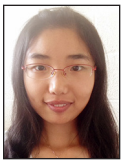
# The Case for Unpredictability and Deception as OS Features

RUIMIN SUN, MATT BISHOP, NATALIE C. EBNER, DANIELA OLIVEIRA, AND DONALD E. PORTER



Ruimin Sun is a first year PhD student in the Department of Electrical and Computer Engineering at the University of Florida. Her research interest lies in operating system security and software vulnerabilities. She's under the direction of Dr. Daniela Oliveira. gracesrm@ufl.edu

Matt Bishop is a Professor in the Department of Computer Science at the University of California, Davis. He does research in many areas of computer security, including data sanitization, vulnerabilities analysis, attribution, the insider problem, and computer security education. mabishop@ucdavis.edu

Natalie C. Ebner is an Assistant Professor in the Department of Psychology and adjunct faculty in the Department of Aging and Geriatric Research at the University of Florida in Gainesville, Florida. Her research adopts an aging perspective on affect and cognition. She conducts experimental research using a multi-methods approach that integrates introspective, behavioral, and neurobiological data. natalie.ebner@ufl.edu

The conventional wisdom is that OS APIs should behave predictably, facilitating software development. From a system security perspective, this predictability creates a disproportionate advantage for attackers. Could making OSes behave unpredictably create a disproportionate advantage for system defenders, significantly increasing the effort required to create malware and launch attacks without too much inconvenience for "good" software? This article explores the potential benefits and challenges of unpredictable and deceptive OS behavior, including preliminary measurements of the relative robustness of malware and production software to unpredictable behavior. We describe Chameleon, an ongoing project to implement OS behavior on a spectrum of unpredictability and deceptiveness.

## Introduction

The art of deception has been successfully used in warfare for thousands of years. Strategists such as Sun Tzu, Julius Caesar, and Napoleon Bonaparte advocated the use of unpredictability and deception in conflicts as a way to confuse and stall the enemy, sap their morale, and decrease their maneuverability. A "holy grail" for system security is to put system defenders in a situation with more options than the attacker.

Unfortunately, current systems are in the exact opposite situation. System defenses generally do not adapt well to new conditions, whereas motivated attackers have effectively unlimited time and resources to find and exploit weaknesses in computer systems.

This situation is rooted in the fact that *predictability* is a first-class system design goal. Predictability simplifies application engineering and usability issues, such as compatibility among different versions of the system. The downside of predictability is a computer system monoculture [1], where vulnerabilities become reliably exploitable on all systems of the same type. With so few operating system kernels, libc implementations, or language runtimes deployed in practice, any predictable exploit applies to a significant fraction of computers in the world.

### *The Need for Unpredictability*

At the system level, approaches to unpredictability generally involve limited randomness. For example, address space layout randomization (ASLR) randomizes the placement of pages of a program in memory during execution. An attack relying on a buffer overflow causing a branch to a library function or gadget will fail, as the address of that target will vary among instances of an operating system. But this randomization is often insufficient. In a recent paper, Bittau et al. [2] demonstrated how, even without specific knowledge of the address space layout randomization (ASLR) scheme of a Web server, an attacker can quickly identify and exploit portions of the address space that are insufficiently random.

Daniela Oliveira is an Associate Professor in the Department of Electrical and Computer Engineering at the University of Florida. Her main research interest is interdisciplinary computer security, where she employs successful ideas from other fields to make computer systems more secure. Her current research interests include employing biology and warfare strategies to protect operating systems. She is also interested in understanding the nature of software vulnerabilities and social engineering attacks. daniela@ece.ufl.edu

Donald E. Porter is an Assistant Professor of computer science at Stony Brook University in Stony Brook, New York. His research aims to improve computer system efficiency and security. In addition to work in system security, recent projects have developed lightweight guest operating systems for virtual environments as well as efficient data structures for caching and persistent storage. porter@cs.stonybrook.edu

Although fixes to ASLR may mitigate this specific attack, this attack shows that *variation without unpredictability is not enough.* Unpredictability by half-measure leaves sufficient residual certainty that allows adversaries to craft reliable attacks even across multiple, differently randomized instances of the system.

Strategies for less predictable operating systems are constrained by concerns for efficiency and reliability. Yet consider what "efficient" and "reliable" mean for an operating system. An operating system's job is to manage tasks that the system is authorized to run, where "authorized" means "in conformance with a security policy." For *un*authorized tasks, such as those an attacker would execute to exploit vulnerabilities or otherwise misuse a system, the operating system should be as inefficient and unreliable as possible. So for "good" users and uses, the operating system should work predictably, but for "bad" users or uses, the system should be unpredictable. The latter case challenges efficiency and reliability. An extension is a *spectrum* of predictability, where the less that actions conform to the security policy, the more unpredictable the results of those actions should be.

### *Software Diversity*

One specific, limited form of unpredictability is diversity. 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 inputs. Most fault-tolerant system designs require sufficient software diversity that faults are independent and can be masked by voting or Byzantine protocols. In practice, the barrier to implementing multiple, complete, monolithic OSes has been insurmountable.

One insight of this work is that diversifying the system implementation becomes easier as more of the system is moved to user space. Several research systems have demonstrated the value of pushing more system-level functionality into user-level libraries, such as moving I/O into user space for higher performance [3] or to reduce virtualization overheads for a single application [4]. Our vision is to mix-and-match different implementations of different components, such that one can run many instances of an application, such as a Web server, and only a few instances will share the same combinations of vulnerabilities. When the implementation effort is smaller and well defined, a small group of developers could easily generate dozens of functional implementations of each subsystem.

Application robustness can also be improved when system-level diversity is incorporated into the development and testing process. Even within POSIX, mature, portable software packages already handle considerable variations in system call behavior. Most of this maturity is the product of labor-intensive testing and bug reports across many platforms over a long period. Rather than require a software developer to manually test the software on multiple platforms, a spectrum-behavior OS would allow developers to more easily test software robustness, running the same test suite against different operating system behaviors.

### *Consistent versus Inconsistent Deception*

Deception has been used in cyberdefense to a limited extent, primarily via *consistent deception* strategies, such as honeypots or honeynets. Consistent deception strategies make the deceiver's system appear as indistinguishable as possible from a production system. This means the deceptive system is just as predictable as the system it is impersonating. The idea of *inconsistent deception* [5], on the other hand, forgoes the need to project a false reality and instead creates an environment laden with inconsistencies designed to keep the attacker from figuring out characteristics of the real system. So long as the attacker is confused and fails to learn anything of value, the deception is successful, even more so if the attacker desists.

Iago attacks [6] are a good example of how inconsistent deception might work in practice. An Iago attack occurs when an untrusted system attacks a trusted program by returning system call results that the trusted program cannot robustly guard against, ultimately causing the trusted program to violate its security goals. We believe similar techniques can be employed for active system defense.

## Unpredictability on Malware

We performed a case study on common malware, showing that malware can be quite sensitive to relatively minor misbehavior by the operating system. We used `ptrace` to alter the information returned by system calls invoked by a keylogger and botnet, introducing unpredictable behavior into their execution. In these cases, the malware ran without crashing, but some I/O were corrupted. Most I/O corruptions were within the specification of the network or potential storage failure modes; a robust application would detect most issues with end-to-end checks such as checksums or, in other cases, checks designed to shield against a malicious OS, such as MAC checks on an encrypted socket.

We selected candidate system calls for spectrum behavior based on analysis of system call behavior of benign processes and malware. We compared the system call patterns of 39 benign applications from SourceForge to 86 malware samples for Linux, including 17 back doors, 20 general exploits, 24 Trojan horses, and 25 viruses. We found that malware invokes a system call set that is smaller than benign software: approximately 50 different system calls.

In selecting strategies for spectrum behavior, our aim is to perturb system calls that harm malware, yet allow benign code to run. We found that a few system calls are critical to process start-up and execution, and cannot be easily varied; most other cases lead to non-fatal deviations. For instance, decreasing the length of a `write()` will cause a keylogger to lose keystrokes, silencing a `send()` might cause a process sending an email to fail, and extending the time of a `nanosleep()` will just slow down a process. We try to balance risks to benign processes with harm to malware through an experimentally determined *unpredictability threshold*, which bounds the amount of unexpected variation in system call behavior.

We studied the following strategies for spectrum behavior:

**Strategy 1: Silence the system call.** We immediately return a fabricated value upon system call invocation. This strategy only succeeds when subsequent system calls are not highly dependent on the silenced action. For example, this strategy worked for `read()` and `write()` but not on `open()`, where a subsequent `read()` or `write()` would fail.

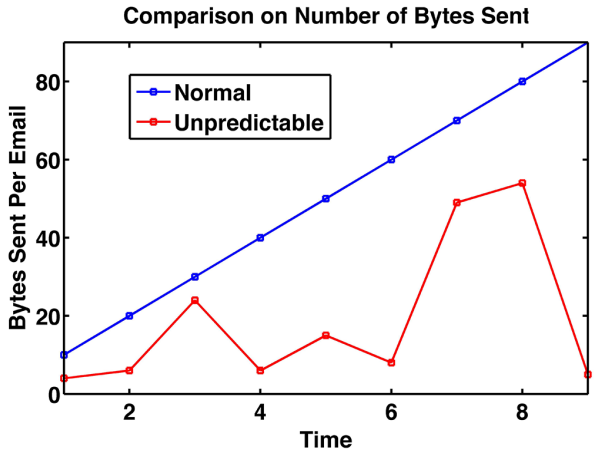



**Figure 1:** Comparison of email bytes sent from bots in normal and unpredictable environments

**Strategy 2: Change buffer bytes.** We randomly change some bytes or shorten the length of a buffer passed to a system call, such as `read()`, `write()`, `send()`, and `recv()`.

This strategy corrupts execution of some scripts, and it can frustrate attempts to read or exfiltrate sensitive data.

**Strategy 3: Add more wait time.** The goal of this strategy is to slow down a questionable process, such as rate-limiting network attacks. We randomly increase the time a `nanosleep()` call yields the CPU.

**Strategy 4: Change file offset.** This approach simulates file corruption by randomly changing the offset in a file descriptor between `read()`s and `write()`s.

We first applied unpredictability to the Linux Keylogger (LKL, http://sourceforge.net/projects/lkl/), a user-space keylogger, using strategies 1, 2, and 4. The keylogger not only lost valid keystrokes but also had some noise data added to the log file.

Next we applied unpredictability to the BotNET (http://sourceforge.net/projects/botnet/) malware, which is mainly a communication library for the IRC protocol that was refined to add spam and SYN-flood capabilities. We used the IRC client platform `irssi` to configure the botnet architecture with a bot herder, bots, and victims. The unpredictable strategies were applied to one of the bots.

We first tested commands that successfully reached the bot, such as `adduser`, `deluser`, `list`, `access`, `memo`, `sendmail`, and `part`. The bot reads commands one byte at a time, and one lost byte will cause a command to fail. Randomly silencing a subset of `read()` system calls in our unpredictable environment results in losing 40% of the commands from the bot herder.

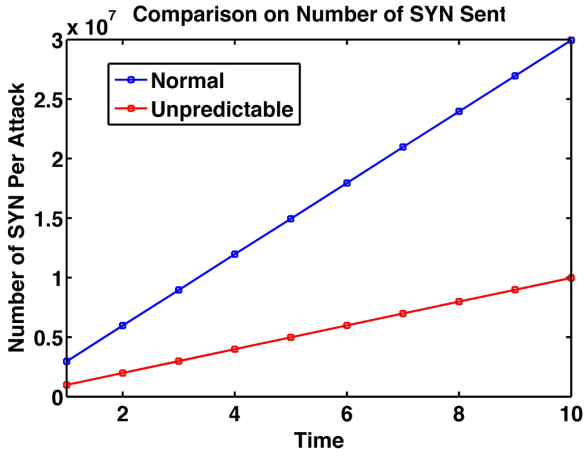



**Figure 2:** Comparison of SYN-flood attacks in normal and unpredictable environments. Unpredictability can increase the DDoS resource requirements.

We measured the impact of the unpredictable environment on the ability of the bot to send spam emails, shown in Figure 1. In the normal environment, nine emails varying in length from 10 to 90 bytes were successfully sent. In the unpredictable environment, only partial random bytes were sent out by arbitrarily reducing the buffer size of `send()` in the bot process. In the case of a spam bot, truncated emails will streamline the filtering process, not only for automatic filters, but also for the end users.

We also performed a SYN-flood attack to analyze the effectiveness of the unpredictable environment in mitigating DDoS attacks. In a standard environment, one client can bring down a server in one minute with SYN packets. When we set the unpredictability threshold to 70% and applied strategies 1 and 3, the rate of SYN packets arriving at the victim server decreased (Figure 2), requiring two additional bots to achieve the same outcome.

Preliminary tests with Thunderbird, Firefox, and Skype running in the unpredictable environment showed that these applications can run normally most of the time, occasionally showing warnings, and with some functionality temporarily unavailable.

A challenge is to dial this behavior in to minimize harm to benign, but not whitelisted, applications while frustrating potentially malicious code.

## Spectrum-Behavior OS

We are building Chameleon, an operating system combining inconsistent and consistent deception with software diversity for active defense of computer systems and herd protection. Chameleon provides three distinct environments for process execution (Figure 3): (1) a diverse environment for whitelisted processes, (2) an unpredictable environment for unknown or suspicious processes (inconsistent deception), and (3) a consistently deceptive environment for malicious processes. Our

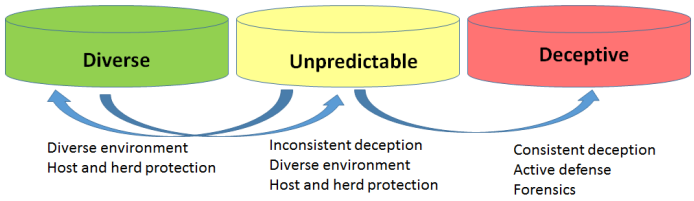



**Figure 3:** Chameleon can transition processes among three operating modes: *diverse*, to protect benign software; *unpredictable*, to disturb unknown software; and *deceptive*, to analyze likely malware.

HotOS '15 paper [7] provides a longer discussion of these issues, as well as a more extensive discussion of prior work on unpredictability and deception as tools for system security.

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 [8], except that Chameleon doesn't run the versions in parallel but, rather, diversifies them across processes. Our 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 [4], 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* environment, where a subset of the system calls are modified or silenced. 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. This environment will be sandboxed, files will be honeyfiles, and external connections 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 them to training sets of known good and malicious code. Based on its behavior, a process can migrate across environments.

## What About the Computer User?

Sacrificing predictability will introduce new, but tractable, research questions, especially around usability. For example, a user who installs a new game with a potential Trojan horse will be tempted to simply whitelist the game if it isn't playable. We believe unpredictability can be adjusted dynamically to avoid interfering with desirable behavior, potentially with user feedback.

We envision Chameleon's architecture adopted in desktop computers for end users. This will allow a common group of whitelisted applications such as browsers or office software to run unperturbed and a suspicious application to be quarantined by Chameleon.

For example, consider Bob, 72, living in a retirement community in Florida. Bob is not computer savvy and tends to click links from spear-phishing emails, which might install malware in his computer. This malware will engage in later attacks compromising other machines and performing DoS attacks in critical infrastructure. Bob never notices malware running in his computer because the malware becomes active only after 1 a.m.

With Chameleon, Bob continues to browse for news, work on documents from his community homeowner association, or Skype with family without problems; these applications are whitelisted, running in the diverse environment. The diverse environment protects whitelisted applications by reducing the likelihood of their being exploited. Further, if Bob downloads a game that also includes a botnet, the unpredictable environment may cause the game to seem poorly designed, the visual images showing some glitches here and there, but Bob's credentials will be safe. Further, the botnet, which Bob will never notice, will fail to operate as the attacker wishes.

Part of the evaluation of Chameleon's success or failure will include usability studies. Our hypothesis is that Chameleon can strike a long-sought balance that preserves usability for desirable uses but thwarts significantly more compromises without frustrating users to the point of disabling the security measure.

## Conclusions

Today's systems are designed to be predictable, and this predictability benefits attackers more than software developers or cybersecurity defenders. This leads us to have the worst of both worlds: rather simple attacks work, and both research and industry are moving towards models of mutual distrust between applications and the operating system [9, 10].

If applications will trust the operating system less in the future, why not leverage this as a way to make malware and attacks harder to write? If successful, sacrificing predictable behavior can finally give systems an edge over one of the primary sources of computer compromises: malware installed by unwitting users.

### *Acknowledgments*

We thank the anonymous HotOS reviewers, Nick Nikiforakis, Michalis Polychronakis, and Chia-Che Tsai for insightful comments on earlier drafts of this paper.

This research is supported in part by NSF grants CNS-1149730, SES-1450624, CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, CNS-1408695. It is also supported by grants OCI-1246061 and DUE-1344369.

***References***

[1] 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.

[2] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking Blind," *in 2014 IEEE Symposium on Security and Privacy (SP),* May 2014, pp. 227–242.

[3] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The Operating System Is the Control Plane," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '14),* 2014, pp. 1–16.

[4] 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.

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

[6] 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.

[7] R. Sun, D. E. Porter, D. Oliveira, and M. Bishop, "The Case for Less Predictable Operating System Behavior," in *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS XV),* 2015.

[8] 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.

[9] 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.

[10] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, 2014, pp. 267–283.