability for part of the clear-cut cases.

We also found a discrepancy between the offline and online results: for all learning models, the offline results are slightly better than the on-the-fly results for all metrics (accuracy, precision, F1-score, fp rate, etc.). We hypothesize that the reasons behind these differences could be the

real-time interactions, among the ML and DL classifiers, and the system processes, such as the change of environments. These interactions for on-the-fly analysis can increase the system overhead, in which more processes are opened and closed during each experiment. To test this hypothesis, we measured the total number of processes monitored for each experiment and found that there was approximately an 50% increase in the number of processes for the online experiments. Another result that corroborates our hypothesis is that the system call interception driver collected fewer system calls for the on-the-fly experiments compared to offline. For example, for the on-the-fly experiments, in 5-minute experiments, many processes could not fill up one sliding window, i.e., the total number of collected n-gram system calls is less than the window size. This brings less context to learn models and negatively affect classification.

Table 4-3. Comparison between ML and DEEPMALWARE for on-the-fly detection.

Model	Upper bound	Lower bound	Accuracy	Precision	Recall	F1	FP rate	Detection time (s)
Random forest	N/A	N/A	78.00%	79.18%	75.79%	77.54%	20%	305.88
DEEPMALWARE	N/A	N/A	89.20%	93.18%	85.99%	89.41%	10.90%	727.96
PROPEDEUTICA	10%	90%	87.00%	88.23%	85.39%	86.79%	11.39%	518.24
	20%	80%	84.07%	86.94%	80.20%	83.43%	11.38%	447.18
	30%	70%	78.72%	85.38%	82.62%	81.91%	13.48%	406.73
	40%	60%	83.48%	85.27%	80.95%	83.05%	13.99%	361.04

Note: the detection time means the duration from the beginning of the malware execution to PROPEDEUTICA detecting of the malware. The detection time may be longer than the experiment length, as in this case, the system is still running without collecting further traces. All the experiments leveraged a conventional CPU.

We evaluated PROPEDEUTICA's performance on seven APTs with various borderline intervals. PROPEDEUTICA successfully detected all the APTs and six of them were subjected to DEEPMALWARE classification. For different borderline intervals, the false positive rate was approximately 10%. Random Forest in isolation detected one APT with a 52.5% false positive rate.

We measured the performance overhead of the system call interception mechanism leveraged by PROPEDEUTICA. This test was performed on a server running Ubuntu 12.04 with 2 CPUs, 4GB Memory, 60GB Disk. Our results show the following performance overheads: 24.2%

Table 4-4. Comparison among DEEPMALWARE and other DL-based malware detection in the literature.

Work	Model	Dataset	Performance
DMIN'16 [104]	DL4MD	45,000 traces	Offline: Accuracy 95.65%, Precision 95.46%, Recall 95.8%
KDD'17 [105]	DNN	26,078 traces	Offline: Accuracy 93.99%
LNCS'16 [106]	LSTM	4,753 traces	Offline: Accuracy 89.4%, Precision 85.6, Recall 89.4%
PROPEDEUTICA	DEEPMALWARE	493,095 traces	Offline: Accuracy 97.0%, Precision 97.5%, Recall 97.0% Online: Accuracy 89.2%, Precision 93.2%, Recall 86.0%

Note: in these experiments, DEEPMALWARE operated both in offline analysis (window size 500, stride 250) and on-the-fly (window size 100, stride 50). For offline analysis we chose to compare the window size and stride that DEEPMALWARE performed the best. For on-the-fly detection, window size 100 and stride 50 were chosen because of the tradeoff between window size and stride (see discussion in Section 4.5.1).

increase in CPU usage, 0.21% increase in bytes written into memory, 4.03% increase in the number of threads, and 2.03% increase in disk usage, compared with experiments running without PROPEDEUTICA. This overhead will not affect user experience.

Finally, we compared PROPEDEUTICA with other DL-based malware detectors in the literature leveraging system/API calls as features. Table 4-4 summarizes the results we obtained. Current literature mainly focuses on offline analysis [104–106] and lacks analysis on the detection time in real-time. DEEPMALWARE in offline analysis can outperform those approaches with 97.03% accuracy and 97.54% precision.

4.6 Discussion

Real-time detection of malware samples seen in the wild is challenging. On the one hand, mission-critical software cannot be mistakenly killed by a malware detector. On the other hand, the detector should not risk allowing possible malware entering the organization perimeter. Besides, real-time malware detection incurs extra workloads and more computational pressure by the interactions between the detector, its supporting system, and the processes opened.

PROPEDEUTICA leverages the fast speed of ML and the high accuracy of emerging DL models. Only software receiving borderline classification from a ML detector needed further analysis by DEEPMALWARE, which saved computational resources, shortened the detection time, and improved accuracy.

Our results showed that PROPEDEUTICA offline analysis achieved a 97.03% accuracy and a 2.43% false positive rate. PROPEDEUTICA on-the-fly detection considered the trade-off between accuracy and detection time. For all configurable borderline intervals of classification, on average, PROPEDEUTICA achieved an 83.32% accuracy and a 12.56% false positive rate, with approximately 45% of the system software needed to be subjected to DEEPMALWARE (compare to 25-50% practical real time detection rates for current detectors [66]). PROPEDEUTICA on-the-fly detection increased the detection F1-score by 8.07% compared with Random Forest, and reduced the detection time by 40.50% compared with DEEPMALWARE. For the 45% of software with inconclusive classification, PROPEDEUTICA achieved accuracy of 93.18% and false positive rate of 10.90%.

For all learning models, we found a discrepancy between the offline and online results: the offline results are slightly better than the on-the-fly results for all metrics (accuracy, precision, F1-score, fp). For example, Random Forest (ML) experienced a decrease of 13% in accuracy when executed offline (89.05%) compared to online (78%) with the same traces.

Despite PROPEDEUTICA promising results, malware detection in real time is still challenging because it incurs extra workloads and more computational pressure by the interactions between the detector, its supporting system, and the processes opened.

Further, PROPEDEUTICA can run into a worst-case scenario, which is having a process continuously loop between Random Forest and DEEPMALWARE. For example, consider a process receiving a borderline classification from Random Forest, and then being moved to DEEPMALWARE. Then DEEPMALWARE classifies it as benign, and the software would continue being analyzed by Random Forest, which again provides a borderline classification for the process, and so on.

As we discussed in Section 4.2, a resourceful and motivated adversary can bypass any protection mechanism. For example, a compromised OS with adversarial examples [138, 139] could affect PROPEDEUTICA's performance.

Recent work has demonstrated that DL/ML-based models are vulnerable to misclassification by well-designed data samples in the test phase, called *adversarial examples* [138, 139]. Some studies leveraged generated binary malware inputs [140–142], or generated adversarial sequences [143] to attack dynamic behavior-based malware detection systems. Papernot *et al.* proposed network distillation to defend adversarial examples [144], however, it is still vulnerable to C&W attacks [138].

The key contribution of PROPEDEUTICA is the intelligent combination of ML with emerging modern DL methods in an effective real-time detector, which can meet the needs of high accuracy, low false-positive rate, and short detection time required to counter the next generation of malware detection.

The system call interception driver was implemented for Windows 7 32-bit for the sake of simplicity, as it allows us to directly hook SSDT without major restrictions. Modern 64-bit Windows versions prevent SSDT hooking due to the Kernel Patch Protection (KPP) mechanism, which requires developers using alternative ways of intercepting kernel events [145]. We consider this limitation acceptable as our main goal is to propose the use of multi-stage classifiers for effective malware detection. We believe that the PROPEDEUTICA long-term adoption will be streamlined by OS developers without the restrictions for third-part code development.

The collection driver is also limited to a subset of system calls officially documented by Microsoft. We found that unofficial documentation about system call parameters could be misleading and, in some cases, lead to Blue Screen Of Death (BSOD). Thus, we decided to collect only information about system calls that would not cause BSOD when intercepted. We also believe that PROPEDEUTICA deployment by OS developers will fix these issues and streamlines its operation in actual scenarios.

In the future, we plan to thoroughly investigate the causes for the discrepancy between offline and on-the-fly malware classification performance and improve PROPEDEUTICA's resilience against machine learning adversarial examples.

CHAPTER 5
CONCLUSIONS

5.1 Summary

This dissertation addressed the problem of system predictability as an enabler of software attacks and how uncertainty can be leveraged to improve system security and software reliability. Specifically, three frameworks: BEAR, CHAMELEON, and PROPEDEUTICA, are developed as part of this dissertation, and their contributions and results are summarized in the following.

- BEAR, a framework for statistical analysis of program sensitivity to OS unpredictability, has the potential to provide invaluable insights to developers. In particular, we found that developers should be careful on handling buffer type and return length of OS system calls, considering how commonly used `read` and `write` system calls are, especially when application workload is heavy. Developers and testers should balance their time not only based on how severe the impact of a system call or a strategy, but also on how frequently it is invoked. If resources are limited, failure-oblivious computing can be a promising way for saving developers and testers from memory bugs. Given an option of more than one system calls for the same functionality, developers should select system calls that are specialized, taking a buffer of vector inside a parameter struct and having the larger parameter list.

- CHAMELEON, a novel Linux framework that introduced uncertainty as an OS built-in feature to rate-limit the execution of borderlined software—possible malware that received a borderline classification by traditional ML detectors, while a second performance expensive deep-learning detector is operating. CHAMELEON's protection target are organizations, where it is a common practice to whitelisted software to run in the organization perimeter. CHAMELEON's evaluation showed that a threshold of 10% caused various levels of disruption to 30% of the analyzed software. Malware was affected more with intrusive strategies caused 62% and non-intrusive strategies caused 68% of the malware to fail to accomplish their tasks. Further, the uncertain environment adversely affected malware even more when the threshold increased. We also found that I/O-bound software was three times more affected by uncertainty than CPU-bound software.

- PROPEDEUTICA, a framework for real-time software classification (malware detection) combining the knowledge in cutting-edge machine learning and deep learning techniques. The results showed that, on average, 45% of the system software needed to be subjected to DEEPMALWARE with all configurable borderline intervals of classification probability. The on-the-fly detection increased F1-score by 8.07% compared to pure ML and reduced the detection time by 40.5% compared to pure DL. For the 45% of software with inconclusive classification, DEEPMALWARE achieved a precision of 93.18% and false positive rate of 10.90%. CHAMELEON's evaluation provided evidence that conventional machine learning and emerging deep learning methods in isolation might be inadequate to provide a combined high accuracy and short detection time required for real-time malware detection.

PROPEDEUTICA, combining the best such methods has the potential to transform the next generation of practical on-the-fly malware detection.

To sum up, 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. Investigating the effectiveness of system uncertainties is a promising avenue in future security research.

5.2 Future Work

The future work is warranted to address the challenges remaining in this dissertation and vision of using uncertainty to improve software reliability and security. For example, interesting pieces of future work would be (i) port BEAR's framework to other types of OSes using portable/generic system calls, (ii) provide insights with a finer granularity for non-normal execution results, e.g. segmentation faults, concurrency issues, (iii) control and measure stress responses from end-users working with the CHAMELEON system, and (iv) explore the discrepancy between detection accuracy offline (analysis after all traces are collected) and on-the-fly (analysis in tandem with trace collection) in PROPEDEUTICA. In particular, this dissertation proposes the following avenues that warrant further research investigation.

- **Fine-grained analysis of the consequences of uncertainty in software.** In Chapter 2 (BEAR framework), we analyzed program sensitivity to OS unpredictability as a function of a program crashing or executing without issues. It would be interesting to broaden BEAR to analyze program fragility to uncertainty in a finer grained style. Crashes could be differentiated by exit with an error, segmentation fault, concurrency issue, etc. Furthermore, in some cases, in spite of a program undergoing uncertainty could run without experiencing a crash, it might present senile behavior, e.g. threads taking longer time to terminate, file locks taking longer time to release. These abnormal results, potentially raised by different issues, should be analyzed in different ways. Previous work [146, 147] mainly focused on resource usage analysis to predict the execution results. However, the abnormal behaviors are not necessarily correlated to increased memory or CPU usages, thus making it challenging to explore the reasons behind. Future work should consider system call parameters, which can provide detailed information about each system call, and system-wide information including processes information co-located with the process of interest.

- **User study assessing CHAMELEON's usability**. Leveraging uncertain strategies is not a one-size-fits-all solution—we need information from real users to improve CHAMELEON's

performance. We discussed in Section 3.3.5 about the trade-off of intrusive and non-intrusive strategies and recommended the involvement of system administrators. However, typical business organizations consist of many departments: production, research and development, purchasing, and marketing, human resource management, accounting and finance. With each department having its own functionalities, the burden of system administrators could be potentially heavy. Therefore, future study targeting automatic adjustment of perturbation strategies and thresholds is warranted. Such study should investigate, for instance, whether benign software is mistakenly perturbed with unacceptable frequency, thus affecting the regular work of employees. Such insights would be invaluable to fine-tune the degree at which CHAMELEON should apply perturbations.

- **Applying perturbation strategies to other OSes.** This dissertation discussed CHAMELEON's framework under Linux, and PROPEDEUTICA's framework under Windows, while both paradigms could be implemented in both OSes, potentially even Mobile OSes. With the boost of designated mobile AI chips, such as Apple's neural engine in the new iPhone, Huawei's neural processing unit in Mate 10, and Qualcomm's and ARM's gearing up to supply AI-optimized hardware [148], it is promising to deploy deep learning and machine learning models on mobile system in the future.

APPENDIX A
SYSTEM CALLS

Table A-1. System call set being hooked by PROPEDEUTICA.

System call name	System call name	System call name
oldNtCreateThread	oldNtCreateThreadEx	oldNtSetContextThread
oldNtCreateProcess	oldNtCreateProcessEx	oldNtCreateUserProcess
oldNtQueueApcThread	oldNtSystemDebugControl	oldNtMapViewOfSection
oldNtOpenProcess	oldNtCreateProcess	oldNtCreateProcessEx
oldNtOpenThread	oldNtQuerySystemInformation	oldNtSetInformationFile
oldNtQueryInformationFile	oldNtCreateMutant	oldNtDeviceIoControlFile
oldNtTerminateProcess	oldNtDelayExecution	oldNtQueryValueKey
oldNtQueryAttributesFile	oldNtResumeThread	oldNtCreateSection
oldNtLoadDriver	oldNtClose	oldNtOpenFile
oldNtNotifyChangeMultipleKeys	oldNtQueryMultipleValueKey	oldNtQueryObject
oldNtRenameKey	oldNtSetInformationKey	oldNtAllocateLocallyUniqueId
oldNtCreateDirectoryObject	oldNtCreateKey	oldNtCreateKeyTransacted
oldNtSetQuotaInformationFile	oldNtSetSecurityObject	oldNtSetValueKey
oldNtSetVolumeInformationFile	oldNtUnloadDriver	oldNtUnlockFile
oldNtUnmapViewOfSection	oldNtWaitForSingleObject	oldNtFlushInstructionCache
oldNtQueryInformationProcess	oldNtSetInformationProcess	oldNtAlertThread
oldNtCallbackReturn	oldNtGetContextThread	oldNtAlertResumeThread
oldNtContinue	oldNtImpersonateThread	oldNtRegisterThreadTerminatePort
oldNtSuspendThread	oldNtTerminateThread	oldNtOpenMutant
oldNtQueryMutant	oldNtReleaseMutant	oldNtSetTimerResolution
oldNtSetSystemTime	oldNtQueryTimerResolution	oldNtQuerySystemTime
oldNtQueryPerformanceCounter	oldNtLockFile	oldNtOpenEvent
oldNtQueryInformationThread	oldNtQueryDirectoryFile	oldNtQueryEaFile
oldNtSetInformationThread	oldNtAccessCheckByTypeAndAuditAlarm	oldNtCreateEvent
oldNtCreateFile	oldNtDeleteFile	oldNtFlushVirtualMemory
oldNtFreeVirtualMemory	oldNtLockVirtualMemory	oldNtProtectVirtualMemory
oldNtUnlockVirtualMemory	oldNtReadVirtualMemory	oldNtWriteVirtualMemory
oldNtReadFile	oldNtWriteFile	oldNtWriteRequestData
oldNtCreatePort	oldNtImpersonateClientOfPort	oldNtListenPort
oldNtQueryInformationPort	oldNtRequestPort	oldNtAlpcAcceptConnectPort
oldNtAlpcConnectPort	oldNtAlpcCreatePort	oldNtAlpcCreatePortSection
oldNtAlpcDeleteResourceReserve	oldNtAlpcDisconnectPort	oldNtReplyWaitReceivePortEx
oldNtPrivilegeCheck	oldNtAlpcOpenSenderProcess	oldNtAlpcQueryInformation
oldNtAreMappedFilesTheSame	oldNtAssignProcessToJobObject	oldNtCancelSynchronousIoFile
oldNtCompressKey	oldNtCreateEventPair	oldNtCreateKeyedEvent
oldNtCreateProfile	oldNtCreateSemaphore	oldNtCreateSymbolicLinkObject
oldNtCreateTransactionManager	oldNtDebugContinue	oldNtDeletePrivateNamespace
oldNtDisableLastKnownGood	oldNtDisplayString	oldNtDrawText
oldNtEnumerateDriverEntries	oldNtEnumerateTransactionObject	oldNtGetCurrentProcessorNumber
oldNtGetNlsSectionPtr	oldNtGetPlugPlayEvent	oldNtGetWriteWatch
oldNtImpersonateAnonymousToken	oldNtInitiatePowerAction	oldNtIsProcessInJob
oldNtIsSystemResumeAutomatic	oldNtLoadKey	oldNtLoadKey2
oldNtMakeTemporaryObject	oldNtMapUserPhysicalPagesScatter	oldNtModifyBootEntry
oldNtOpenPrivateNamespace	oldNtOpenResourceManager	oldNtOpenSemaphore
oldNtOpenSession	oldNtPrePrepareEnlistment	oldNtQueryInformationEnlistment
oldNtQueryInformationResourceManager	oldNtQueryInformationTransaction	oldNtQueryInformationWorkerFactory
oldNtReadOnlyEnlistment	oldNtRegisterProtocolAddressInformation	oldNtReplacePartitionUnit
oldNtResetWriteWatch	oldNtResumeProcess	oldNtSaveKeyEx
oldNtSetDefaultLocale	oldNtSetInformationDebugObject	oldNtSetInformationJobObject
oldNtSetInformationResourceManager	oldNtSetInformationTransactionManager	oldNtSetIntervalProfile
oldNtSetSystemPowerState	oldNtSetTimer	oldNtSinglePhaseReject
oldNtVdmControl	oldNtWaitLowEventPair	

Note: the set contains 155 system calls, and is the largest set to the best of our knowledge.

APPENDIX B
MALWARE

Table B-1. List of the 100 malware samples used in our evaluation.

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

Note: malware are named with their executable file name.

APPENDIX C
BENIGN SOFTWARE

Table C-1. List of the 113 benign software tested in our evaluation.

CPU-bound	CPU-bound	CPU-bound	CPU-bound	CPU-bound	CPU-bound	I/O-bound	I/O-bound	I/O-bound	I/O-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					

Note: software adversely affected by non-intrusive strategies (*) and intrusive strategies (’) are marked. Software tested under different workloads are marked with ().

Table C-2. BEAR’s evaluation contains 26 benign software tested under different workloads.

SPEC CPU2006	Phoronix-test-suite	GNU projects
astar	cyclictest ’	ruby
bzip2 * ’	fs-mark ’	python ’
gromacs	iozone ’	ping
h246ref	multichase ’	sort
lbm		head
mcf		tail
omnetpp		cat
sjeng ’		cp
soplex * ’		gcc
specrand		javac ’
		diff

Note: software marked with * (non-intrusive strategies) and ’ (intrusive strategies) were adversely affected by the unfavorable environment.

REFERENCES

[1] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," University of California, Berkeley, Tech. Rep., 2002.

[2] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 158–168, June 2006.

[3] B. Salamat, A. Gal, and M. Franz, "Reverse stack execution in a multi-variant execution environment," in *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008.

[4] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *USENIX Security*, 2006.

[5] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson, "Security through redundant data diversity," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2008.

[6] PaX Project, "Address space layout randomization," Mar 2003, http://pageexec.virtualave.net/docs/aslr.txt.

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

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

[9] D. A. Holland, A. T. Lim, and M. I. Seltzer, "An architecture a day keeps the hacker away," *SIGARCH Comput. Archit. News*, vol. 33, no. 1, pp. 34–41, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1055626.1055632

[10] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Digest of the Eighth Annual International Symposium on Fault-Tolerant Computing*, 1978, pp. 3–9.

[11] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Rich'e, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 277–290.

[13] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos, "Netcheck: Network diagnoses from blackbox traces," in *USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[14] J. Rasley, E. Gessiou, T. Ohmann, Y. Brun, S. Krishnamurthi, and J. Cappos, "Detecting latent cross-platform api violations," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

[15] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 253–264. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451145

[16] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Workshop of Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[17] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 265–278.

[18] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 81–96. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541986

[19] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 267–283. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685070

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

[21] R. Sun, D. E. Porter, D. Oliveira, and M. Bishop, "The case for less predictable operating system behavior," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/sun

[22] R. Sun, M. Bishop, N. C. Ebner, D. Oliveira, and D. E. Porter, "The Case for Unpredictability and Deception as OS Features," *USENIX ;login*, 2015.

[23] R. Sun, X. Yuan, A. Lee, M. Bishop, D. E. Porter, X. Li, A. Gregio, and D. Oliveira, "The dose makes the poisonleveraging uncertainty for effective malware detection," in *Dependable and Secure Computing, 2017 IEEE Conference on*. IEEE, 2017, pp. 123–130.

[24] R. Sun, M. Botacin, N. Sapountzis, X. Yuan, M. Bishop, D. Porter, X. Li, A. Gregio, and D. Oliveira, "A praise for defensive programming: Leveraging uncertainty for effective malware mitigation," in *arXiv preprint arXiv:1802.02503*, Under Submission.

[25] R. Sun, X. Yuan, P. He, Q. Zhu, A. Chen, A. Gregio, D. Oliveira, and X. Li, "Learning fast and slow: Propedeutica for real-time malware detection," *arXiv preprint arXiv:1712.01145*, Under Submission.

[26] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet, "Specialization tools and techniques for systematic optimization of system software," *ACM Trans. Comput. Syst.*, vol. 19, no. 2, pp. 217–251, May 2001. [Online]. Available: http://doi.acm.org/10.1145/377769.377778

[27] J. P. Sterbenz and P. Kulkarni, "Diverse infrastructure and architecture for datacenter and cloud resilience," in *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*. IEEE, 2013, pp. 1–7.

[28] L. N. Bairavasundaram, S. Sundararaman, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Tolerating file-system mistakes with envyfs," in *Proceedings of the USENIX Annual Technical Conference*, 2009, pp. 7–7.

[29] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden, "Tolerating byzantine faults in transaction processing systems using commit barrier scheduling," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM, 2007, pp. 59–72.

[30] A. Singh, N. Sinha, and N. Agrawal, "Avatars for pennies: Cheap n-version programming for replication," in *Workshop on Hot Topics in System Dependability (HotDep)*, 2010, pp. 1–3.

[31] L. Spitzner, *Honeypots: Tracking Hackers*. Addison Wesley Reading, 2003.

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

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

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

[35] J. Röning, M. Lasko, A. Takanen, and R. Kaksonen, "Protos-systematic approach to eliminate software vulnerabilities," *Invited presentation at Microsoft Research*, 2002.

[36] D. Aitel, "An introduction to spike, the fuzzer creation kit," *Black Hat*, vol. 1, 2002. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt

[37] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, p. 34, 2011. [Online]. Available: https://www.peach.tech/

[38] M. Zalewski, "Online; accessed 17-may-2017. american fuzzy lop." [Online]. Available: lcamtuf.coredump.cx/afl/

[39] R. Swiecki, "Honggfuzz," *Available online at: http://code. google. com/p/honggfuzz*, 2016.

[40] S. Hocevar, "zzufmulti-purpose fuzzer," 2011. [Online]. Available: Availableonlineat:http://caca.zoy.org/wiki/zzuf

[41] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[42] M. Kerrisk, "Lca: The trinity fuzz tester," 2018. [Online]. Available: https://lwn.net/Articles/536173/

[43] D. Jones, "Bugs found by trinity," 2018. [Online]. Available: http://codemonkey.org.uk/projects/trinity/bugs-fixed.php

[44] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on*. IEEE, 1997, pp. 72–79.

[45] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[46] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr., "Enhancing server availability and security through failure-oblivious computing," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251275

[47] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "Assure: Automatic software self-healing using rescue points," *SIGPLAN Not.*, vol. 44, no. 3, pp. 37–48, Mar. 2009. [Online]. Available: http://doi.acm.org/10.1145/1508284.1508250

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

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

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

[51] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, ser. SP '05, 2005, pp. 32–46.

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

[53] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," vol. 5, no. 2. IEEE, 2007.

[54] N. Solutions, "Norman sandbox whitepaper," 2003.

[55] U. Bayer, C. Kruegel, and E. Kirda, "Ttanalyze: A tool for analyzing malware," in *15th European Institute for Computer Antivirus Research (EICAR)*, 2006.

[56] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, November 2007, pp. 116–127.

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

[58] V. Kumar, H. Chauhan, and D. Panwar, "K-means clustering approach to analyze nsl-kdd intrusion detection dataset," *International Journal of Soft Computing and Engineering (IJSCE)*, 2013.

[59] A. S. Abed, T. C. Clancy, and D. S. Levy, "Applying bag of system calls for anomalous behavior detection of applications in linux containers," in *2015 IEEE Globecom Workshops*. IEEE, 2015, pp. 1–5.

[60] A. Mohaisen, O. Alrawi, and M. Mohaisen, "Amal: High-fidelity, behavior-based automated malware analysis and classification," *Computers & Security*, vol. 52, pp. 251 – 266, 2015.

[61] Y. Fan, Y. Ye, and L. Chen, "Malicious sequential pattern mining for automatic malware detection," *Expert Systems with Applications*, vol. 52, pp. 16–25, 2016.

[62] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 1916–1920.

[63] Y. Li, R. Ma, and R. Jiao, "A hybrid malicious code detection method based on deep learning," *International Journal of Security and Its Applications*, vol. 9, no. 5, 2015.

[64] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs," in *Web Intelligence*

Workshops (WIW), IEEE/WIC/ACM International Conference on. IEEE, 2016, pp. 104–111.

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

[66] Bromium, "Bromium end point protection," 2019. [Online]. Available: https://www.bromium.com/

[67] R. Sun, "System calls in perturbation set," 2018. [Online]. Available: https://docs.google.com/document/d/1E7futmb_XisCHJkh2VTJtRxswR_uSqmezYVWSl9Tgyg/edit?usp=sharing

[68] V. Vipindeep and P. Jalote, "List of common bugs and programming practices to avoid them," *Electronic, March*, 2005.

[69] T. R. Foundation, "The r project for statistical computing https://www.r-project.org/," 2019. [Online]. Available: https://www.r-project.org/

[70] F. Gravetter and L. Wallnau, *Statistics for the Behavioral Sciences*, 8th ed. Wadsworth/Thomson Learning, 2009.

[71] "Coreutils - gnu core utilities http://www.gnu.org/software/coreutils/coreutils.html." [Online]. Available: http://www.gnu.org/software/coreutils/coreutils.html

[72] "Spec cpu 2006 https://www.spec.org/cpu2006/." [Online]. Available: https://www.spec.org/cpu2006/

[73] "The phoronix test suite http://www.phoronix-test-suite.com/." [Online]. Available: http://www.phoronix-test-suite.com/

[74] J. H. McDonald, *Handbook of Biological Statistics*, 3rd ed. Sparky House Publishing, 2014.

[75] "Chi square distribution table http://sites.stat.psu.edu/~mga/401/tables/Chi-square-table.pdf, url = http://sites.stat.psu.edu/ mga/401/tables/Chi-square-table.pdf, bdsk-url-1 = http://sites.stat.psu.edu/ mga/401/tables/Chi-square-table.pdf."

[76] "Cramer's v measures of association http://groups.chass.utoronto.ca/pol242/Labs/LM-3A/LM-3A_content.htm, url = , bdsk-url-1 = ."

[77] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and Security Isolation of Library OSes for Multi-Process Applications," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2014, pp. 9:1–9:14.

[78] "CERT Advisories. (http://www.cert.org/advisories)." [Online]. Available: http://www.cert.org/advisories/

[79] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis, "Detecting targeted attacks using shadow honeypots," in *USENIX Security*, 2005, pp. 129–144.

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

[81] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter, "A study of modern linux api usage and compatibility: What to support when you're supporting," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[82] N. S. Agency, "Security-enhanced linux (http://www.nsa.gov/research/selinux/)." [Online]. Available: http://www.nsa.gov/research/selinux/

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

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

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

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

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

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

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

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

[91] A. Calleja, J. Tapiador, and J. Caballero, "A look into 30 years of malware development from a software metrics perspective," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 325–345.

[92] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 377–396.

[93] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07, 2007, pp. 5–14.

[94] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 3548, pp. 174–187, 2005.

[95] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09, 2009, pp. 351–366.

[96] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012, 2012, pp. 122–132.

[97] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.

[98] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9, 2009, pp. 8–11.

[99] S. Revathi and A. Malathi, "A detailed analysis on nsl-kdd dataset using various machine learning techniques for intrusion detection," *International Journal of Engineering Research and Technology. ESRSA Publications*, 2013.

[100] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Machine learning*, vol. 23, no. 1, pp. 69–101, 1996.

[101] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[102] Y. Bengio, Y. LeCun *et al.*, "Scaling learning algorithms towards ai," *Large-scale kernel machines*, vol. 34, no. 5, pp. 1–41, 2007.

[103] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[104] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, "Dl4md: A deep learning framework for intelligent malware detection," in *Proceedings of the International Conference on Data Mining (DMIN)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016, p. 61.

[105] Q. Wang, W. Guo, K. Zhang, A. G. Ororbia II, X. Xing, X. Liu, and C. L. Giles, "Adversary resistant deep neural networks with an application to malware detection," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 1145–1153.

[106] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2016, pp. 137–149.

[107] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[108] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," *arXiv preprint arXiv:1712.03141*, 2017.

[109] I. Arghire, "Windows 7 most hit by wannacry ransomware," http://www.securityweek.com/windows-7-most-hit-wannacry-ransomware, 2017.

[110] M. Russinovich, "Process monitor v3.40," https://technet.microsoft.com/en-us/sysinternals/bb896645.aspx, 2017.

[111] drmemory, "drstrace," http://drmemory.org/strace_for_windows.html, 2017.

[112] A. B. Insung Park, "Core os tools," https://msdn.microsoft.com/en-us/magazine/ee412263.aspx, 2009.

[113] Microsoft, "Windbg logger," https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/logger-and-logviewer, 2017.

[114] R. Sun. Propedeutica driver publicly available at https://github.com/gracesrm/LKM-system-call-fuzzing, year = 2018.

[115] M. Jurczyk. (2017) Ntapi. [Online]. Available: http://j00ru.vexillium.org/syscalls/nt/32/

[116] G. Creech and J. Hu, "Generation of a new ids test dataset: Time to retire the kdd collection," in *2013 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2013, pp. 4487–4492.

[117] CSDMC, "Cybersecurity Data Mining Competition (CSDMC) 2010 ," https://www.azsecure-data.org/other-data.html, 2010.

[118] J. Pfoh, C. Schneider, and C. Eckert, *Nitro: Hardware-Based System Call Tracing for Virtual Machines", bookTitle="Advances in Information and Computer Security: 6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8-10, 2011. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[119] B. Rozenberg, E. Gudes, Y. Elovici, and Y. Fledel, "NewApproach for detecting unknown malicious executables," *Journal of Forensic Research*, vol. 01, no. 3.

[120] H. Xiao and T. Stibor, "A supervised topic transition model for detecting malicious system call sequences," in *Proceedings of the 2011 Workshop on Knowledge Discovery, Modeling and Simulation*, ser. KDMS. New York, NY, USA: ACM, 2011, pp. 23–30.

[121] Anubis, "Analyzing unknown binaries," http://anubis.iseclab.org., 2010.

[122] A. A. Telyatnikov. (2016) DbgPrint Logger. [Online]. Available: https://alter.org.ua/soft/win/dbgdump/

[123] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*. MIT press, 1999.

[124] B. Caillat, B. Gilbert, R. Kemmerer, C. Kruegel, and G. Vigna, "Prison: Tracking process interactions to contain malware," in *IEE 17th High Performance Computing and Communications (HPCC), IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and IEEE 12th International Conference on Embedded Software and Systems (ICESS)*. IEEE, 2015, pp. 1282–1291.

[125] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[126] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[127] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[128] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," *International Conference on Learning Representations*, 2016.

[129] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[130] PyTorch. (2018). [Online]. Available: http://pytorch.org

[131] Rekings. (2018) Security through insecurity. [Online]. Available: https://rekings.org/

[132] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 230–253.

[133] Winautomation. (2017) Powerful desktop automation software. [Online]. Available: http://www.winautomation.com

[134] M. B. Christopher, *PATTERN RECOGNITION AND MACHINE LEARNING*. Springer-Verlag New York, 2016.

[135] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 2016, pp. 785–794.

[136] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," in *European conference on computational learning theory*. Springer, 1995, pp. 23–37.

[137] R. Caruana, N. Karampatziakis, and A. Yessenalina, "An empirical evaluation of supervised learning in high dimensions," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 96–103.

[138] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Security and Privacy (SP)*. IEEE, 2017, pp. 39–57.

[139] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," *arXiv preprint arXiv:1606.04435*, 2016.

[140] W. Yang, D. Kong, T. Xie, and C. A. Gunter, "Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps," in *33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 288–302.

[141] J. W. Stokes, D. Wang, M. Marinescu, M. Marino, and B. Bussone, "Attack and defense of dynamic analysis-based, adversarial neural malware classification models," *arXiv preprint arXiv:1712.05919*, 2017.

[142] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 62–79.

[143] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Adversarial examples on discrete sequences for beating whole-binary malware detection," *arXiv preprint arXiv:1802.04528*, 2018.

[144] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *Security and Privacy (SP)*. IEEE, 2016, pp. 582–597.

[145] M. F. Botacin, P. L. de Geus, and A. R. A. Grégio, "The other guys: automated analysis of marginalized malware," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 87–98, Feb 2018. [Online]. Available: https://doi.org/10.1007/s11416-017-0292-8

[146] K. S. Trivedi, K. Vaidyanathan, and K. Goseva-Popstojanova, "Modeling and analysis of software aging and rejuvenation," in *Simulation Symposium, 2000.(SS 2000) Proceedings. 33rd Annual*. IEEE, 2000, pp. 270–279.

[147] A. Bovenzi, D. Cotroneo, R. Pietrantuono, and S. Russo, "Workload characterization for software aging analysis," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 2011, pp. 240–249.

[148] “A brief guide to mobile ai chips.” [Online]. Available: https://www.theverge.com/2017/10/19/16502538/mobile-ai-chips-apple-google-huawei-qualcomm

BIOGRAPHICAL SKETCH

Ruimin Sun was a Ph.D. student in the Department of Electrical and Computer Engineering, at the University of Florida. She worked in the Florida Institute of Cybersecurity (FICS). Her research interests are security and reliability in ubiquitous systems (desktop, mobile, IoT/CPS), and machine learning / deep learning based performance analysis. She did an internship in VMware working on machine learning research during the summer 2018. She got her M.S. degree from the Department of Electrical and Computer Engineering at the University of Florida, and her B.S. degree from the Instrument Science and Engineering Department at Southeast University, China.